Adapting to change: designing for modularity and maintainability in Swift

Jesse Squires
jessesquires.com • @jesse_squires

I work at

Construction and software have a lot in common

Connecting model and UI layers

When you forget to implement accessibility features

Client-server communication

When you can only do half of the refactoring

When your new feature has user privacy or usability issues

DESIGNING

BUILDING

PROTOTYPING

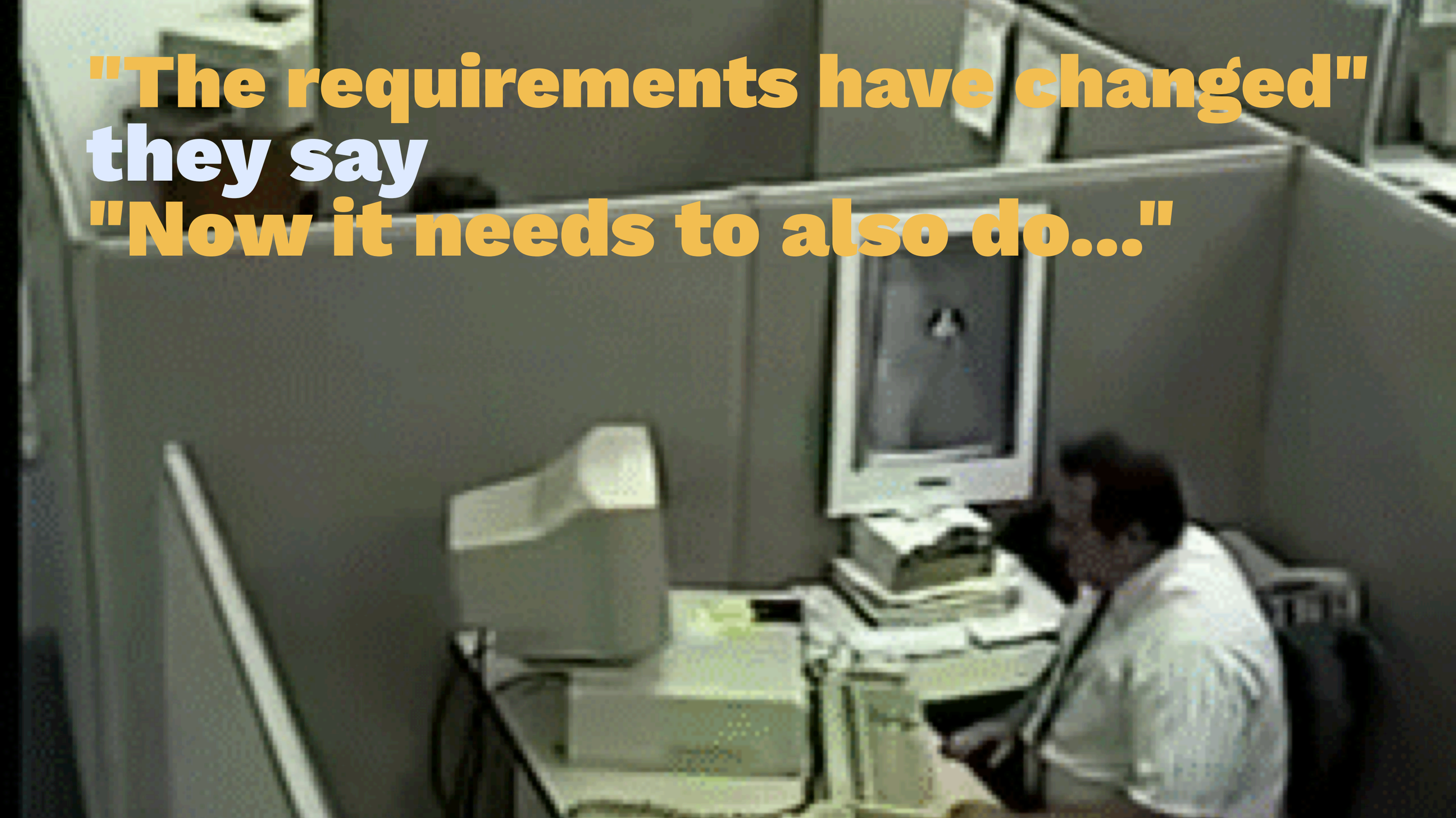
VERIFYING


So, you're writing some code


And everything is going great


Then your product manager is like,
"Hey, you got a sec? Can we talk about..."
GIFAK.NET

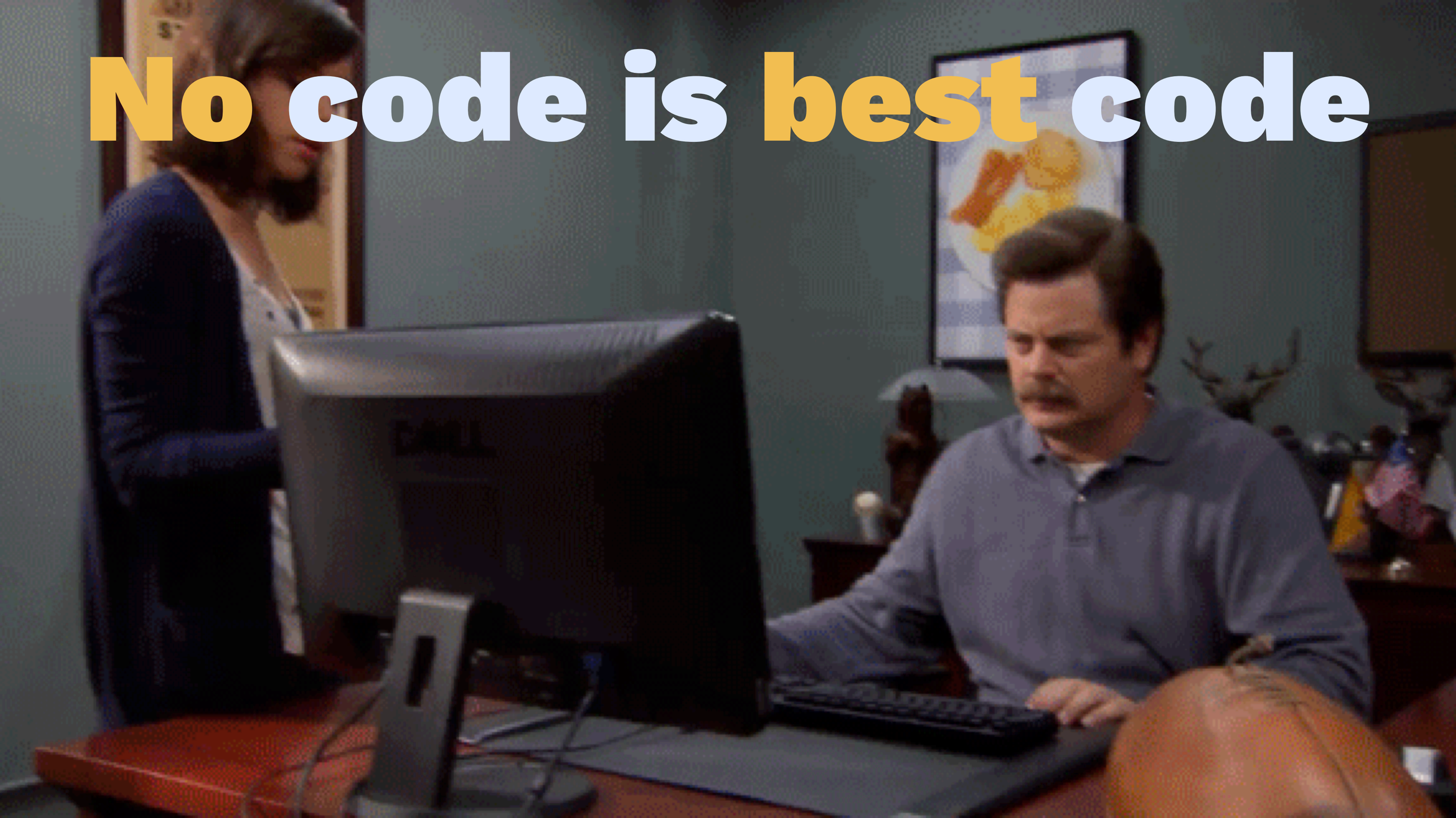

You accept the meeting invite.
You know what's coming...
CM 690


"The requirements have changed"
they say
"Now it needs to also do..."

But, you've coded yourself into a corner

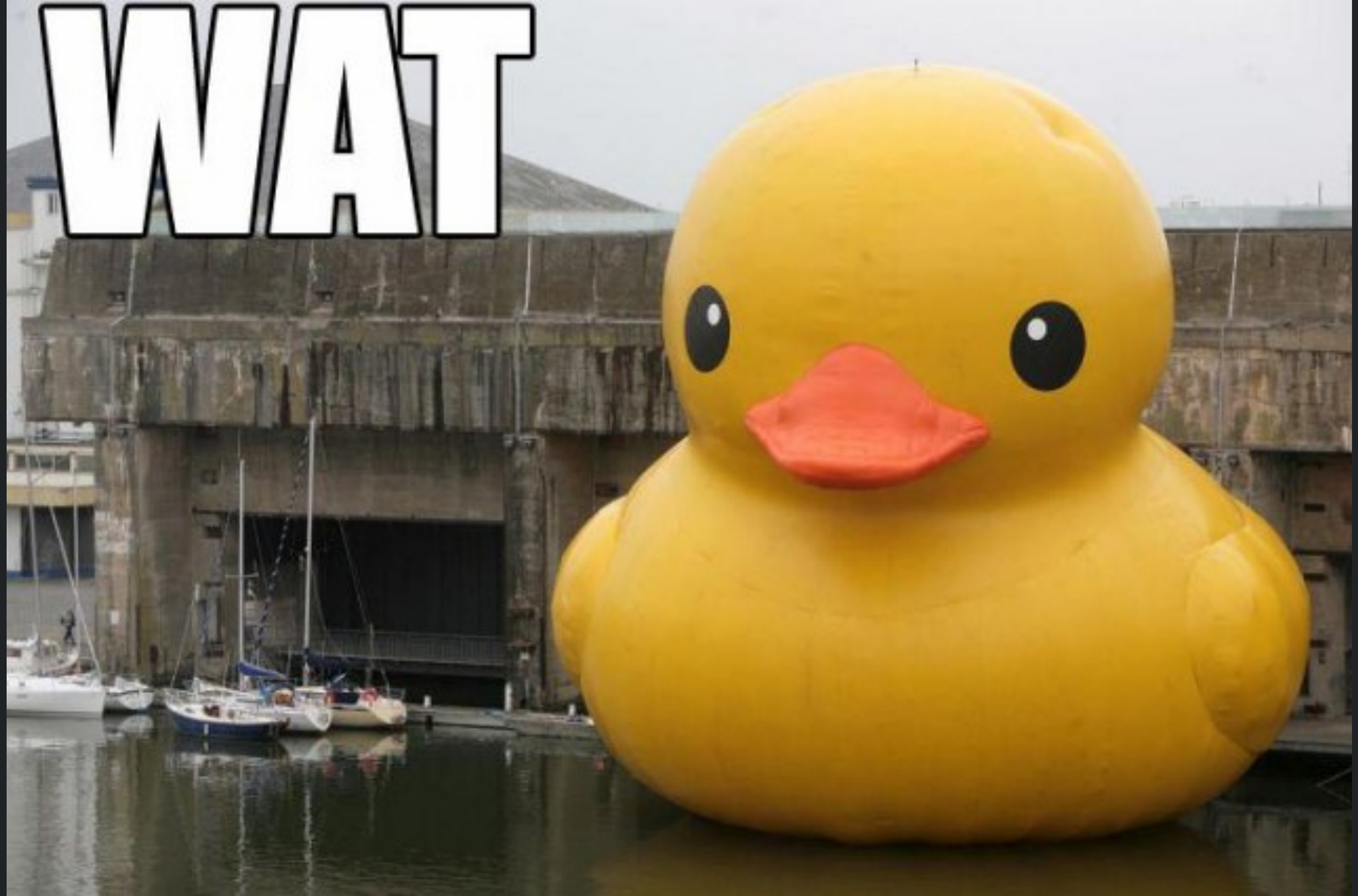

And you can't get out


How can we
prevent this?


No code is best code

Solution: better* code

* Slightly less terrible


WAT

Slava Pestov
@slava_pestov

A large portion of every code base is just programmers showing off, creating unnecessary complexity to prove how smart they are

S O L I D
design principles

(Good ideas here for object-oriented **and** functional programming!)

Single responsibility

Open/closed

Liskov substitution

Interface segregation

Dependency inversion

Any type you create should only have one reason to change

Single responsibility ❌

```
class ImageDownloader {

    let cache: [URL: UIImage]
    let fileManager: FileManager
    let session: URLSession

    func getOrDownloadImage(url: URL,
                            completion: @escaping (UIImage?) -> Void)
}
```

Single responsibility ✅

```
class Downloader<T> {
    func downloadItem(url: URL, completion: @escaping (Result<T>) -> Void)
}

class Cache<T> {
    func store(item: T, key: String, completion: @escaping (Bool) -> Void)

    func retrieveItem(key: String, completion: @escaping (Result<T>) -> Void)
}

class DataProvider<T> {
    let downloader: Downloader<T>
    let cache: Cache<T>

    func getItem(url: URL, completion: @escaping (Result<T>) -> Void)
}
```

Open for extension closed for modification

How can we change behavior without modifying a type?

- Subclass 😭
- Inject dependency
  - Inject protocol
  - Pass a function

Open / closed

Examples

`UITableViewDelegate`

```
func filter(_ isIncluded: (Element) throws -> Bool) rethrows -> [Element]
```

Liskov substitution

Types should be **replaceable** with instances of their **subtypes** without altering **correctness**

```
protocol ParentProtocol { }

protocol ChildProtocol: ParentProtocol { }

func foo(bar: ParentProtocol) { }
```

Interface segregation

Use many specific interfaces, rather than one general purpose

```
protocol UITableViewDataSource {
    func tableView(_, cellForRowAt: ) -> UITableViewCell
    func numberOfSections(in: ) -> Int
    func tableView(_, numberOfRowsInSection: ) -> Int
}

protocol UITableViewDataSourceEditing {
    func tableView(_, commit: , forRowAt: )
    func tableView(_, canEditRowAt: ) -> Bool
}
```

Dependency inversion

Decouple via protocols and injection

```
class MyViewController: UIViewController {

    let userManager: CurrentUserManager
    let defaults: UserDefaults
    let urlSession: URLSession

    init(userManager: CurrentUserManager = .shared,
         defaults: UserDefaults = .standard,
         urlSession: URLSession = .shared)
}
```

Intertwined components create cognitive and maintenance burdens

Let's talk about CLARITY

Example: drawing line graphs 

```
let p1 = Point(x1, y1)
let p2 = Point(x2, y2)
let slope = p1.slopeTo(p2)
```

Need to check if the slope is:

— undefined (vertical line)

— zero (horizontal line)

— positive

— negative

We could write this

```
if slope == 0 {
    // horizontal line

} else if slope.isNaN {
    // vertical line, undefined

} else if slope > 0 {
    // positive slope

} else if slope < 0 {
    // negative slope
}
```

Or, we could add extensions and **remove** the comments.

```
if slope.isHorizontal {

} else if slope.isUndefined {

} else if slope.isPositive {

} else if slope.isNegative {

}
```

This code reads like a sentence.

Another example: custom layout

```
func oldBehaviors(for
    attributes: [UICollectionViewLayoutAttributes]) -> [UIAttachmentBehavior] {
    return animator.behaviors.flatMap {
        $0 as? UIAttachmentBehavior
        }.filter {
            guard let item = $0.items.first
                as? UICollectionViewLayoutAttributes else {
                    return false
            }
            return !attributes.map {
                $0.indexPath
                }.contains(item.indexPath)
    }
}
```

```
func oldBehaviors(for
    attributes: [UICollectionViewLayoutAttributes]) -> [UIAttachmentBehavior] {

    let attributesIndexPaths = attributes.map { $0.indexPath }

    let attachmentBehaviors = animator.behaviors.flatMap {
        $0 as? UIAttachmentBehavior
    }

    let filteredBehaviors = attachmentBehaviors.filter {
        guard let attributes = $0.attributes else {
            return false
        }
        return !attributesIndexPaths.contains(attributes.indexPath)
    }

    return filteredBehaviors
}
```

Keep it small and simple

Write code, not comments

Separate components

Inject dependencies

Avoid over-abstraction

Avoid unnecessary complexity

Thanks!

Jesse Squires
jessesquires.com • @jesse_squires

Swift Weekly Brief:
swiftweekly.github.io • @swiftlybrief