Metrics & Methodologies for Test Suite Design

Sean Olszewski

June 18, 2018

Hello!

Engineer for Pivotal Labs

Daily TDD Practitioner

Musician & Sound Designer

@__chefski__

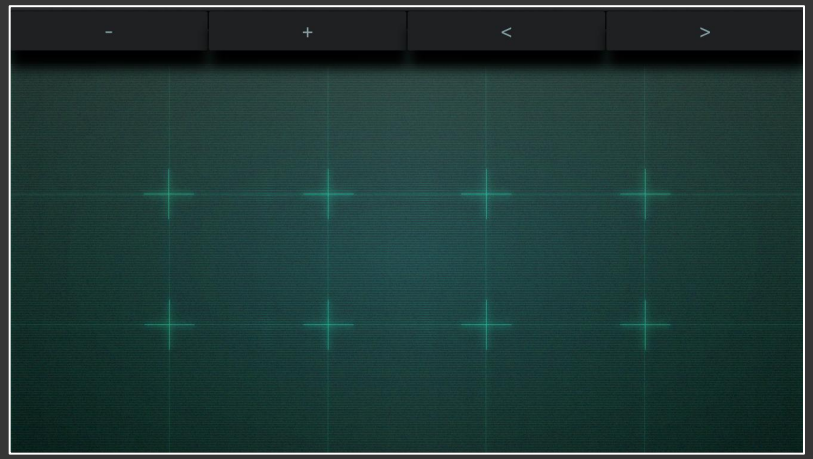
Session Overview

- **My App - Arper**
  Ground the talk in something concrete
- **Test Suite Engineering**
  Talk about ways to engineer effective test suites
- **Review**
  Go over the topics we went through

Arper

Arper

- Music app
- Ambient/drone music
- Entirely test-driven
  - Quick/Nimble
- Under active development



Demo of Arper

Arper - Components

ButtonBankView

Lays out buttons so they can be played. Notifies a delegate that a button is pressed.

MIDINoteMapping

Determines which button corresponds to a note.

AudioEngineManager

Handles note routing into the audio engine. Contains logic for interpreting various controls.

AudioEngine

Handles converting notes into sound.

Test Suite Engineering

Concepts

- **Responsibilities**
  The purpose and benefits of a test suite
- **Metrics**
  Measurable and meaningful details for assessing the efficacy of your test suites
- **Patterns**
  Clearly defined and repeatable ways to code test suites
- **Methodologies**
  Ways that you can use patterns & metrics to engineer an effective test suite

Terminology

- **Test**
  A way to prove something works, usually automatically.
- **Test Subject**
  The component you are interested in proving works.
- **Behavior**
  What a test subject is supposed to do; what we are interested in testing
- **Test Double**
  A component which stands-in for a dependency of the test subject (**mocks**, **spies**, **fakes**, etc)

Responsibilities

The purpose and benefits of a test suite

Proving what you're building is coded correctly.

Showing how your code works by example.

Improving the ability to make changes to your code base.

Create software at a lower cost and a faster rate.

Metrics

Measurable and meaningful details for assessing the efficacy of your test suites

Metrics

Signal-to-Noise Ratio

how clearly a test failure indicates a specific fault or issue in your code base

Maintenance Cost

how much effort must go into keeping a test or test suite effective

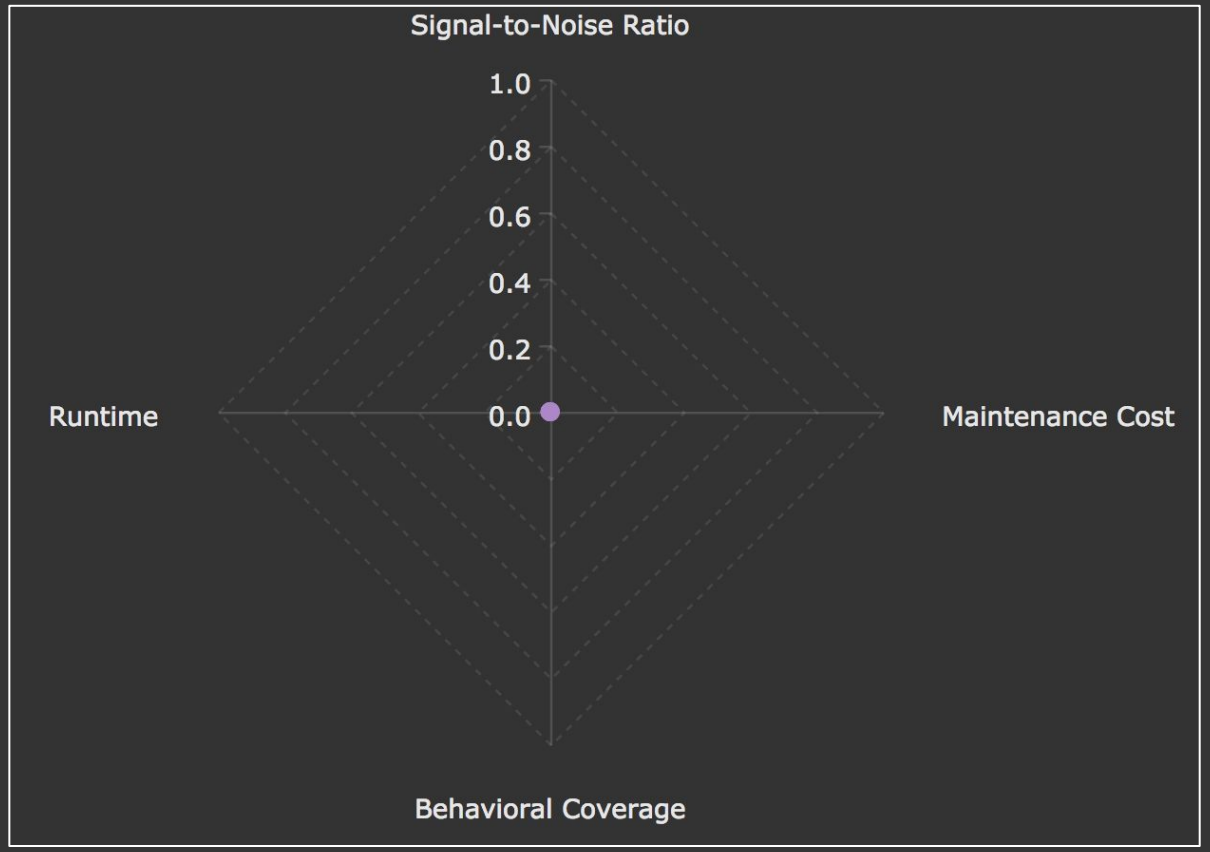
Behavioral Coverage

how many of the system's behaviors are exercised by the test or test suite

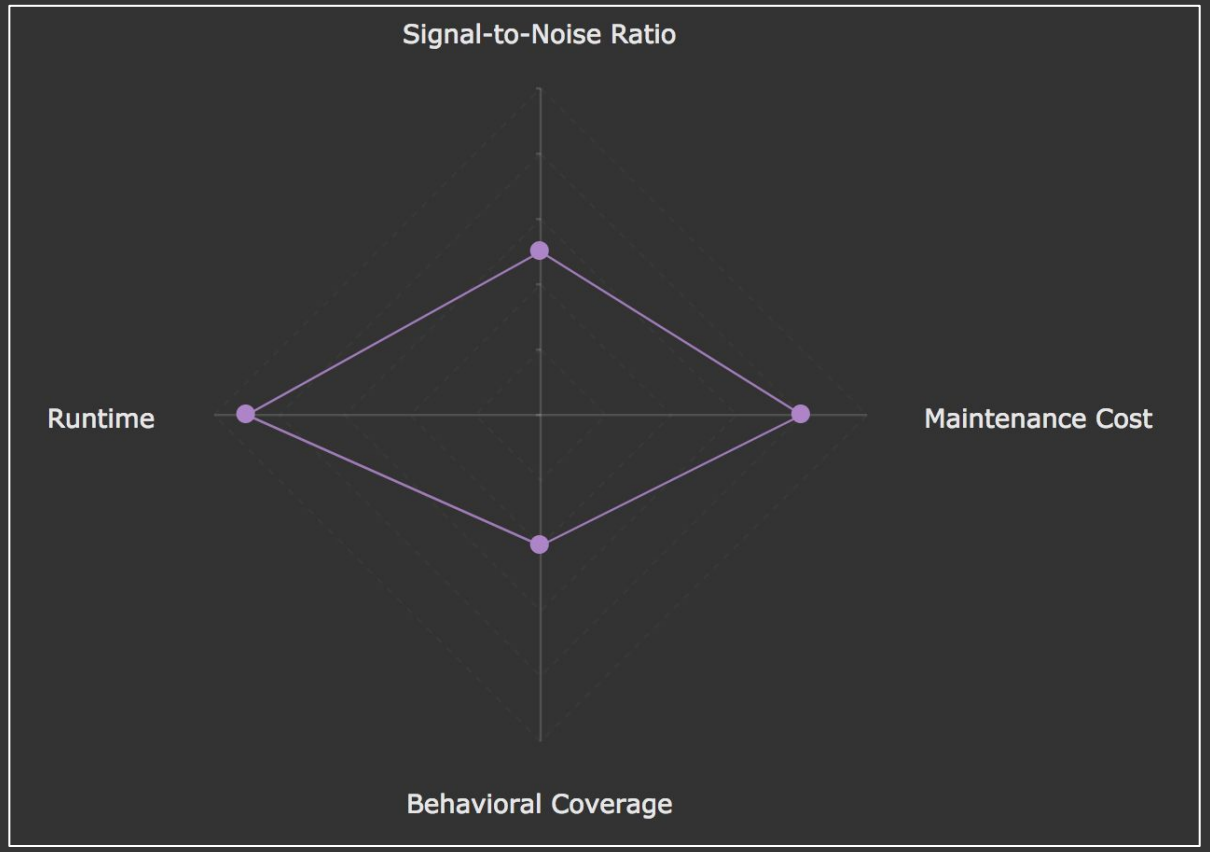
Runtime

how long a test or test suite must run for before it reveals an issue

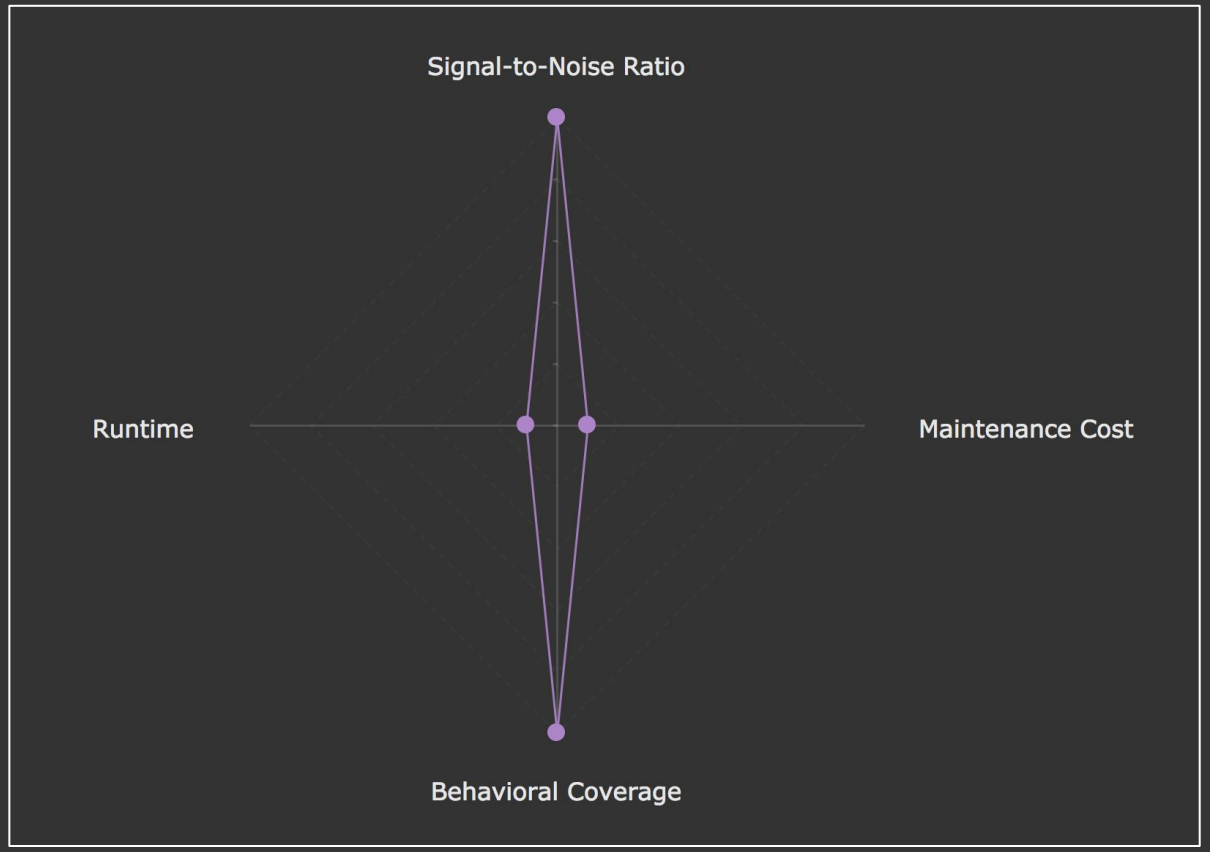
Metrics


Signal-to-Noise Ratio
1.0
0.8
0.6
0.4
0.2
0.0
Runtime
Maintenance Cost
Behavioral Coverage

Metrics


Signal-to-Noise Ratio
Runtime
Maintenance Cost
Behavioral Coverage

Metrics


Signal-to-Noise Ratio
Runtime
Maintenance Cost
Behavioral Coverage

Patterns

Clearly defined and repeatable ways to code test suites

Patterns
Overview

Collaboration Test

A test that proves a subject uses a dependency correctly

"Does this method call another method on a passed-in dependency?"

Functional Test

A test that proves the subject returns a specific output for a specific input

"Does this method return ***y*** when I give it ***x***?"

Contract Test

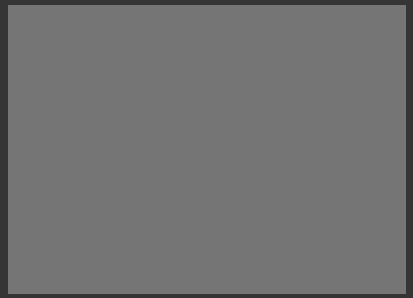
A test of an abstract interface that proves certain behaviors about all the implementers of an interface

"Does every implementation of this method return 15 unique elements?"

Arper - Components

ButtonBankView

Lays out buttons so they can be played. Notifies a delegate that a button is pressed.



Problem

A user must be able to press a button so that the synthesizer can know to make a sound

View layer must not have any other logic

Solution

Create a *UIView* which encapsulates handling the buttons

Delegate out responding to button presses

Use *IndexPaths* to refer to a particular button

```swift
@objc protocol ButtonBankViewDelegate: AnyObject {
    func received(noteEvent: NoteEvent, from indexPath: IndexPath)
}

class ButtonBankView: UIView {
    var buttonBank: [[UIButton]] { get }

    public weak var delegate: ButtonBankViewDelegate?

    init(frame: CGRect, delegate: ButtonBankViewDelegate)
}

```

Collaboration Test

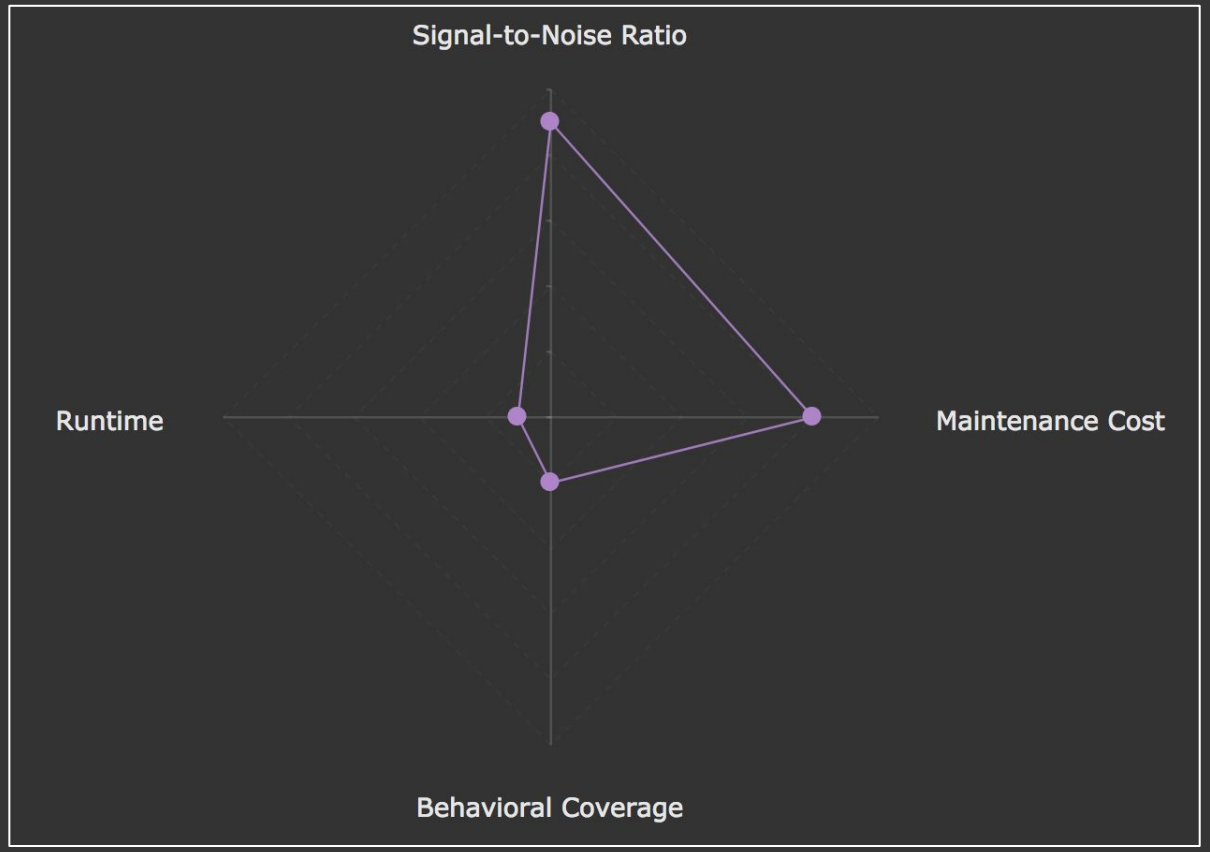
A test which proves a subject uses a dependency correctly

Uses test doubles for validation

Not a test of behavior

Effective for testing against non-deterministic or complex dependencies

Metrics


Signal-to-Noise Ratio
Runtime
Maintenance Cost
Behavioral Coverage

```
1  class ButtonBankViewSpec: QuickSpec {
2      override func spec() {
3
4          var subject: ButtonBankView!
5          var delegateSpy: ButtonBankDelegateSpy!
6
7          describe("ButtonBankView") {
8
9              beforeEach {
10                 delegateSpy = ButtonBankDelegateSpy()
11                 subject = ButtonBankView(frame: .zero, delegate: delegateSpy)
12             }
13
14             it("calls its delegate any time a button receives a touch event") {}
15         }
16     }
17 }
```

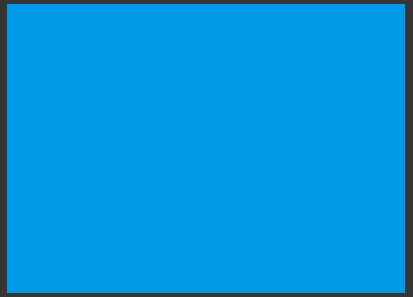
```swift
@objc protocol ButtonBankViewDelegate: class {
    func received(noteEvent: NoteEvent, from indexPath: IndexPath)
}

protocol Spy {
    var methodCalls: [String] { get }
}

class ButtonBankDelegateSpy: Spy, ButtonBankViewDelegate {
    private(set) var methodCalls = [String]()
    private(set) var events = [NoteEvent]()
    private(set) var indexPaths = [IndexPath]()

    func received(noteEvent: NoteEvent, from indexPath: IndexPath) {
        methodCalls.append(#function)
        events.append(noteEvent)
        indexPaths.append(indexPath)
    }
}
```

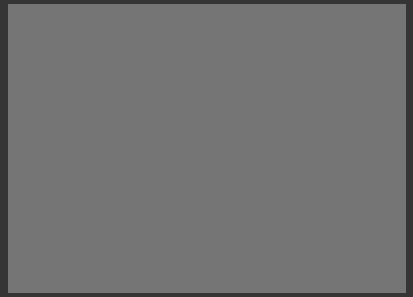
```swift
@objc protocol ButtonBankViewDelegate: class {
    func received(noteEvent: NoteEvent, from indexPath: IndexPath)
}

describe("ButtonBankView") {

    beforeEach {
        delegateSpy = ButtonBankDelegateSpy() // ButtonBankDelegateSpy conforms to ButtonBankDelegate
        subject = ButtonBankView(frame: .zero, delegate: delegateSpy)
    }

    it("notifies its delegate any time a button receives a touch event") {
        let expectedEvents: [NoteEvent] = [.noteOn, .noteOff]
        let expectedIndexPaths: [IndexPath] = [IndexPath(row: 0, section: 0),
                                               IndexPath(row: 0, section: 0)]

        let firstButton = subject.buttonBank.first?.first // buttonBank is a 2-dimensional array
        firstButton?.sendActions(for: .touchDown)
        firstButton?.sendActions(for: .touchUpInside)

        // delegateSpy updates when received(noteEvent:from:) is called
        expect(delegateSpy.methodCalls).to(equal(["received(noteEvent:from:)"]))
        expect(delegateSpy.events).to(equal(expectedEvents))
        expect(delegateSpy.indexPaths).to(equal(expectedIndexPaths))
    }
}
```

Arper - Components

MIDINoteMapping

Determines which button corresponds to a note.





Problem

A user must be able to have a button press correspond to a note

Must support many permutations, as there are many ways to associate buttons with notes

Must use numbers to represent notes

Should use MIDI as inspiration

Solution

Create a *MIDINoteMapping* abstraction which encapsulates mapping buttons to note numbers

Receive an *IndexPath*, return a note number that's valid per MIDI spec

Use *UInt8* to refer to a note number

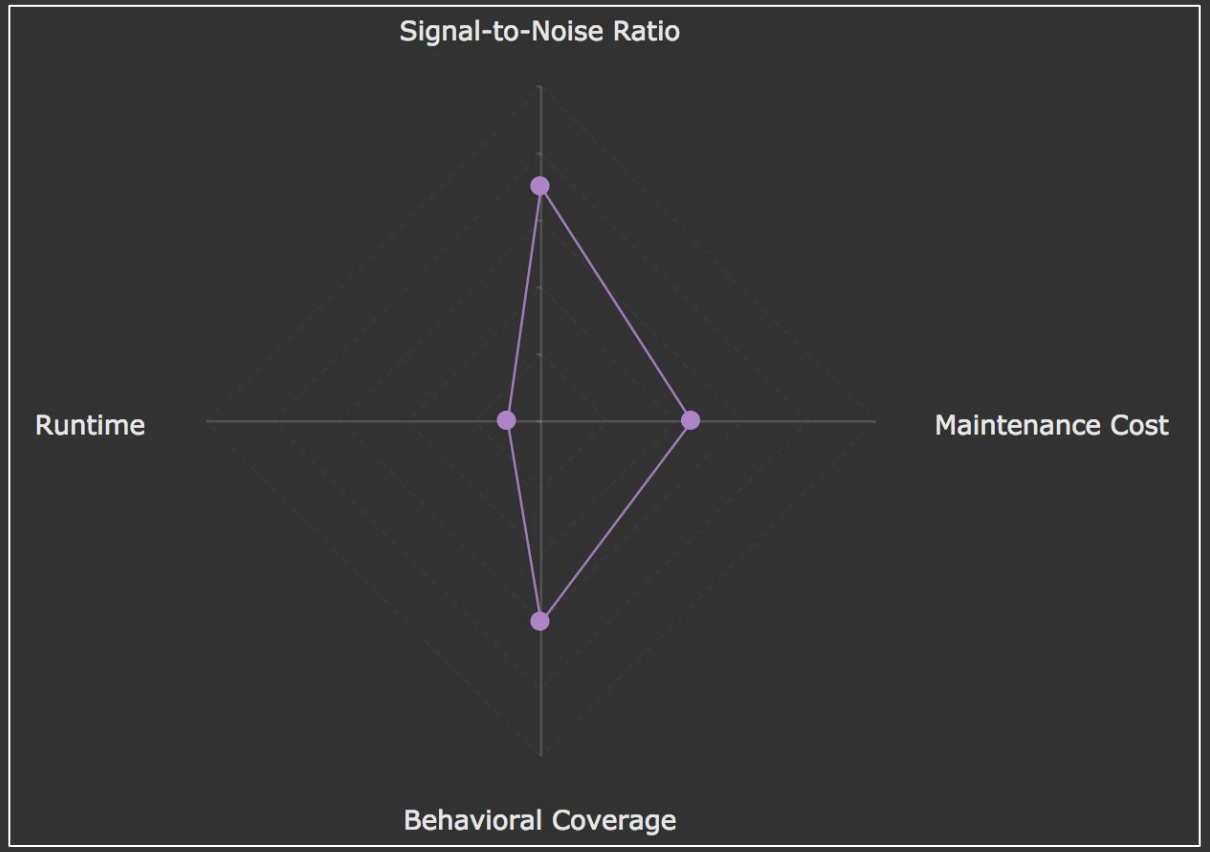
Functional Test

A test which proves the subject's interface works, with no validation of how the subject uses its dependencies.

Proves the subject returns a specific output for specific input

Sometimes called a blackbox test

Metrics

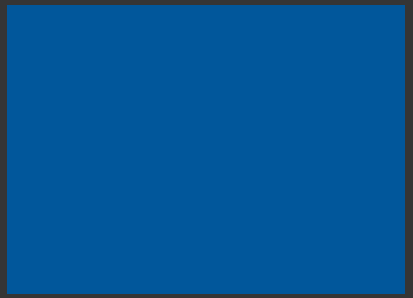

Signal-to-Noise Ratio
Runtime
Maintenance Cost
Behavioral Coverage

```swift
protocol AnyMIDINoteMapping {
    var baseNote: UInt8 { get }
    init(baseNote: UInt8)
    func noteForButton(at indexPath: IndexPath) -> UInt8
}

struct ChromaticNoteMapping: AnyMIDINoteMapping {
    let baseNote: UInt8
}

describe("ChromaticNoteMapping") {
    it("maps IndexPaths into note numbers using the chromatic scale"){

        let chromaticScale: [UInt8] = [60, 61, 62, 63, 64,
                                       65, 66, 67, 68, 69,
                                       70, 71, 72, 73, 74]

        let chromaticNoteMapping = ChromaticNoteMapping(baseNote: 60)

        expect(notes(using: chromaticNoteMapping)).to(equal(chromaticScale))
    }
}
```

```swift
func notes(using mapping: AnyMIDINoteMapping) -> [UInt8] {

    let notes: [UInt8] = (0...2).flatMap { rowNumber in
        return notesForRow(number: rowNumber, using: mapping)
    }

    return notes
}

func notesForRow(number: Int, using mapping: AnyMIDINoteMapping) -> [UInt8] {

    return (0...4).map { columnNumber in

        let indexPath = IndexPath(row: number, section: columnNumber)
        return mapping.noteForButton(at: indexPath)

    }
}
```

```
1  struct ChromaticNoteMapping {
2      var baseNote: UInt8 { get }
3      init(baseNote: UInt8)
4      func noteForButton(at indexPath: IndexPath) -> UInt8
5  }
6
7  describe("ChromaticNoteMapping") {
8      it("maps IndexPaths into note numbers using the chromatic scale"){
9
10         let chromaticScale: [UInt8] = [60, 61, 62, 63, 64,
11                                        65, 66, 67, 68, 69,
12                                        70, 71, 72, 73, 74]
13
14         let chromaticNoteMapping = ChromaticNoteMapping(baseNote: 60)
15
16         expect(notes(using: chromaticNoteMapping)).to(equal(chromaticScale))
17
18     }
19 }
```

Arper - Components



AudioEngineManager

Handles note routing into the audio engine. Contains logic for interpreting various controls.

AudioEngine

Handles converting notes into sound.

Example of Balancing Metrics

SNR, Maintenance Cost, & Behavioral Coverage

```
describe("AKAudioEngineManager") {
    beforeEach {
        audioEngineSpy = AudioEngineSpy()
        subject = AKAudioEngineManager(audioEngine: audioEngineSpy,
                                       noteMappings: [ChromaticNoteMapping(baseNote: 0)])
    }

    it("routes notes to the audio engine for rendering") {
        let firstIndexPath = IndexPath(row: 0, section: 0)
        let secondIndexPath = IndexPath(row: 0, section: 1)

        subject.received(noteEvent: .noteOn, from: firstIndexPath)
        subject.received(noteEvent: .noteOff, from: firstIndexPath)
        subject.received(noteEvent: .noteOn, from: secondIndexPath)
        subject.received(noteEvent: .noteOff, from: secondIndexPath)

        expect(audioEngineSpy.methodCalls.first).to(equal("render(notesNumbered:)"))
        expect(audioEngineSpy.methodCalls.last).to(equal("stopRendering(of:)"))

        expect(audioEngineSpy.renderedNoteNumbers).to(equal([0, 1]))
        expect(audioEngineSpy.stoppedNoteNumbers).to(equal([0, 1]))
    }

    // 12 more tests are omitted
}
```

Balancing Metrics

Balancing Metrics

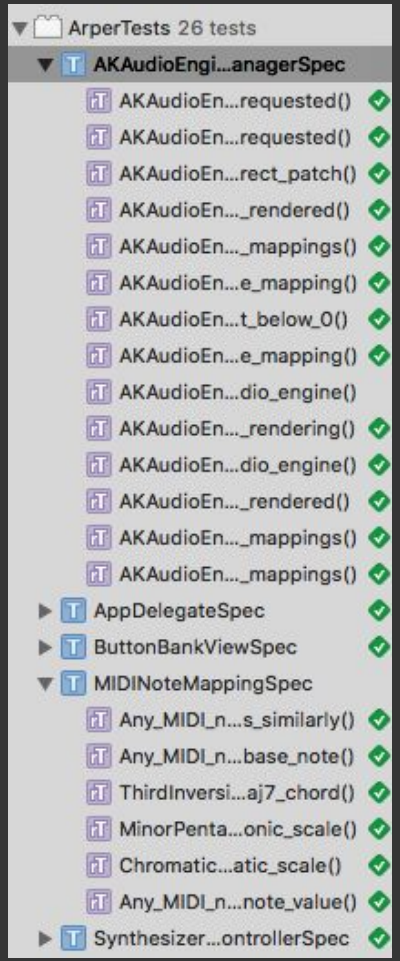
```
1 describe("AKAudioEngineManager") {
2     beforeEach {
3         audioEngineSpy = AudioEngineSpy()
4         subject = AKAudioEngineManager(audioEngine: audioEngineSpy,
5                                        noteMappings: [ChromaticNoteMapping(baseNote: 0)])
6     }
7     // 13 more tests are omitted
8 }
```

Test Saboteuring

The process of intentionally introducing issues into a code base to assess its test suite's efficacy

Balancing Metrics

```
1 describe("AKAudioEngineManager") {
2     beforeEach {
3         audioEngineSpy = AudioEngineSpy()
4         subject = AKAudioEngineManager(audioEngine: audioEngineSpy,
5                                        noteMappings: [ChromaticNoteMapping(baseNote: 0)])
6     }
7     // 13 more tests are omitted
8 }
```



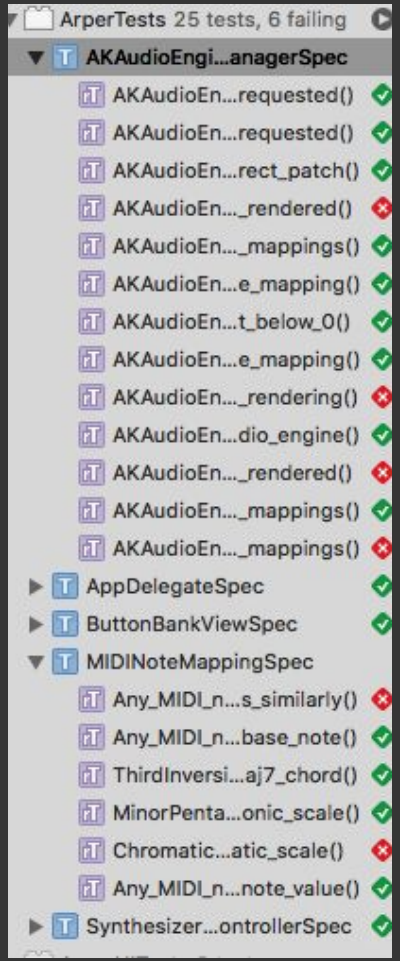
Balancing Metrics

Before

```
1 struct ChromaticNoteMapping: MIDINoteMapping {
2     let intervals: [UInt8] = [1]
3     //
4 }
```

After

```
1 struct ChromaticNoteMapping: MIDINoteMapping {
2     let intervals: [UInt8] = [0]
3     //
4 }
```



Possible Solution

Isolate *AudioEngineManager* from changes in *ChromaticNoteMapping* with a test double

Contract Test

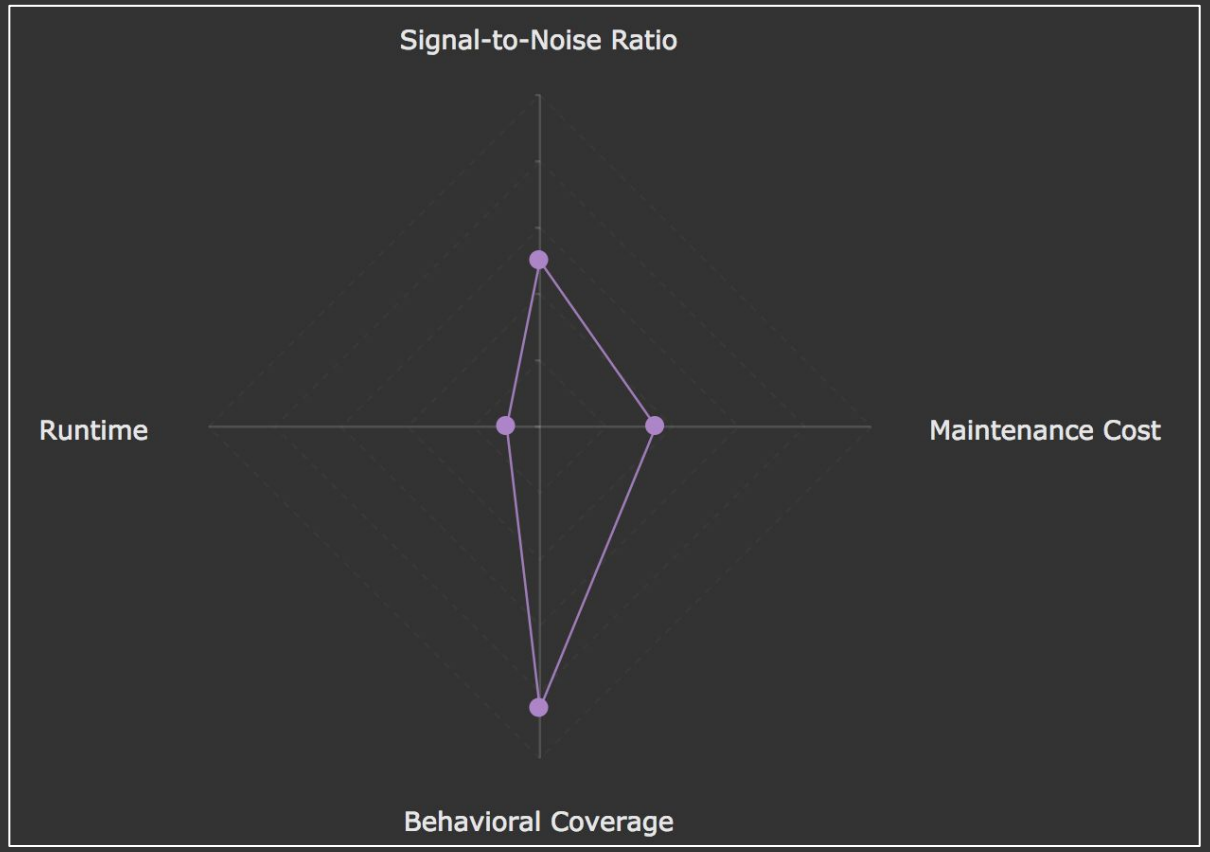
A test of an abstract interface that proves certain behaviors about all the implementers of an interface.

In Swift, used for types that conform to protocols or subclass.

Used when a type system doesn't completely cover the behavior of the subject's interface.

Test multiple types at once

Metrics


Signal-to-Noise Ratio
Runtime
Maintenance Cost
Behavioral Coverage

Balancing Metrics

```
describe("Any MIDI note mapping") {
    var mappings: [AnyMIDINoteMapping]!

    beforeEach {
        mappings = allMIDINoteMappings(usingBaseNote: 60)
    }

    it("maps 15 notes with no repeating notes") {}

    it("defaults notes above the allowable range to the maximum MIDI note value") {}

    it("maps notes starting from a provided base note") {}
}
```

Balancing Metrics

```
1 func allMIDINoteMappings(usingBaseNote baseNote: UInt8) -> [AnyMIDINoteMapping] {
2     return [
3       MinorPentatonicNoteMapping(baseNote: baseNote),
4       ChromaticNoteMapping(baseNote: baseNote),
5       ThirdInversionMajor7NoteMapping(baseNote: baseNote),
6       MIDINoteMappingFake(baseNote: baseNote)
7     ]
8 }
```

Balancing Metrics

```
describe("Any MIDI note mapping") {
    var mappings: [AnyMIDINoteMapping]!

    beforeEach {
        mappings = allMIDINoteMappings(usingBaseNote: 60)
    }

    it("maps 15 notes with no repeating notes") {}

    it("defaults notes above the allowable range to the maximum MIDI note value") {}

    it("maps notes starting from a provided base note") {}
}
```

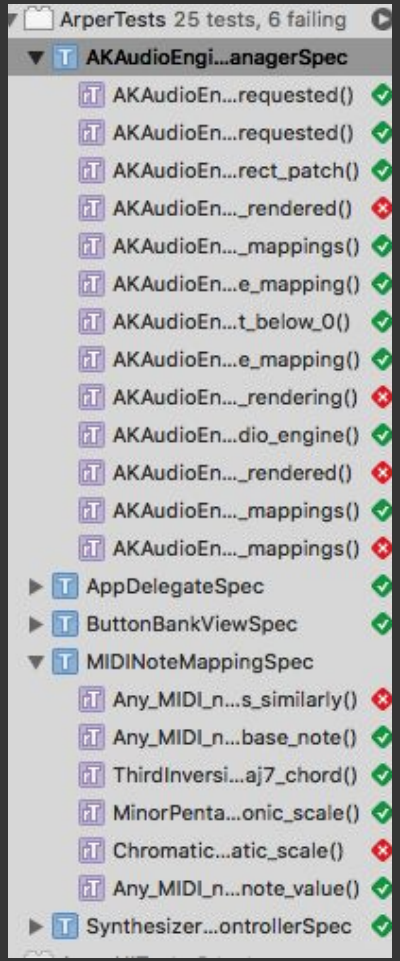
Balancing Metrics

```
describe("Any MIDI note mapping") {
    var mappings: [AnyMIDINoteMapping]!

    beforeEach {
        mappings = allMIDINoteMappings(usingBaseNote: 60)
    }

    it("maps 15 notes with no repeating notes") {
        for anyMapping in mappings {
            expect(notes(using: anyMapping)).to(haveCount(15))
            expect(notes(using: anyMapping)).to(haveUniqueElements(15))

            let theFirstNote = anyMapping.noteForButton(at: IndexPath(row: 0, section: 0))
            let theFirstNoteMappedAgain = anyMapping.noteForButton(at: IndexPath(row: 0, section: 0))

            expect(theFirstNote).to(equal(theFirstNoteMappedAgain))
        }
    }
}
```

```swift
class MIDINoteMappingFake: AnyMIDINoteMapping {

    let baseNote: UInt8

    required init(baseNote: UInt8) {
        self.baseNote = baseNote
    }

    func noteForButton(at indexPath: IndexPath) -> UInt8 {
        return noteMapping[indexPath.row][indexPath.section]
    }
}

private extension MIDINoteMappingFake {

    var noteMapping: [[UInt8]] {
        let mapping: [[UInt8]] = (0...2).map { rowNumber in
            return notesForRow(number: rowNumber)
        }

        return mapping
    }

    func notesForRow(number: UInt8) -> [UInt8] {
        return (0...4).map { columnNumber in

            if number == 0 && columnNumber == 0 {
                return min(baseNote, 127)
            }

            let rowOffset = UInt8(number) * 5
            let columnOffset = UInt8(columnNumber)
            let note = baseNote + rowOffset + columnOffset

            return min(note, 127)
        }
    }
}
```

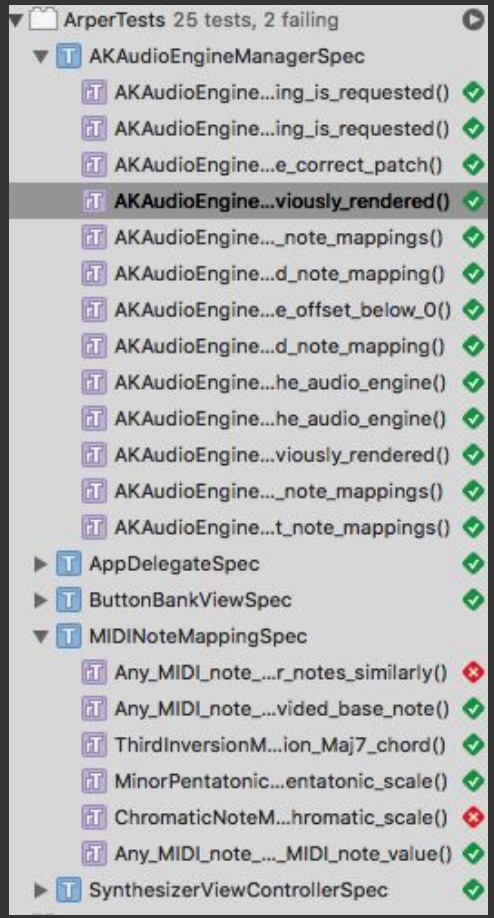
Balancing Metrics

```
1 describe("AKAudioEngineManager") {
2     beforeEach {
3         audioEngineSpy = AudioEngineSpy()
4         subject = AKAudioEngineManager(audioEngine: audioEngineSpy,
5                                        noteMappings: [ChromaticNoteMapping(baseNote: 0)])
6     }
7     // 13 more tests are omitted
8 }
```



Balancing Metrics

```
describe("AKAudioEngineManager") {
    beforeEach {
        audioEngineSpy = AudioEngineSpy()
        subject = AKAudioEngineManager(audioEngine: audioEngineSpy,
                                       noteMappings: [MIDINoteMappingFake(baseNote: 0)])
    }
}
```



What was improved?

Signal-to-Noise Ratio

Balancing Metrics

What was worsened?

Maintenance Cost

Behavioral Coverage

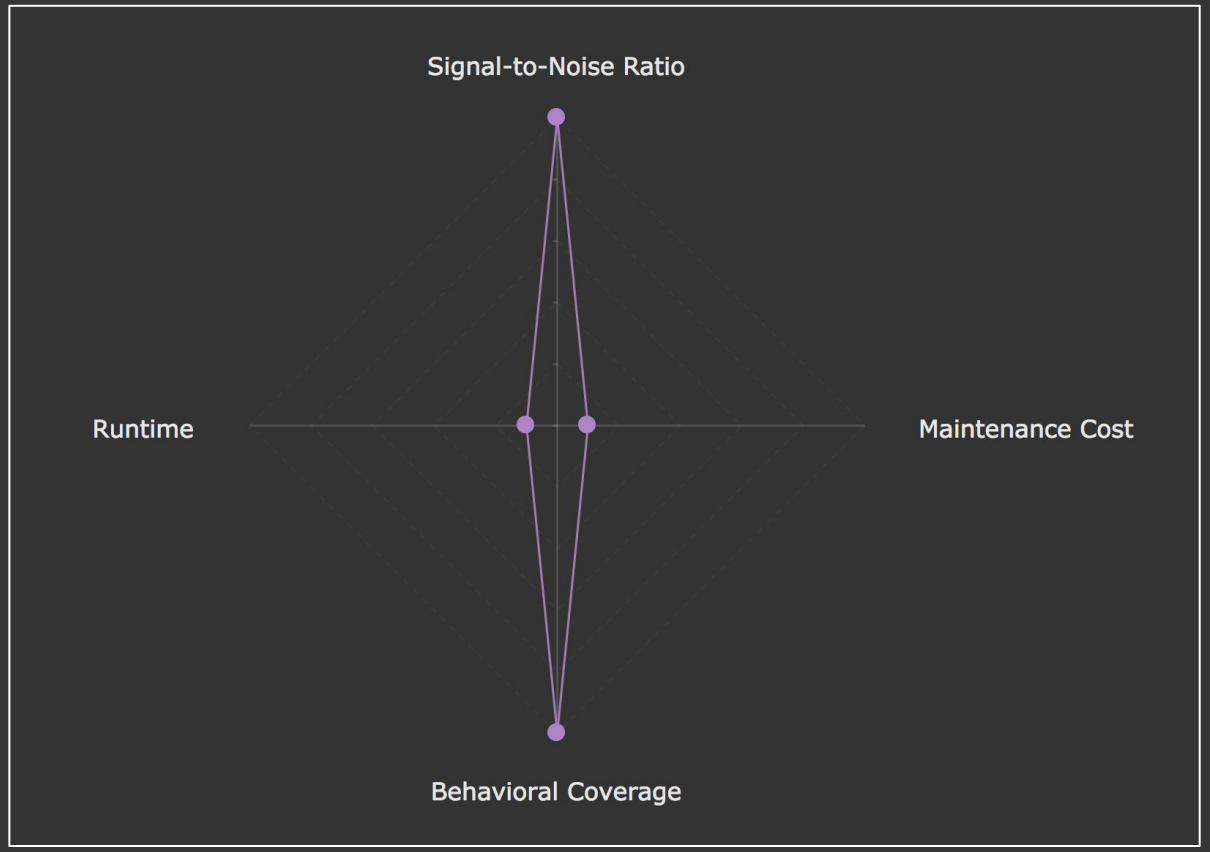
Balancing Metrics

Was it the right decision to make?

Balancing Metrics

Depends on what you need your test suite to do for you

Metrics


Signal-to-Noise Ratio
Runtime
Maintenance Cost
Behavioral Coverage

Review

Review
Metrics

Signal-to-Noise Ratio

how clearly a test failure indicates a specific fault or issue in your code base

Maintenance Cost

how much effort must go into keeping a test or test suite effective

Behavioral Coverage

how many of the system's behaviors are exercised by the test or test suite

Runtime

how long a test or test suite must run for before it reveals an issue

Review

Testing Patterns

Collaboration Test

A test that proves a subject uses a dependency correctly

"Does this method call another method on a passed-in dependency?"

Functional Test

A test that proves the subject returns a specific output for a specific input

"Does this method return ***y*** when I give it ***x***?"

Contract Test

A test of an abstract interface that proves certain behaviors about all the implementers of an interface

"Does every implementation of this method return exactly 15 elements?"

Review

Testing Methodologies

Test Doubling

Replacing a subject's dependency with an implementation only meant for testing.

Use sparingly.

Test Saboteuring

Introducing issues into your code to assess how effective your test suite is.

Practice often.

Behavior Driven Development

Designing components and tests with a focus on what something is supposed to do, without regard for implementation.

Practice always.

Thank you!

Sean Olszewski

github.com/SeanROlszewski

@__chefski__