Emerging Best Practices in Swift

Ash Furrow

Ash Furrow - Solving Problems the Swift Way

I was Afraid

Everything turned out

Fine

Best Practices in Swift

What do they look like?

Also, how do we find new ones?

Agenda

- We’ve been here before
- Learning is forever, deal with it
- Never throw ideas away
- How to force yourself to think
- Always be abstracting

Let’s go!

This looks strangely familiar…

“Those who don’t study history are doomed to repeat it.”

—Lots of people, for hundreds of years

Wow, that’s depressing.

"Those who don't understand the past
can't make informed decisions about the present."

–Me, today

iOS 5 or earlier?

Before Object Literals

```
NSArray *array = [NSArray arrayWithObjects:
    @"This",
    @"is",
    @"so",
    @"tedious", nil];

NSDictionary *dictionary = [NSDictionary
    dictionaryWithObjectsAndKeys:
    @"Who would do this?", @"Not me", nil];

NSNumber *number = [NSNumber numberWithInt:401];
```

Before Object Literals *and* ARC

```
NSArray *array = [[NSArray arrayWithObjects:
    @"This",
    @"is",
    @"so",
    @"tedious", nil] alloc];

NSDictionary *dictionary = [[NSDictionary
    dictionaryWithObjectsAndKeys:
    @"Who would do this?", @"Not me", nil] alloc];

NSNumber *number = [[NSNumber numberWithInt:401] alloc];
```

After Object Literals

```
NSArray *array =
    @[ @"This", @"is", @"much", @"better" ];

NSDictionary *dictionary =
    @{ @"Who likes this?": @"Me!" };

NSNumber *number = @(401);
```

Object Literals

- Clearly *way* better
- Adopted by everyone almost *immediately*
- Became a “best practice”

Λ{

Blocks & GCD

- Introduced in iOS 4
- Adopted slowly, but surely
- Required new ways of thinking
- Did using blocks became a “best practice”?
    - Sort of…

Blocks

Enable

other best practices

Contextual Code Execution

Functional Reactive Programming

^{

Futures

Inline Network Operations

Generic Datasource Objects

Callbacks

Promises

Deferred Customization

Collections Operations

Embrace

Change

Swift 2

Swift 2

- Lots of new syntax
- New syntax lets us do new things
- However! Syntax is only a *tool*
    - Like blocks, Swift 2 syntax is most useful when it enables new *ideas*

Swift 2

- guard
- defer
- throws
- etc…

Should I use guard?

What can I *do* with guard?

Examples

Pyramid of Doom

```
if let thing = optionalThing {
    if thing.shouldDoThing {
        if let otherThing = thing.otherThing {
            doStuffWithThing(otherThing)
        }
    }
}
```

Clause Applause

```
if let thing = optionalThing,
   let otherThing = thing.otherThing
   where thing.shoudDoThing {
    doStuffWithThing(otherThing)
}
```

Avoid Mutability

```swift
func strings(
    parameter: [String],
    startingWith prefix: String) -> [String] {

    var mutableArray = [String]()
    for string in parameter {
        if string.hasPrefix(prefix) {
            mutableArray.append(string)
        }
    }

    return mutableArray
}
```

ಠ_ಠ

Avoid Mutability

```swift
func strings(
    parameter: [String],
    startingWith prefix: String) -> [String] {

    return parameter.filter { $0.hasPrefix(prefix) }
}
```

Currying

- One of those weird words you avoid because people who say it are sometimes jerks
- It's actually a pretty straightforward concept
- Currying is a function that returns another function
- Useful for sharing code that's *mostly* the same

Before Currying

```swift
func containsAtSign(string: String) -> Bool {
    return string.characters.contains("@")
}

...

input.filter(containsAtSign)
```

Currying

```
func contains(substring: String) -> (String -> Bool) {
    return { string -> Bool in
        return string.characters.contains(substring)
    }
}

...

input.filter(contains("@"))
```

Currying

```swift
func contains(substring: String)(string: String) -> Bool {
    return string.characters.contains(substring)
}

...

input.filter(contains("@"))
```

Extract Associated Values

- Use Swift enums
- Attach associated values
- Extract using a case

Extract Associated Values

```
enum Result {
    case Success
    case Failure(reason: String)
}

switch doThing() {
case .Success:
    print("🎉")
case .Failure(let reason):
    print("Oops: \(reason)")
}
```

Extract Associated Values

```
enum Result {
    case Success
    case Failure(reason: String)
}

if case .Failure(let reason) = doThing() {
    print("😢 \(reason)")
}
```

That's all just

Syntax

What matters are

Ideas

Protocol-Oriented Programming

… just go watch the WWDC video.

Let’s ask Others

Syntax vs Idea

- How to tell if something is universally a good idea, or just enables other ideas?
  - You can’t
  - It’s a false dichotomy
  - I lied to you
  - I’m so sorry

You've just got to

Try stuff

Never throw away

Ideas

Never Throw Away Ideas

- Swift was released
- We treated Swift like object literals instead of like blocks
  - Some of us thought Swift was universally better
  - My fault, oops 😅

Older ideas have

Merit

iOS developers throw things away

A lot

Why?

Beginner learns thing

Is bad at thing

Blames thing

Thing must be bad

Beginner gets more experience

New thing comes out

Learning new thing is easier than old thing

New thing must be good

New ideas are

Appealing

iOS is constantly changing

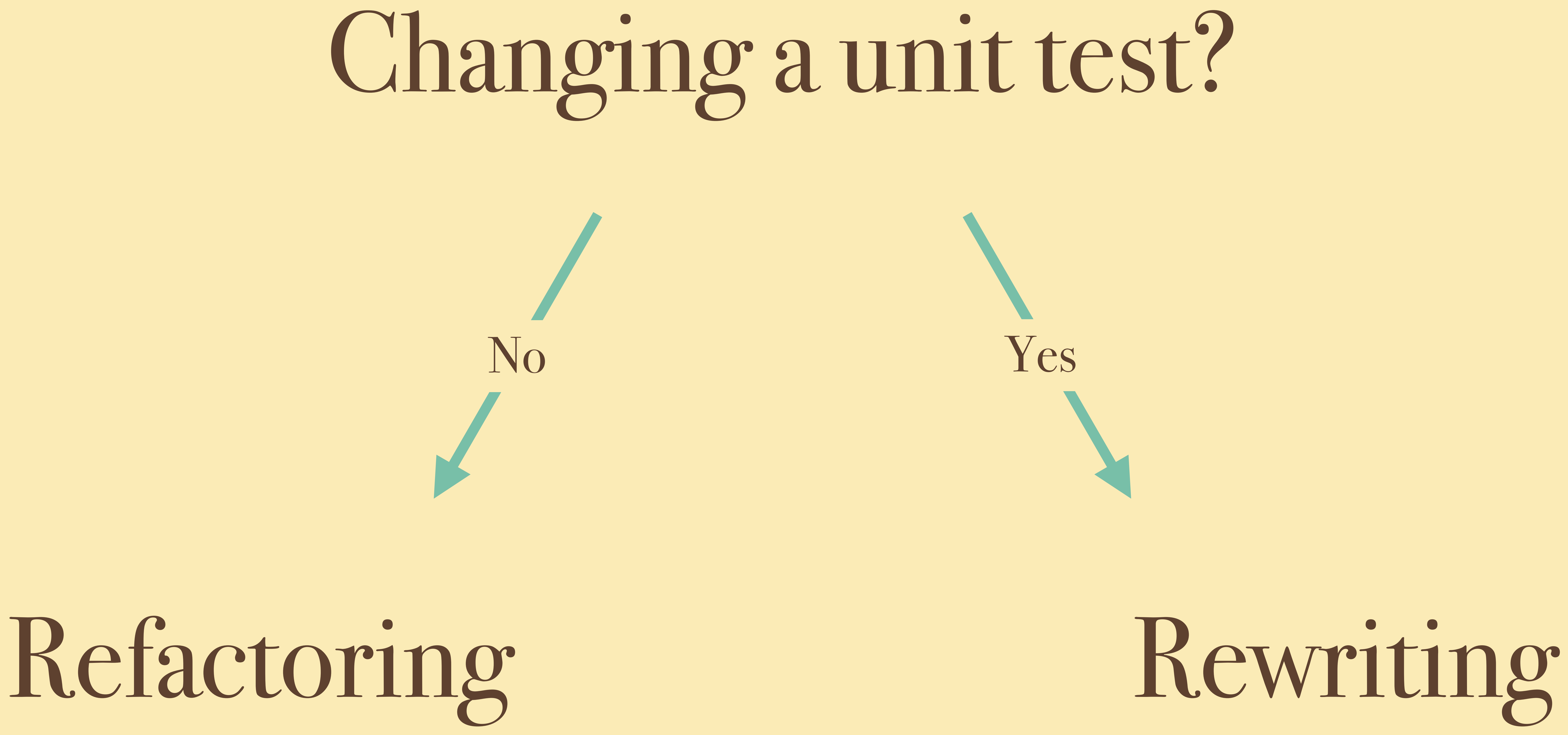
Always a fresh supply of old APIs
for us to blame

Let's talk about

Refactoring

What is Not Refactor?

- Refactoring does not add new functionality
- Refactoring does not change a type’s interface
- Refactoring does not change a type’s behaviour


Changing a unit test?
No
Yes
Refactoring
Rewriting

Rewrites are Bad

Joel on Software

Things You Should Never Do, Part I

by Joel Spolsky

Favour incremental change

Code isn’t necessarily valuable

But throwing it away is dangerous

Things to never throw away:

Code & Ideas

Unit tests will help

Change

Unit Testing & Thinking

- So, uhh, unit testing
- Controversial in iOS
    - Not so much everywhere else
- Why?
    - We’ll get to that

Benefits of Testing

- (Let’s presume that unit testing is a good idea)
- I really don’t care that much about the tests
- I care more about how writing tests makes me think about what I’m writing

Benefits of Testing

- Limited object scope is good
  - High cohesion, low coupling
- How to limit scope?
  - Controlling public interface and dependencies

Dependency injection?

Dependency Injection

- €5 word for a ¢5 idea
- Your things shouldn’t create the things they need

Example

Without Dependency Injection

```swift
class ViewController: UIViewController {
    let networkController = NetworkController()

    func viewDidLoad() {
        super.viewDidLoad()
        networkController.fetchStuff {
            self.showStuff()
        }
    }
}
```

With Dependency Injection

```
class ViewController: UIViewController {
    var networkController: NetworkController?

    func viewDidLoad() {
        super.viewDidLoad()
        networkController?.fetchStuff {
            self.showStuff()
        }
    }
}
```

Dependency Injection

- Rely on someone else to configure your instance
  - Could be another part of your app (eg: prepareForSegue)
  - Could be a unit test
- Protocols work really well for this

Dependency Injection

```
protocol NetworkController {
    func fetchStuff(completion: () -> ())
}

...

class APINetworkController: NetworkController {
    func fetchStuff(completion: () -> ()) {
        // TODO: fetch stuff and call completion()
    }
}
```

Dependency Injection

```swift
protocol NetworkController {
    func fetchStuff(completion: () -> ())
}

...

class TestNetworkController: NetworkController {
    func fetchStuff(completion: () -> ()) {
        // TODO: stub fetched stuff
        completion()
    }
}
```

Dependency Injection

- Use of protocols limits coupling between types
    - Adding a method to a protocol becomes a decision you have to make
- Dependency injection can als be used for shared state, like singletons

Without Dependency Injection

```
func loadAppSetup() {
    let defaults =
        NSUserDefaults.standardUserDefaults()

    if defaults.boolForKey("launchBefore") == false {
        runFirstLaunch()
    }
}
```

How would you even test that?

With Dependency Injection

```
func loadAppSetup(defaults: NSUserDefaults) {
    if defaults.boolForKey("launchBefore") == false {
        runFirstLaunch()
    }
}
```

Don’t be an ideologue

Cheat with Dependency Injection

```
func loadAppSetup(
    defaults: NSUserDefaults = .standardUserDefaults()){

    if defaults.boolForKey("launchBefore") == false {
        runFirstLaunch()
    }
}
```

Cheat with Dependency Injection

```
loadAppSetup() // In your app

loadAppSetup(stubbedUserDefaults) // In your tests
```

Cheat with Dependency Injection

```swift
class ViewController: UIViewController {
    lazy var networkController: NetworkController =
        APINetworkController()

    func viewDidLoad() {
        super.viewDidLoad()
        networkController.fetchStuff {
            self.showStuff()
        }
    }
}
```

“TDD if necessary, but not necessarily TDD.”

—Mackenzie King (Canada’s Winston Churchill)

Unit Testing

- Don’t test private functions
  - Also, start marking functions as private
- Remember, we want to avoid rewriting
- Don’t test the implementation
- Don’t use “partial mocks”
  - See @searls post on partial mocks

Unit Testing

- So why don’t iOS developers do unit testing?
  - It’s unfamiliar and no one forces us to do it

Testing code makes me a

Better Developer

Abstract
Everything

Two or more lines of repeated code?

Find a better way

(╯°□°)╯︵ ┻━┻

Look for Abstractions

- You're already learning new syntax
- Look for new abstractions along the way
- Not all ideas will work out
- But you should still do it
- Experiment!

No such thing

as a

Failed experiment

Always opportunities to

Learn

Wrap Up

- We have a history of being awesome, let’s keep it up
- Learning isn’t just for when Xcode is in beta
- Ideas are more valuable than code, but throwing away either is dangerous
- Effective unit tests make it easy to change code
- Operate at the highest level of abstraction you can at any given time

Make

Better Mistakes

Tomorrow