ADAPTING TO CHANGE

DESIGNING FOR MODULARITY AND MAINTAINABILITY IN SWIFT

Jesse Squires
jessesquires.com • @jesse_squires

Where I work

Construction productivity software

It's like GitHub and Xcode for construction

CONSTRUCTION AND SOFTWARE HAVE MANY THINGS IN COMMON

CONNECTING MODEL AND UI LAYERS

FORGETTING TO IMPLEMENT ACCESSIBILITY FEATURES


CLIENT-SEVER COMMUNICATION
MANY DIFFERENT TEAMS WORKING TOGETHER

WHEN A NEW FEATURE HAS USER PRIVACY OR USABILITY ISSUES

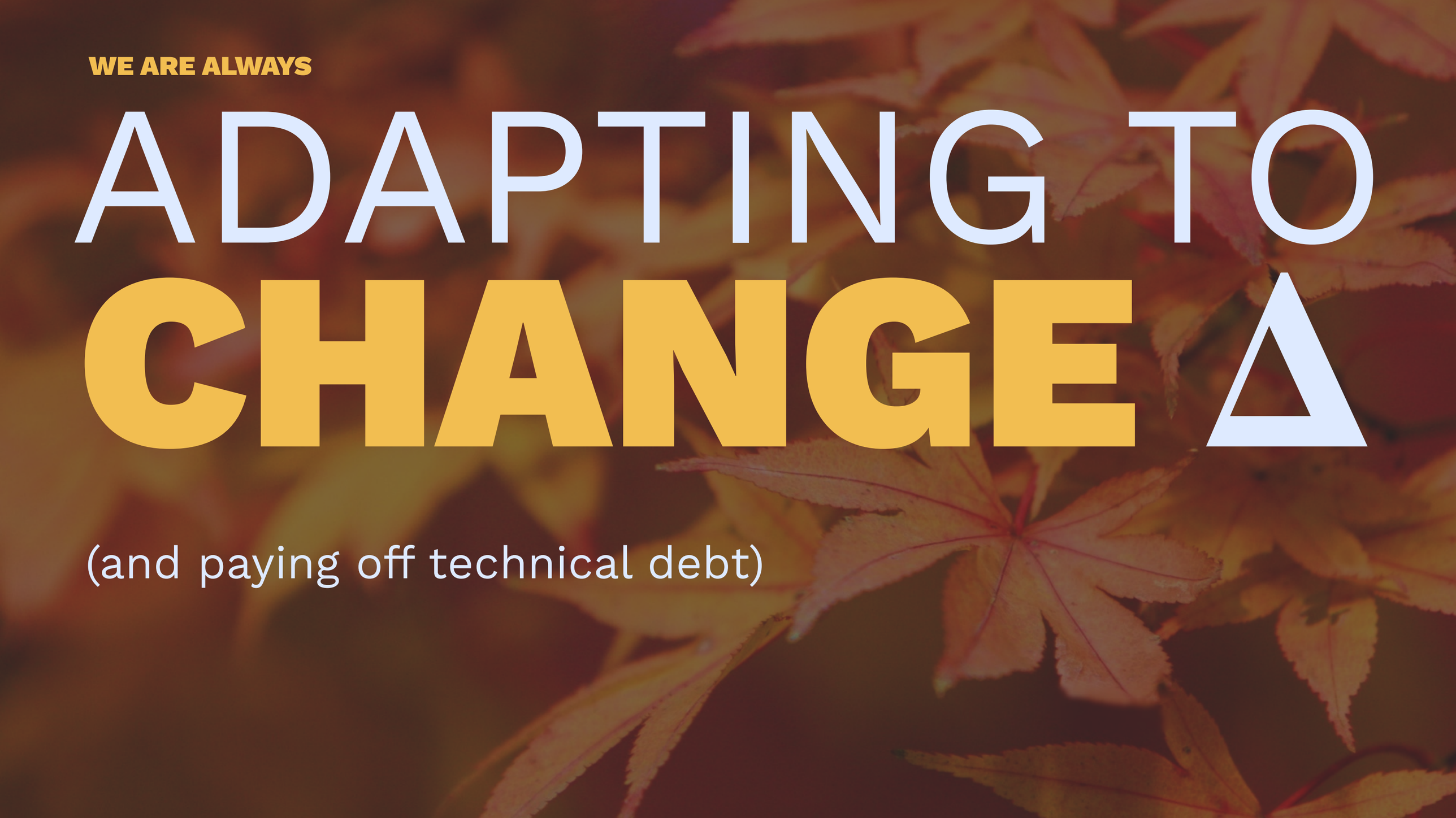
WHEN BAD ARCHITECTURE AND DESIGN DECISIONS CREATE BUGS

DESIGNING

BUILDING

PROTOTYPING

VERIFYING

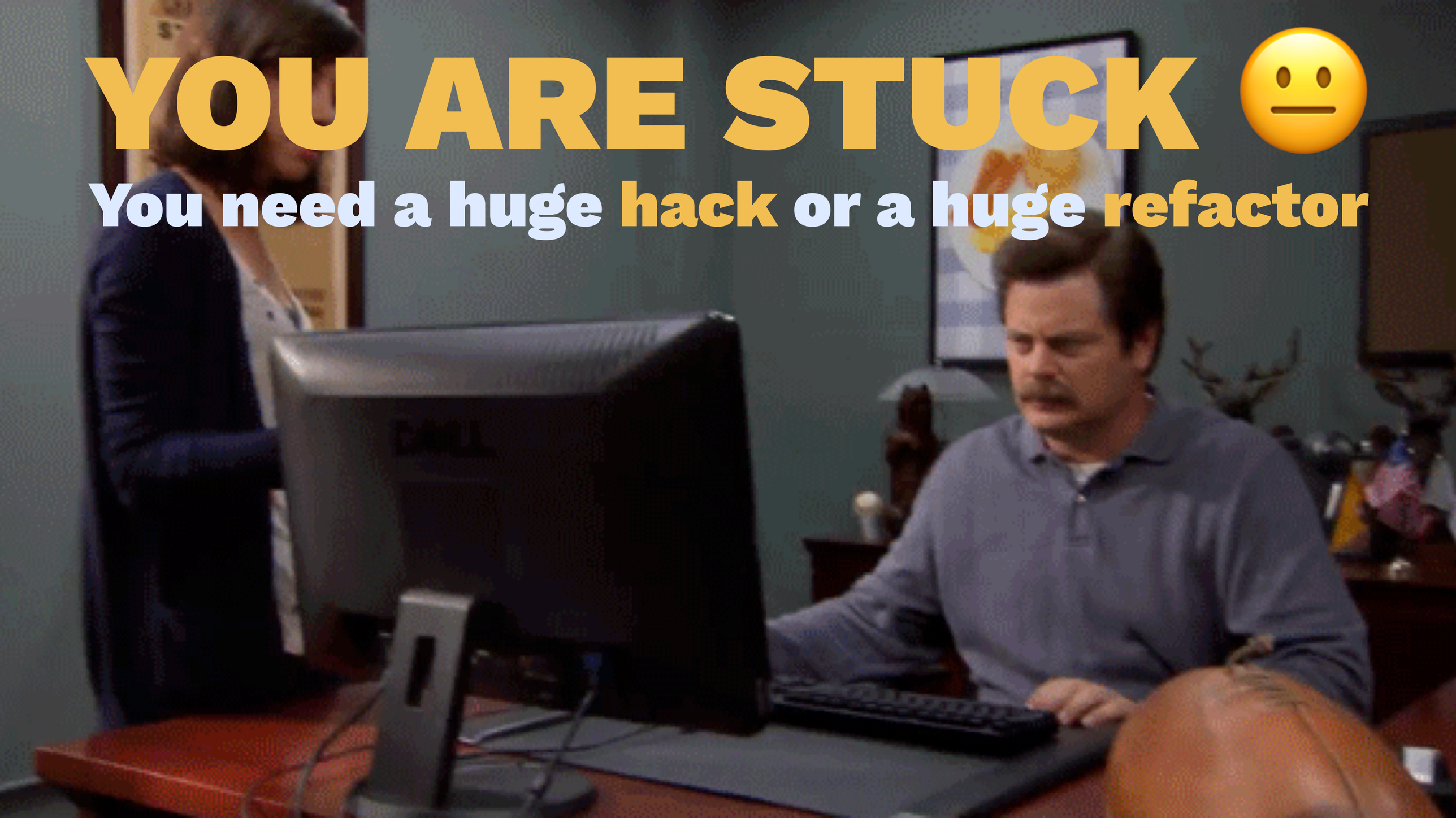

WE ARE ALWAYS
ADAPTING TO
CHANGE Δ
(and paying off technical debt)


So, you're writing some sweet codes 🤓


And everything is going great 😎

Then you hit a wall

(but, not because the Swift compiler is slow)


YOU ARE STUCK 😐
You need a huge hack or a huge refactor

How can we prevent* this?

We can't! LOLOLOLOLOLOLOL 🤡

But, we can try.

* Make it slightly less terrible

VIPER

FLUX

WTF

BBQ

MVC

MVP

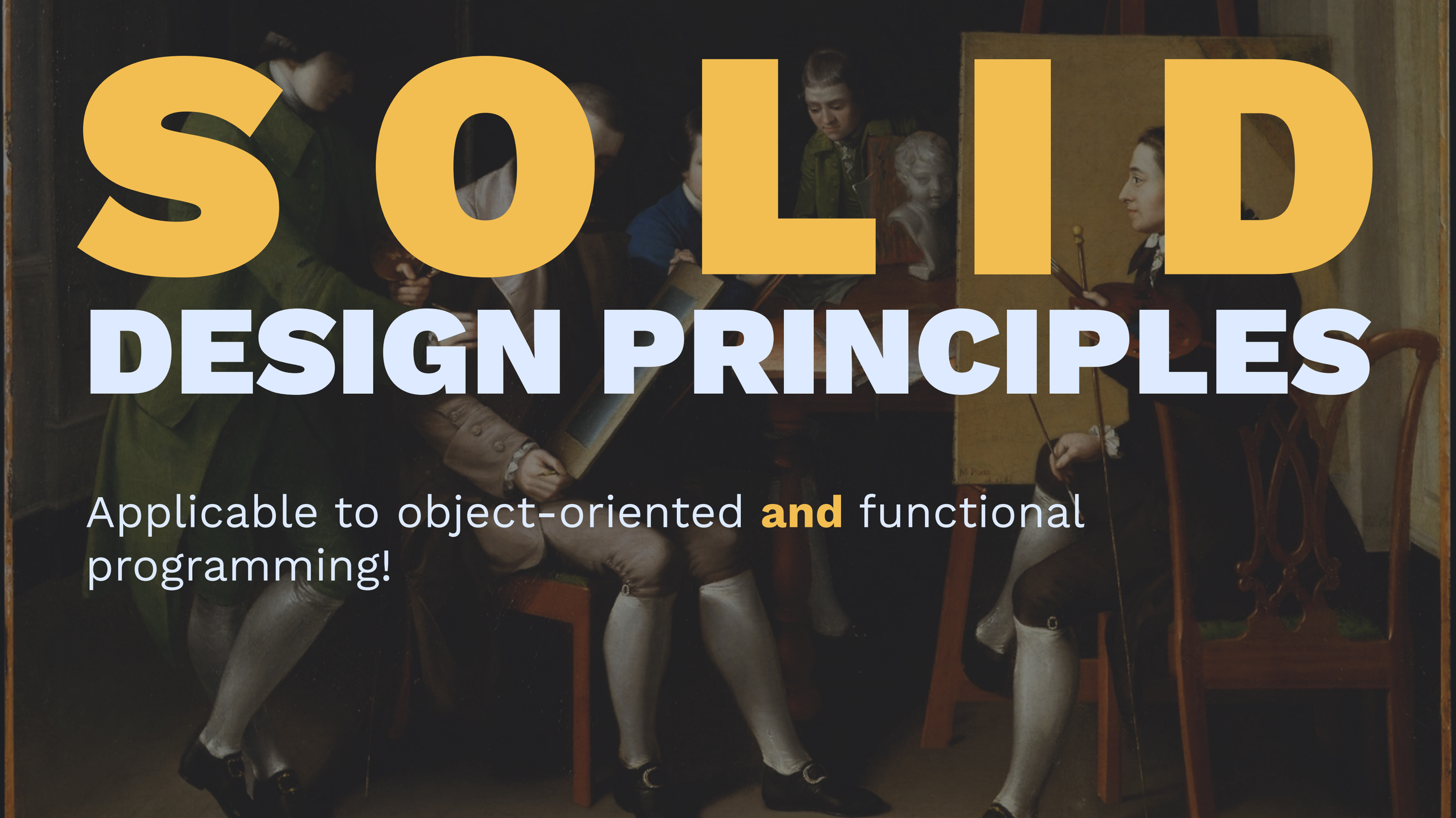
FRP

OMG

MVVMVC

MVVM

All of these architectures have the same goals **SEPARATION OF CONCERNS** AND **CLEAR DATA FLOW**



SOLID
DESIGN PRINCIPLES

Applicable to object-oriented **and** functional programming!

SINGLE RESPONSIBILITY

OPEN / CLOSED

LISKOV SUBSTITUION

INTERFACE SEGREGATION

DEPENDENCY INVERSION

Single responsibility

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)
}
```

Single responsibility ❌

```
extension UIImage {
    func cropFilterTransformAndExportPNGData(
            size: CGSize?,
            filter: CIFilter?,
            transform: CGAffineTransform?) -> Data
}

let png = img.cropFilterTransformAndExportPNGData(size: s,
                                                  filter: f,
                                                  transform: t)
```

Single responsibility ✅

Easy to **reason** about each small function **individually**

```
extension UIImage {
    func crop(to size: CGSize) -> UIImage

    func transform(_ t: CGAffineTransform) -> UIImage

    func filter(_ f: CIFilter) -> UIImage

    func asPNGData() -> Data
}

let pngData = img.crop(to: size).transform(t).filter(f).asPNGData()
```

Open for extension closed for modification

How can we change behavior without modifying a type?

— Subclass 😭

— Inject dependency
  - protocols
  - closures

Open / closed

Using enums as configuration ❌

```
enum MessagesTimestampPolicy {
    case all
    case alternating
    case everyThree
    case everyFive
}

class MessagesViewController: UIViewController {
    var timestampPolicy: MessagesTimestampPolicy
}
```

Need to implement switch statements **everywhere** 😭
Every time you add a new `case` you have to update all `switch` statements

```
class MessagesViewController: UIViewController {

    func timestampFor(indexPath: IndexPath) -> Date? {
        switch self.timestampPolicy {
            case .all:
            // ...
            case .alternating:
            // ...
            case .everyThree:
            // ...
            case .everyFive:
            // ...
        }
    }

    // ...
}
```

Open / closed

A simpler, more flexible design: **configuration objects**

```
struct MessagesTimestampConfig {
    let timestamp: (IndexPath) -> Date?

    let textColor: UIColor
    let font: UIFont
}

func applyTimestampConfig(_ config: MessagesTimestampConfig)
```

Liskov substitution

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

Liskov substitution

🤢 Adding a new message type makes program incorrect

```
class Message { }
class TextMessage: Message { }
class PhotoMessage: Message { }

if model is TextMessage {
    // do text things
} else if model is PhotoMessage {
    // do photo things
} else {
    fatalError("Unexpected type \(model)")
}
```

Liskov substitution

```
enum MessageContent { /* ... */ }

protocol Message {
    var content: MessageContent { get }
}

class TextMessage: Message {
    var content: MessageContent { return self.text }
}

class PhotoMessage: Message {
    var content: MessageContent { return self.photo }
}

let content = message.content
// use content...
```

Interface segregation

Use many specific interfaces, rather than one general purpose interface

```
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

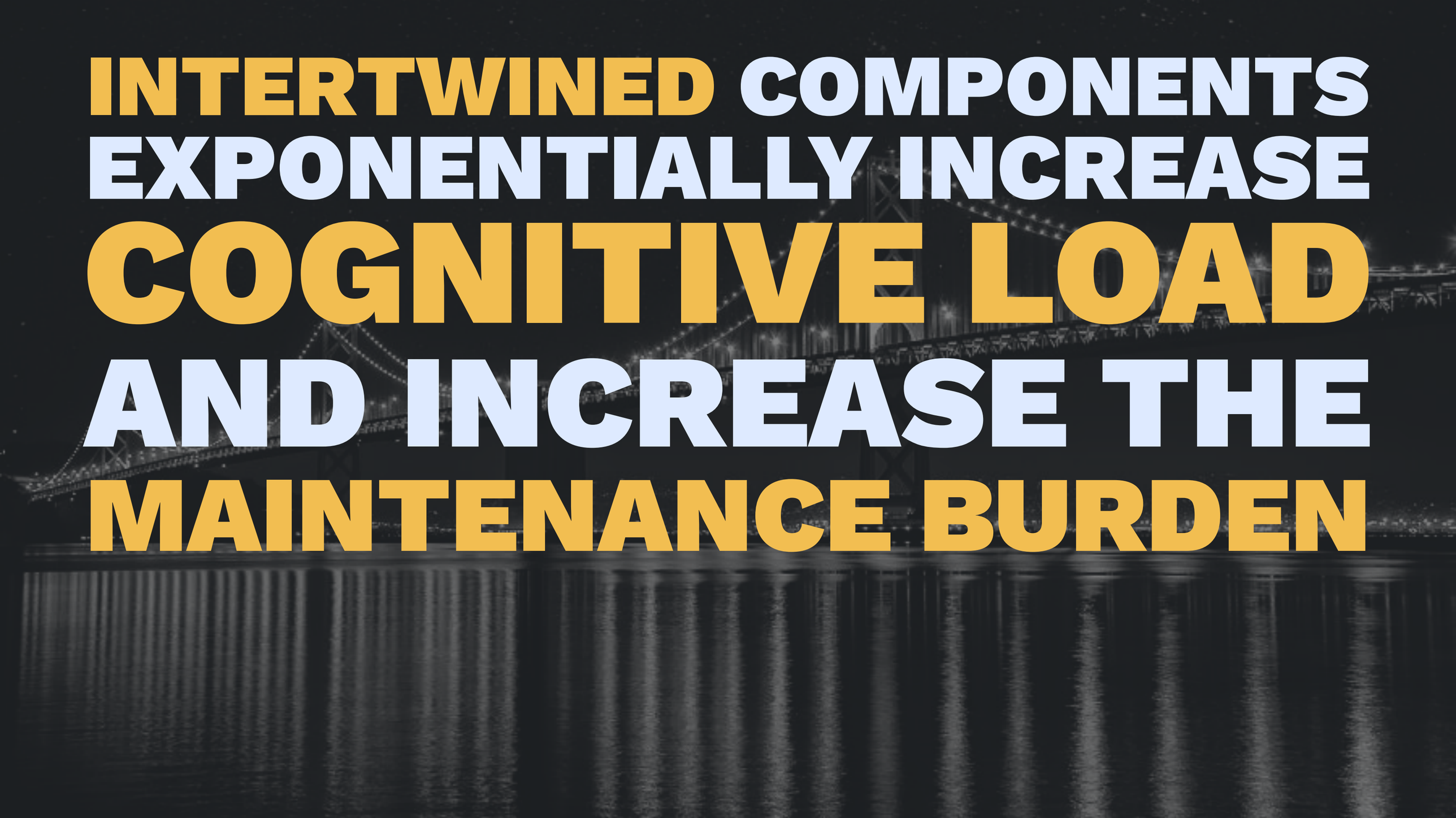
Write against interfaces, not concrete types
Decouple via protocols and dependency injection

```
class MyViewController: UIViewController {

    let sessionManager: SessionManagerProtcool

    init(sessionManager: SessionManagerProtcool = SessionManager.shared) {
        self.sessionManager = sessionManager
    }

    // ...
}
```


INTERTWINED COMPONENTS
EXPONENTIALLY INCREASE
COGNITIVE LOAD
AND INCREASE THE
MAINTENANCE BURDEN

FINALLY

LET'S TALK ABOUT CLARITY

"It was hard to write, it should be hard to read."

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 code like this

```
if slope == 0 {
    // slope is a horizontal line
    // do something...

} else if slope.isNaN {
    // slope is undefined, a vertical line
    // handle this case...

} else if slope > 0 {
    // positive slope

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

Or, we could add extensions for our **specific domain**

```
extension FloatingPoint {

    var isUndefined: Bool { return isNaN }

    var isHorizontal: Bool { return isZero }
}

extension SignedNumeric where Self: Comparable {

    var isPositive: Bool { return self > 0 }

    var isNegative: Bool { return self < 0 }
}
```

And then **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: dynamic collection layout

```
func getBehaviors(for
     attributes: [UICollectionViewLayoutAttributes]) -> [UIAttachmentBehavior] {
    // get the attachment behaviors
    return self.animator.behaviors.flatMap {
        $0 as? UIAttachmentBehavior
        }.filter {
            // remove non-layout attribute items
            guard let item = $0.items.first
                as? UICollectionViewLayoutAttributes else {
                    return false
            }
            // get attributes index paths
            // see if item index path is included
            return !attributes.map {
                $0.indexPath
                }.contains(item.indexPath)
    }
}
```

```swift
extension UIAttachmentBehavior {
    var attributes: UICollectionViewLayoutAttributes? { /* ... */ }
}

extension UIDynamicAnimator {
    var attachmentBehaviors: [UIAttachmentBehavior] { /* ... */ }
}

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

    let attributesIndexPaths = attributes.map { $0.indexPath }
    let attachmentBehaviors = self.animator.attachmentBehaviors

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

    return filteredBehaviors
}
```

WHEN YOU START TO WRITE

A COMMENT

SEE IF YOU CAN WRITE

CODE INSTEAD

COMMENTS CAN GET STALE BUT CODE DOESN'T LIE

```
// show prompt if user has permissions
if !user.hasPermissions {
    displayPrompt()
}
```

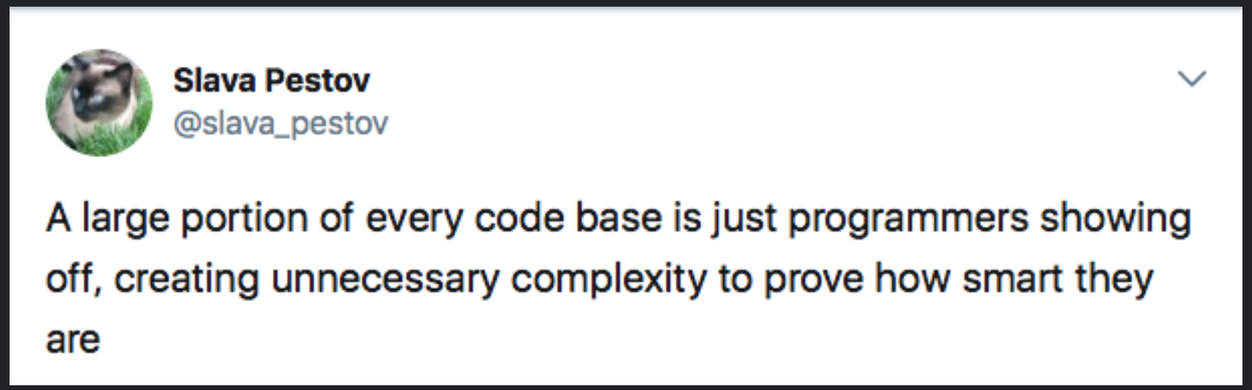
🤔

Only add comments to truly exceptional or non-obvious code

```
// Intentionally not a struct!
// Using immutable reference types for performance reasons
final class Message {
    let uid: UUID
    let timestamp: Date
    let sender: User
    let text: String
}
```


Keep it small and simple
Write code, not comments
Separate components
Inject dependencies
Avoid over-abstraction
Avoid unnecessary complexity

When programmers get bored, they find ways to make easy things more complicated. 🙃




Thanks!
Jesse Squires
@jesse_squires • jessesquires.com
Swift Unwrapped
@swift_unwrapped
Swift Weekly Brief
@swiftlybrief • swiftweekly.github.io