# Testing After Unit Tests

Quetzal Bradley
Connected Systems Architecture

# Why do we test?

# Why do we test?

- To try and predict whether our customers will be satisfied
  - Ship a finite number of features
  - Infinite possible tests for that finite feature list
  - Theory: Choose least number (cost) of tests that maximizes correlation between “tests pass 100%” and “customers satisfied”

# Is 100% Code Coverage Enough?

- What we know:
  - It is possible to call the code and have a correct result
- What we don't know:
  - Everything else!  We know very little about the quality of the code.
- Code coverage is a negative metric.
  - Lack of coverage is good information on untested code

# Testing the State Highways try #1

- Writing unit tests for code coverage
  - Take one 3-cylinder hatchback and run down every road at least once

# Woops!

# A code example

```
static int StringLength(string input)
{
    return input.Length;
}

static void Main(string[] args)
{
    // tests, 100% coverage!
    Assert.AreEqual(4, "four");
}
```

# Woops!

```
static void Main(string[] args)
{
    // call “perfectly” tested function:
    int length = StringLength(null);
}
```

# Testing the state highways try #2

- Prioritize the roads
  - Try to hit every road, but focus multiple attention on the highways and primary arteries
- Diversify the data
  - Send every kind of vehicle and configuration down the most important roads
- Feedback and iterate
  - Are we finding issues?

# What is a viable strategy?

- Use unit tests with high coverage for breadth
- Target high value states by using at least the following techniques:
  - Sample tests
  - Data oriented primary functionality tests
  - Stress
  - Integration tests
  - Manual testing
- Feedback and iterate

# Sample Tests

- Samples are simulations of the code our customers will write
  - Highly correlated with customer pain
- Writing the samples is great
- Running them is better

# Data oriented primary functionality tests (depth)

- Running the same code, but different data
- Randomness, unpredictability, or huge data sets are helpful
- Oracles are hard
  - The oracle is the code that knows whether a result is the correct result or an incorrect result

# Stress (breadth)

- Increase state coverage by perturbing ambient state
- Great for multithreaded but good for single-threaded code as well
  - Running different paths that interact at some point

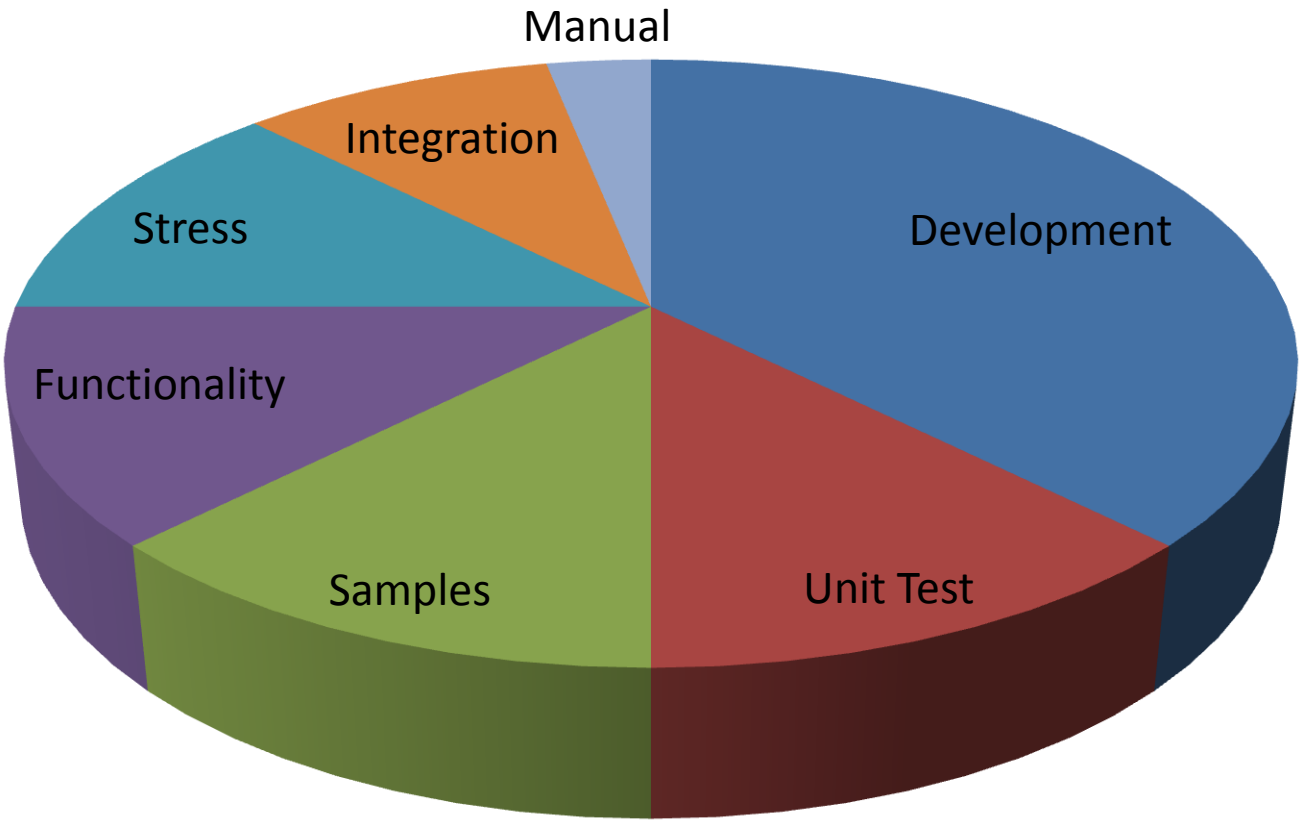
# Integration Tests

- We would like to be sure that every end-to-end customer use succeeds
- The connection between larger pieces
- Hard to write and hard to maintain
  - Large number of steps to desired state
  - When code is changing they are always broken

# Manual Testing

- A small number of manual tests checked by a human infrequently can have a huge ROI
  - Difficult to measure
    - Subtle performance issues (slight lags)
    - Visual effects (flicker)
  - Frequently changing
  - Unpredictable
- Checklists can be boring
  - Mitigation: directed ad-hoc
  - Mitigation: vendors

# Effort expectations



# How much testing is enough?

- Ship a finite number of features
- Infinite possible tests for that finite feature list
- Theory: Choose least number (cost) of tests that maximizes correlation between “tests pass 100%” and “customers satisfied”
- Practice: Cover low hanging fruit, watch signs, intuition, iterate

# Questions?