Unit Testing Strategies

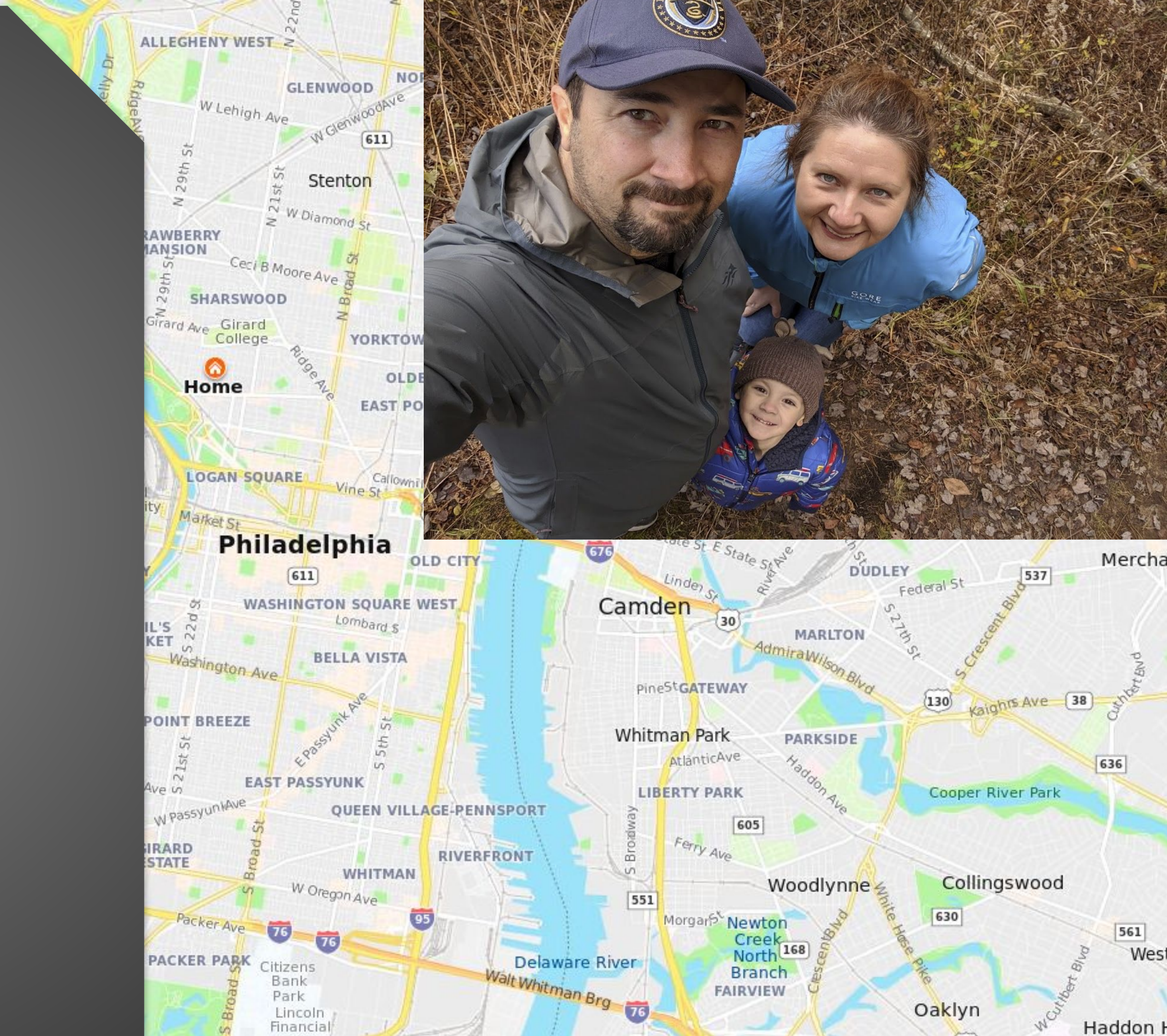
Planning for effective unit testing

About your speaker

Mike Rapa
20+ years in software
B.S. Computer Science @ Thomas Jefferson University
M.E. Software Engineering @ Penn State

Allscripts
Healthcare software and services
10k+ Employees

@mikerapa
github.com/mikerapa



Agenda

Introduction

What are Unit Tests? Why do it?

Failure

Why unit testing processes fail

Preparing for successful unit testing

Doing all the necessary planning to improve your chances of success

What is an automated unit test?
What is not?

Unit test: Code developed for the purpose of testing portions of an application with known inputs and expected outputs.

Integration Test: Testing multiple modules of an application together

Functional Test: Test that validate business requirements

Why have automated unit tests?

Promote sound programming practices

Develop higher quality software

More productive code reviews

Code base that is easier to refactor

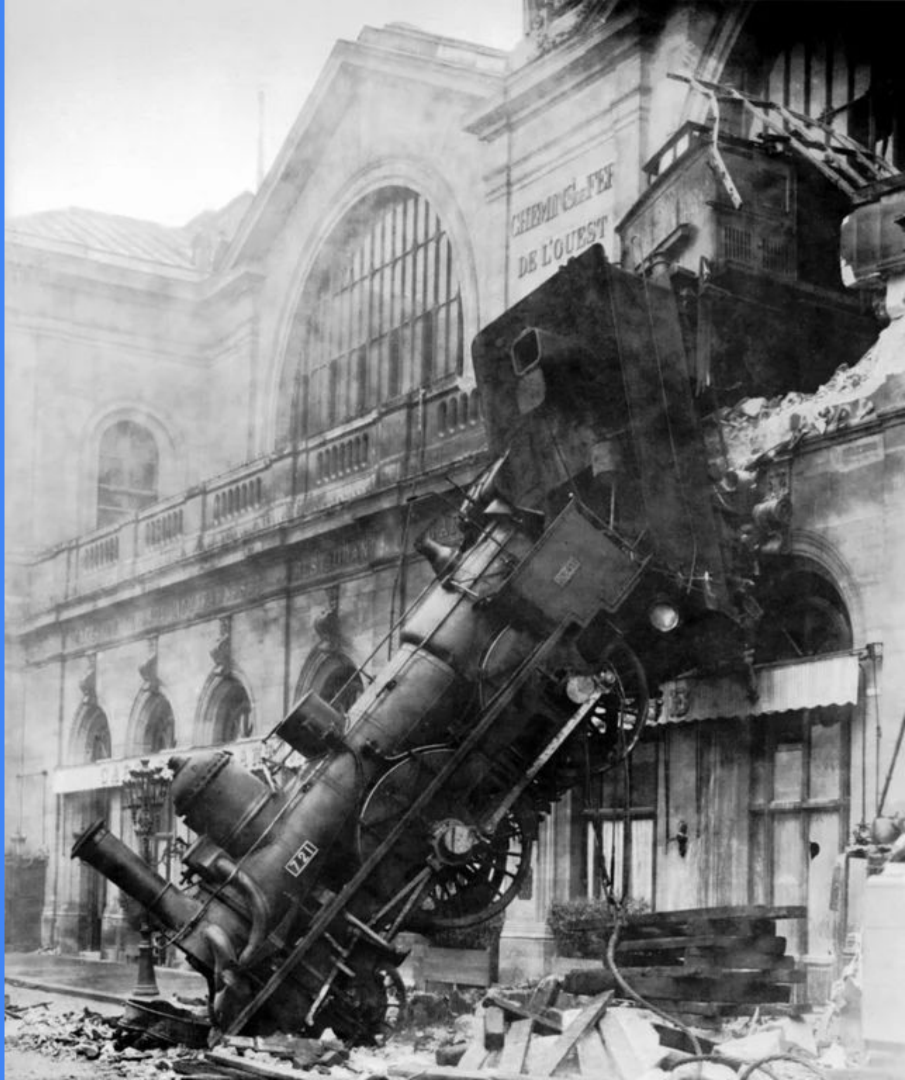
Improves longevity of your code base

Easier to on-board developers/teams

Find bugs when it’s least expensive to fix them

Free your QA/QE staff from repetitive tasks to work on creative testing instead

Failure



Typical Unit Testing Plan

Developers write the unit tests

Having unit tests is good, having more is better

The End

Abandoned Unit Tests

“Tests were commented out each time one broke”

“The tests were bound to a database or some other external service which made them too bridle”

“The effort required to maintain the tests exceeds the perceived value”

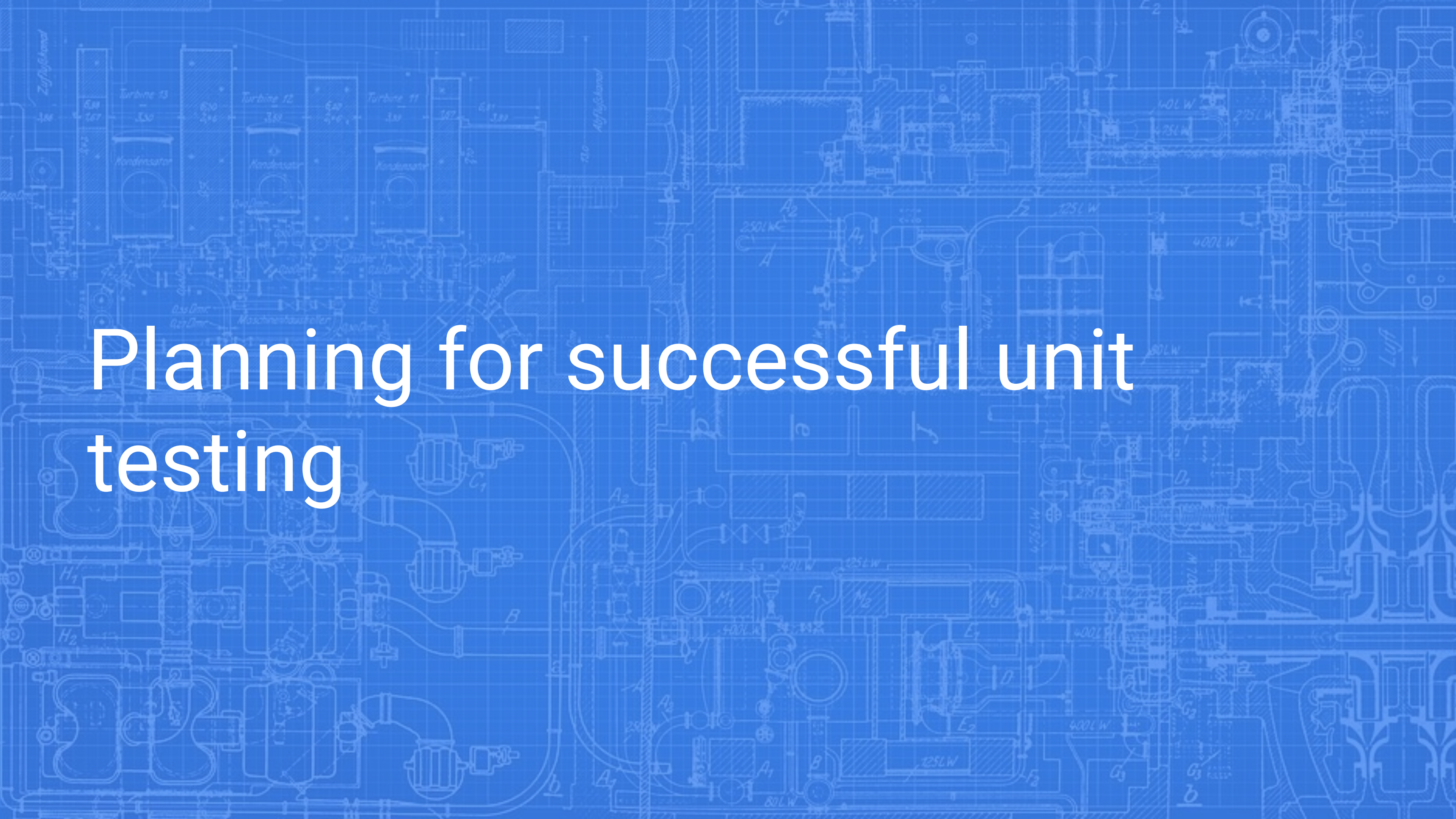
“The developer who worked on those tests is gone”

“Tests weren’t modified to match changes in the requirements”

Abandoned Unit Tests (Continued)

“We break things we don’t test, and test things we don’t break”

“The new leader doesn’t want to waste money developing code that doesn’t ship with the product”



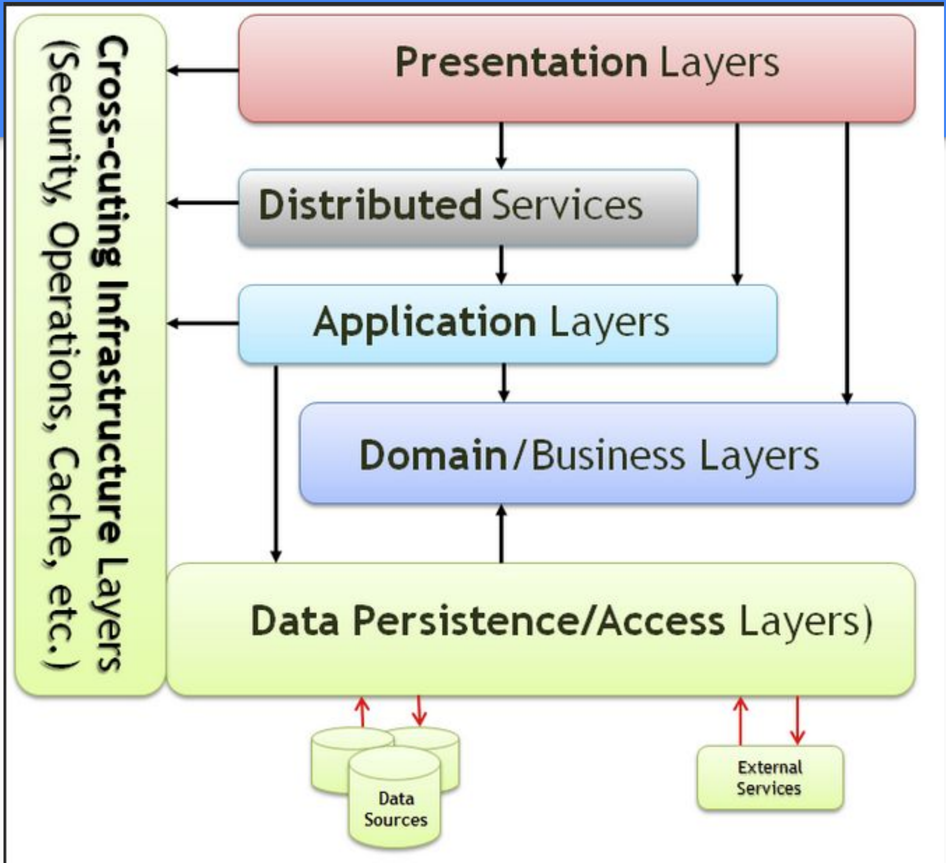
Planning for successful unit testing

Setting boundaries

Decide what should and should not be included in your tests

External data sources?

Which layers of code? All the way up to the UI? All the way down to the database?



Technical Unit Tests

Performance

Memory

Scalability

Full Application Testing

Manual

A tester is running test steps based on written test plans or ad hoc testing

- Microsoft Test Manager
- TestLodge
- qTest

Unit

Code written to test business rules, technical requirements by calling into application code

- JUnit
- NUnit
- MSTest
- Jasmine
- Cucumber/SpecFlow

Performance

Testing the performance of the application by simulating concurrent usage

- WebLoad
- LoadRunner
- JMeter
- LoadNinja

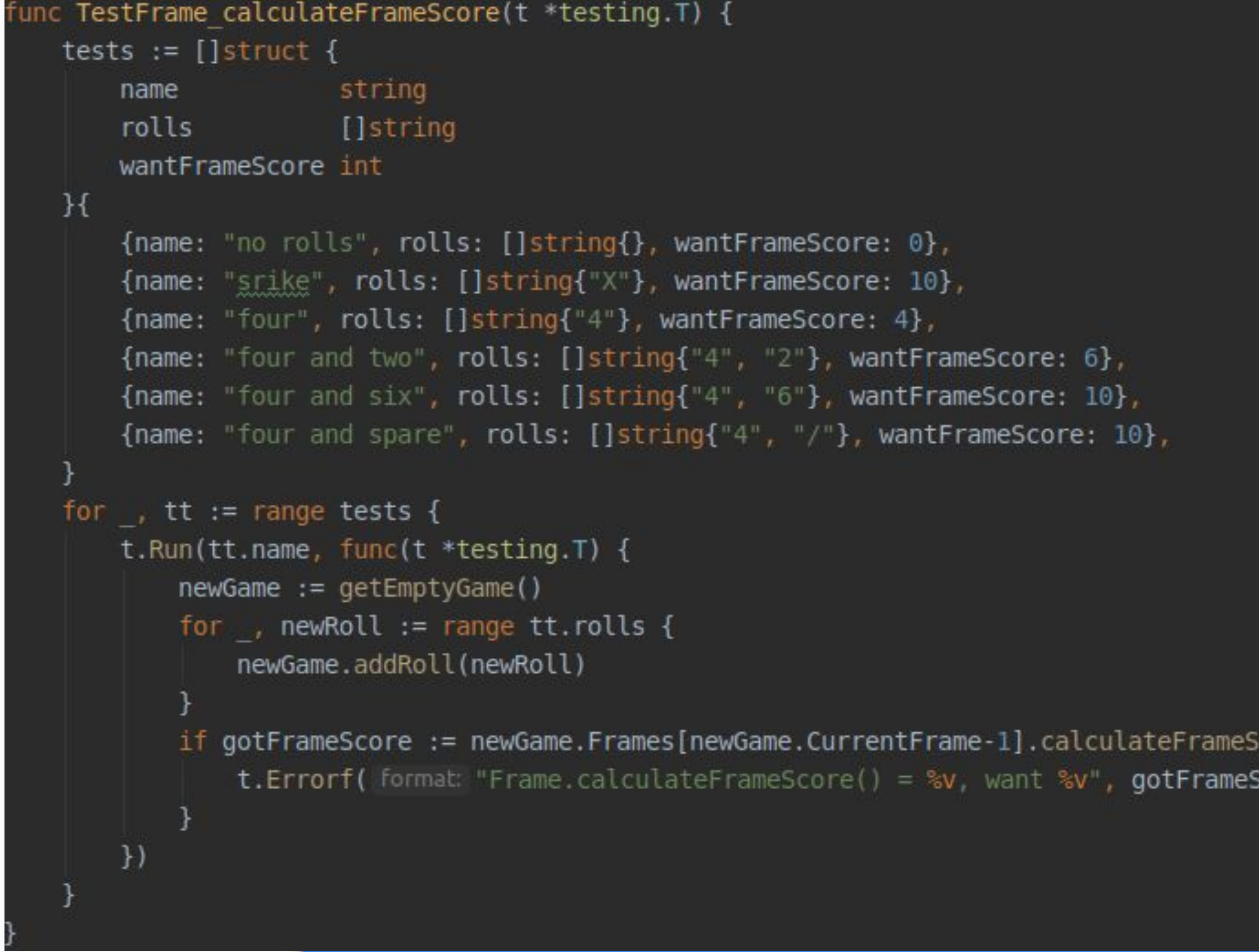
UI Automation

Automation of user activity in the UI

- Selenium
- Coded UI

Who designs the tests?

Everyone with a stake in the quality of your application



```go
func TestFrame_calculateFrameScore(t *testing.T) {
	tests := []struct {
		name          string
		rolls         []string
		wantFrameScore int
	}{
		{name: "no rolls", rolls: []string{}, wantFrameScore: 0},
		{name: "srike", rolls: []string{"X"}, wantFrameScore: 10},
		{name: "four", rolls: []string{"4"}, wantFrameScore: 4},
		{name: "four and two", rolls: []string{"4", "2"}, wantFrameScore: 6},
		{name: "four and six", rolls: []string{"4", "6"}, wantFrameScore: 10},
		{name: "four and spare", rolls: []string{"4", "/"}, wantFrameScore: 10},
	}
	for _, tt := range tests {
		t.Run(tt.name, func(t *testing.T) {
			newGame := getEmptyGame()
			for _, newRoll := range tt.rolls {
				newGame.addRoll(newRoll)
			}
			if gotFrameScore := newGame.Frames[newGame.CurrentFrame-1].calculateFrameS
				t.Errorf( format: "Frame.calculateFrameScore() = %v, want %v", gotFrameS
			}
		})
	}
}
```

Who writes tests and when?

Who?

- Developers only?
- Product designers (if you’re using BDD or abstracted test cases from code)

When?

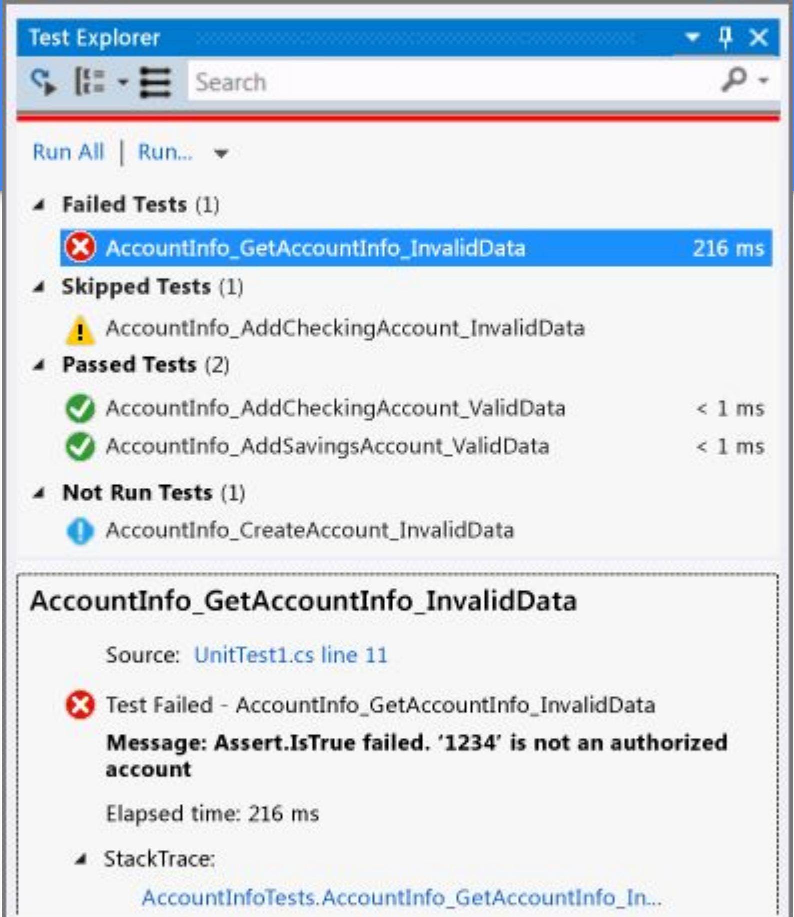
- In sprint?
- Before a feature is developed? Before a bug is fixed?
- Before the product/version ships?
- In response to customer support cases?

Executing the tests

Manually run by developers while making code changes

Executed automatically in CI/CD pipeline

Gating criteria for integrated builds, shipping, acceptance testing



Failed Tests (1)
AccountInfo_GetAccountInfo_InvalidData 216 ms
Skipped Tests (1)
AccountInfo_AddCheckingAccount_InvalidData
Passed Tests (2)
AccountInfo_AddCheckingAccount_ValidData < 1 ms
AccountInfo_AddSavingsAccount_ValidData < 1 ms
Not Run Tests (1)
AccountInfo_CreateAccount_InvalidData

AccountInfo_GetAccountInfo_InvalidData

Source: UnitTest1.cs line 11

Test Failed - AccountInfo_GetAccountInfo_InvalidData

Message: Assert.IsTrue failed. '1234' is not an authorized account

Elapsed time: 216 ms

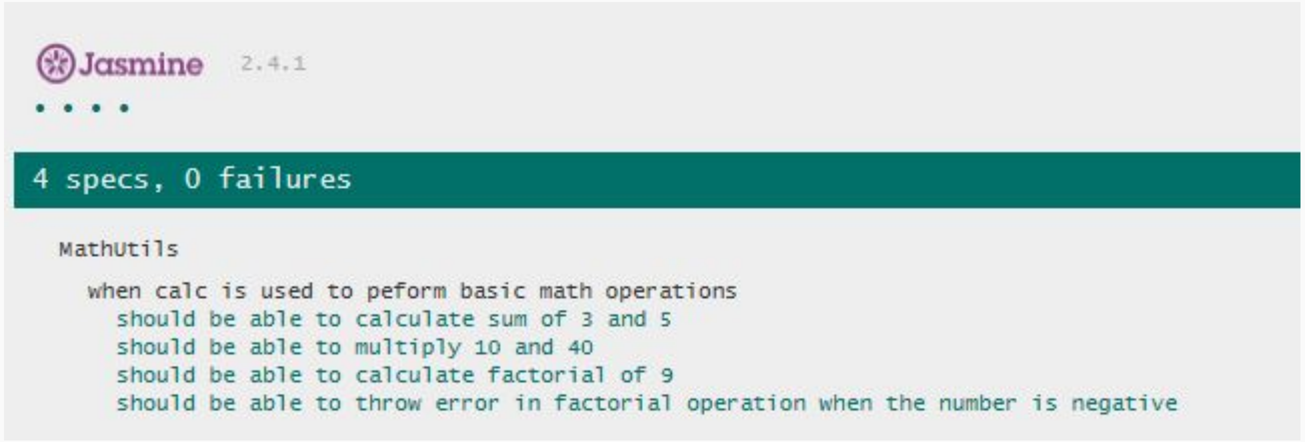
StackTrace:

AccountInfoTests.AccountInfo_GetAccountInfo_In...

Transparency

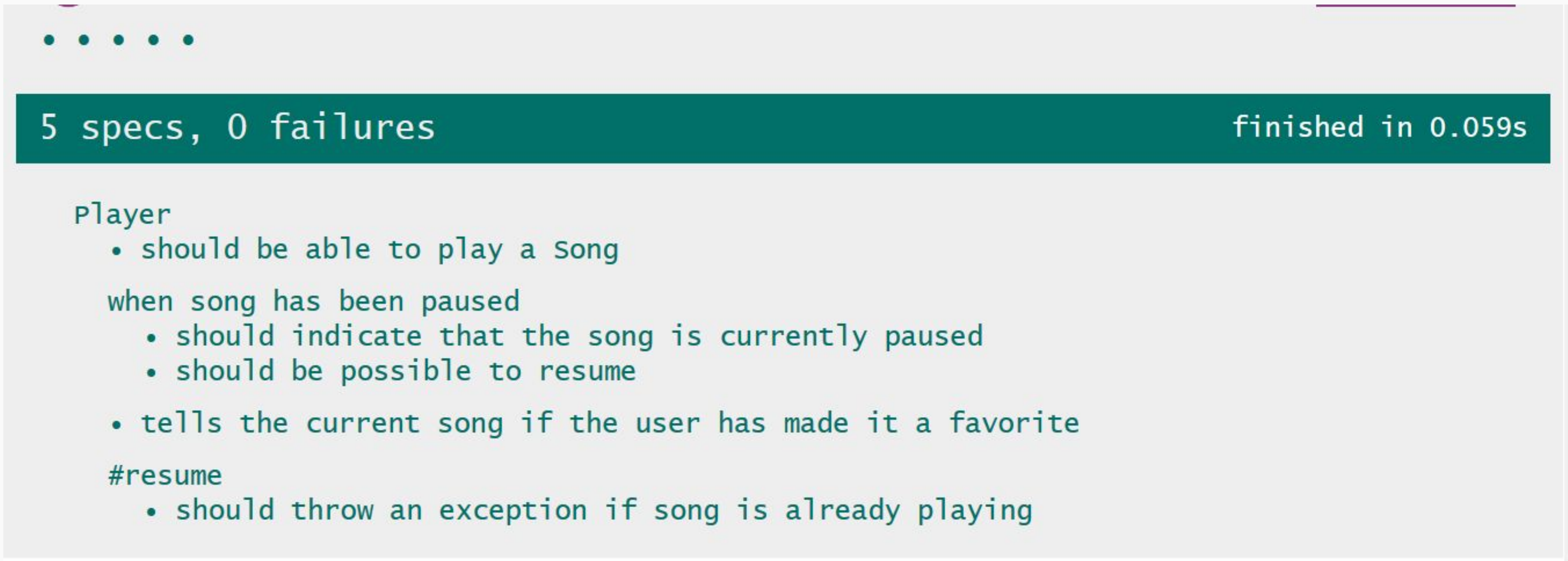
Everyone should be able to see the results

Everyone should be able to understand what is covered



```
• • • •

4 specs, 0 failures

  MathUtils

    when calc is used to peform basic math operations
      should be able to calculate sum of 3 and 5
      should be able to multiply 10 and 40
      should be able to calculate factorial of 9
      should be able to throw error in factorial operation when the number is negative
```

Tests should document your business rules


```
•••••

5 specs, 0 failures                                    finished in 0.059s

  Player
    • should be able to play a Song

    when song has been paused
      • should indicate that the song is currently paused
      • should be possible to resume

    • tells the current song if the user has made it a favorite

    #resume
      • should throw an exception if song is already playing
```

Architecture

Fixtures and mocks should be deliberate and reusable

Architects should design complex fixtures and mocks

Failing to manage the complexity of your test setup basically ensures that your unit tests will be abandoned

Tests with lots of duplicated code are a shortcut to abandonment

Code is code

If you have a coding standard, it applies to your unit testing code

If you have a style guide, it applies to your testing code

If you have code analysis tools, they should inspect your unit testing code

Oversight

Unit tests should be part of your **code review** process

Automated test cases should be **reviewed** by whoever writes the business rules

Creating unit tests should be **represented in stories or tasks** and part of your ‘definition of done’

Demonstrated with features in sprint demos, or handoff demonstrations

Tooling

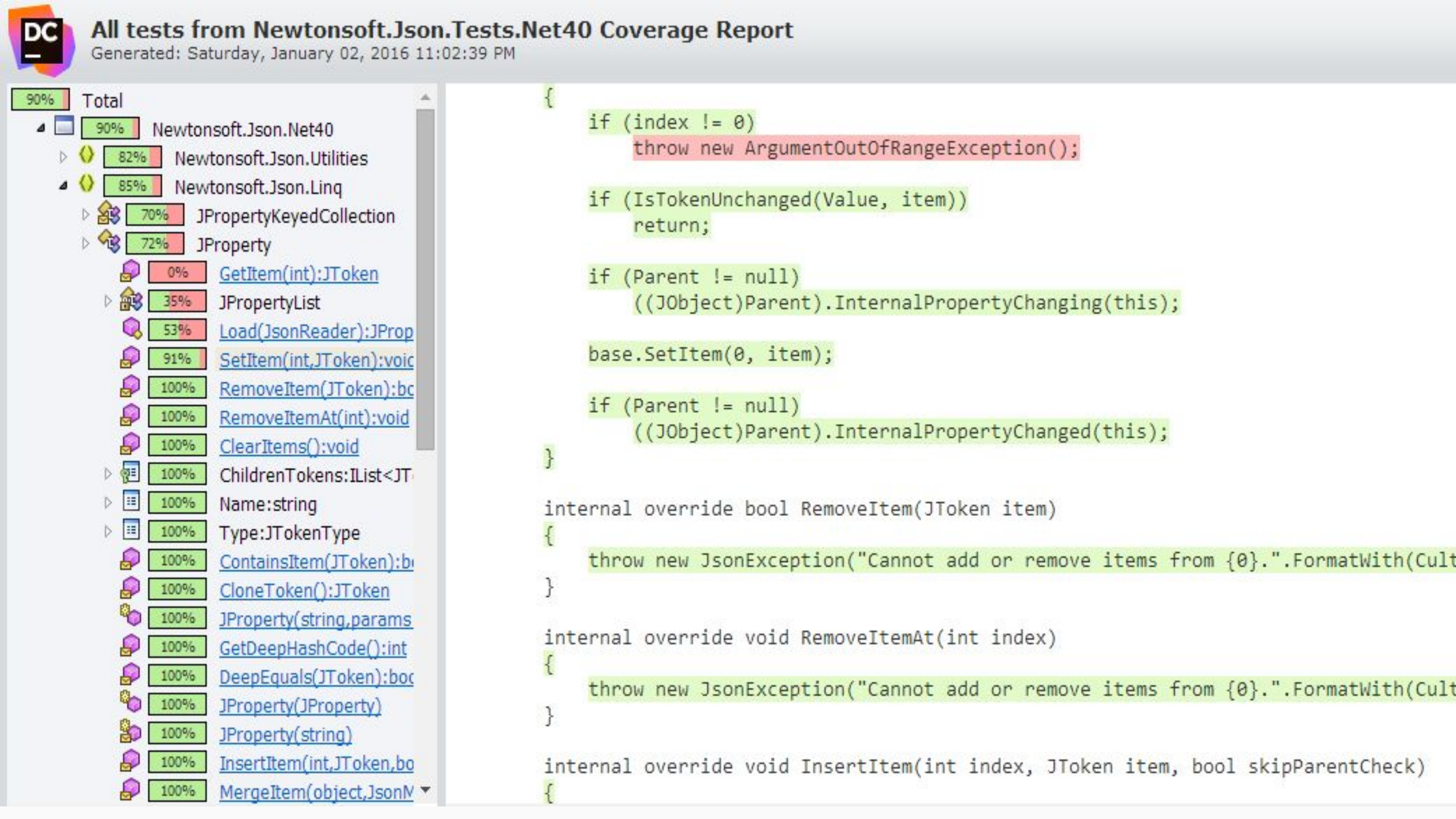
Standardize on:

1. Unit testing framework
2. Mocking library
3. memory/performance testing components
4. CI/CD pipeline

Consistency is more important than choosing the latest trending libraries

Code Coverage

Without code coverage metrics	With code coverage metrics
May be only testing happy paths	Ensure that all code paths are covered
Don’t know how complete your tests are	Recognize unreachable code
	Track progress



All tests from Newtonsoft.Json.Tests.Net40 Coverage Report

Generated: Saturday, January 02, 2016 11:02:39 PM

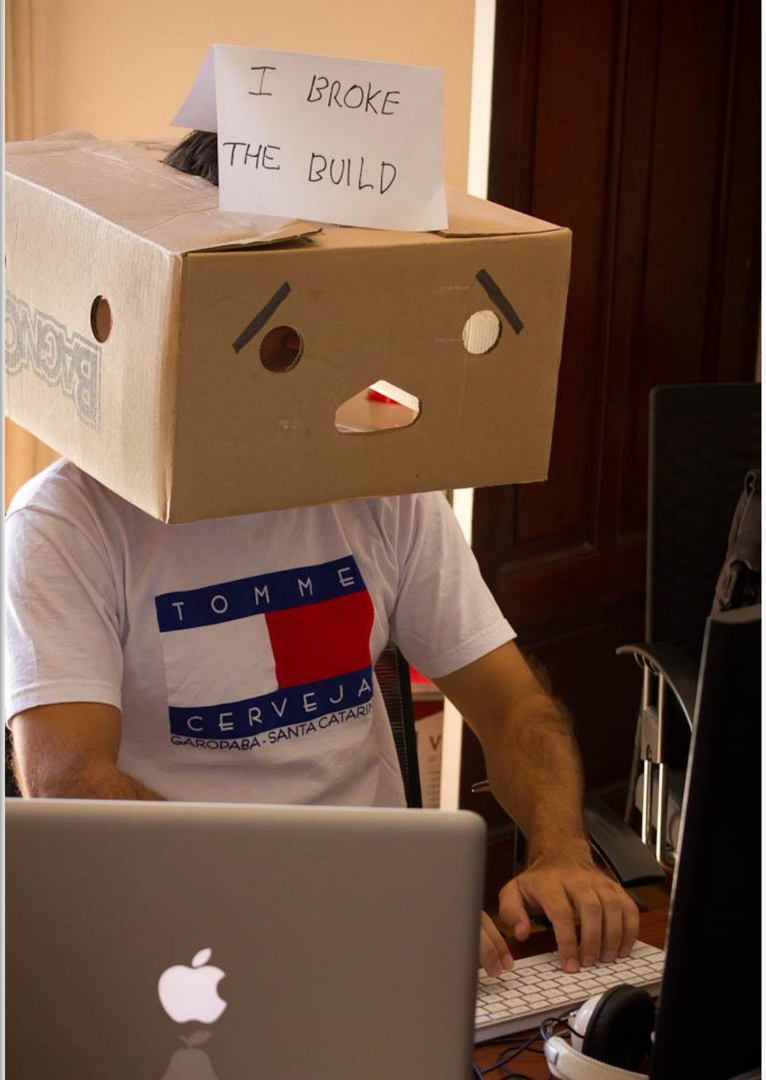
```
90%  Total
  90%  Newtonsoft.Json.Net40
    82%  Newtonsoft.Json.Utilities
    85%  Newtonsoft.Json.Linq
      70%  JPropertyKeyedCollection
      72%  JProperty
        0%    GetItem(int):JToken
        35%   JPropertyList
        53%   Load(JsonReader):JProp
        91%   SetItem(int,JToken):voic
        100%  RemoveItem(JToken):bc
        100%  RemoveItemAt(int):void
        100%  ClearItems():void
        100%  ChildrenTokens:IList<JT
        100%  Name:string
        100%  Type:JTokenType
        100%  ContainsItem(JToken):bı
        100%  CloneToken():JToken
        100%  JProperty(string,params
        100%  GetDeepHashCode():int
        100%  DeepEquals(JToken):boc
        100%  JProperty(JProperty)
        100%  JProperty(string)
        100%  InsertItem(int,JToken,bo
        100%  MergeItem(object,JsonM
```

```
{
    if (index != 0)
        throw new ArgumentOutOfRangeException();

    if (IsTokenUnchanged(Value, item))
        return;

    if (Parent != null)
        ((JObject)Parent).InternalPropertyChanging(this);

    base.SetItem(0, item);

    if (Parent != null)
        ((JObject)Parent).InternalPropertyChanged(this);
}

internal override bool RemoveItem(JToken item)
{
    throw new JsonException("Cannot add or remove items from {0}.".FormatWith(Cult
}

internal override void RemoveItemAt(int index)
{
    throw new JsonException("Cannot add or remove items from {0}.".FormatWith(Cult
}

internal override void InsertItem(int index, JToken item, bool skipParentCheck)
{
```

How unit tests relate to the build

Tests that don't compile, should break the build

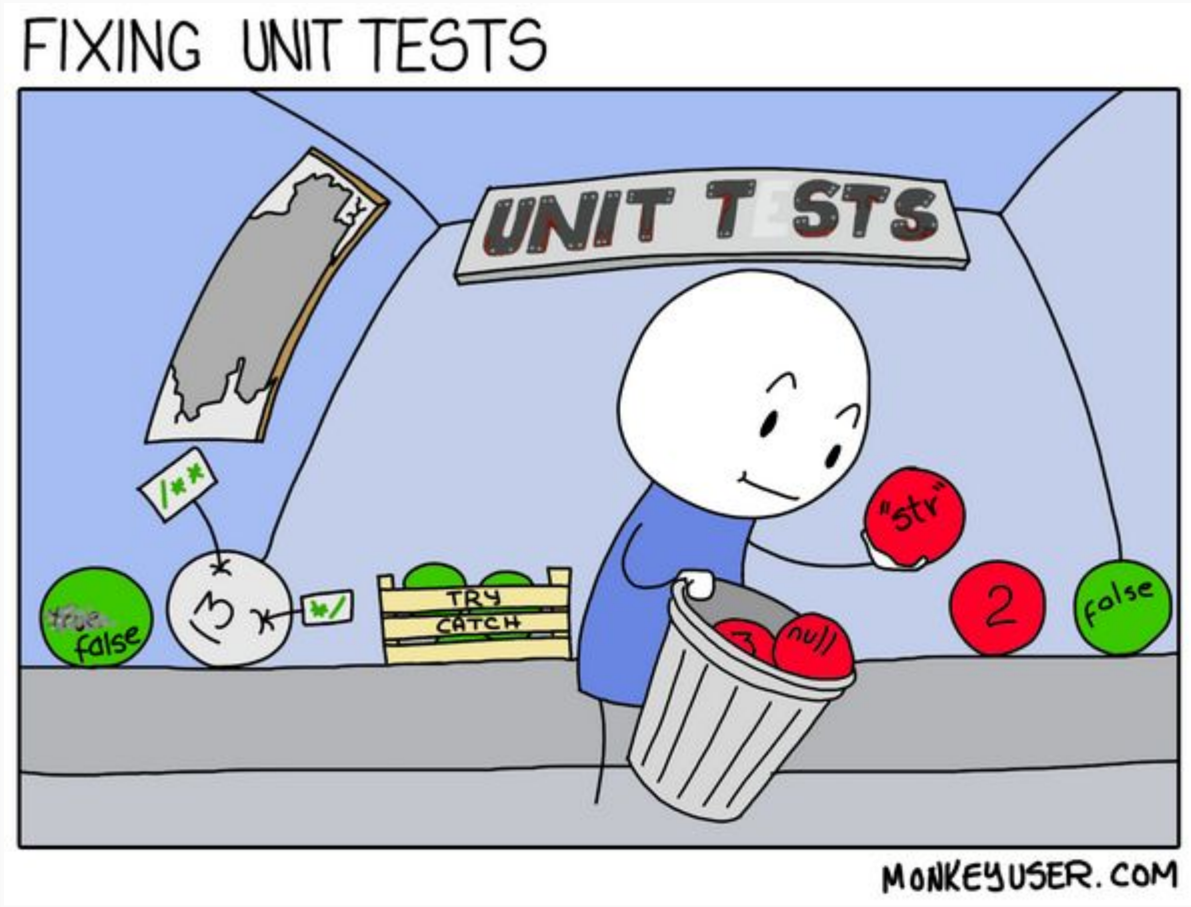
Test results should be part of the build output



Maintaining Unit Tests

Every developer should be able to add tests and fix broken tests

If your application is undergoing a technical uplift, shift in architecture or refactor, your tests should do the same



End of Life

Test should be eliminated when:

- Supplanted by another unit test
- The team decides to replace this test with an automated UI test or some other form of test
- Functional design change makes the test obsolete

Decide who can remove tests. Does the team need to agree?

Plan

Rules and best practices. Enforcement. Oversight.

Scope, tooling, structure,

Roles of developers, architects, testers and designers

Metrics

Lifecycle of your tests

Communication

The team needs to buy in, not just a few people

Document your plan somehow (unit testing specific documentation or integrated into whatever process documentation you have)

Most teams have unit testing skeptics, who think unit tests don’t add value, or threaten someone’s job or requires new learning

Q&A