

SOLID Design Principles



Jon Reid
@qcoding

Single Responsibility Principle

Open-Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Principle

Principle

Principle

Principle

Principle

Principle:

- Idea you conform to
- Guidepost for behavior
- Gravity
- Axiom / Fundamental truth
- Religion

Dogma: But what's it *for*?

- Religious/moral principles
- Dogma: “a principle or set of principles laid down by an authority as incontrovertibly true”
- “Good” vs. “Bad”
- “Good for _____”
- “Bad for _____”
- SOLID, huh! What is good for?

Examples of when something was harder to change than expected:

- String parsing in Swift
- Design around Notch!
- Extensions?
- Libraries
- Breaking circular dependencies

Consequences of when something was harder to change than expected:

- Frustration
- Breakage
- Time
- Unexpected compromise
- Conflict
- Product canceled, entire team fired

SOLID: Roots in Object Oriented Programming

But with Swift embracing Functional Programming...

OBJECT ORIENTED PROGRAMMING



SOLID questions

- What is it good for?
- Is it still relevant for Swift?

Single Responsibility Principle

Open-Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Open-Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Single Responsibility Principle

Open-Closed Principle

A module should be open for extension, but closed for modification.

```
struct Authenticator {  
    let publicKey = "68a325daae67ba95cf3ef28c2e1684c8"  
    let privateKey = "4bc5be0d70ea1ad761fa11c4dc4a3fb649e"  
  
    func hash(timestamp: String) -> String {  
        let hash = md5(timestamp + privateKey + publicKey)  
        return "&hash=\(hash)"  
    }  
}
```

Examples of OCP violations:

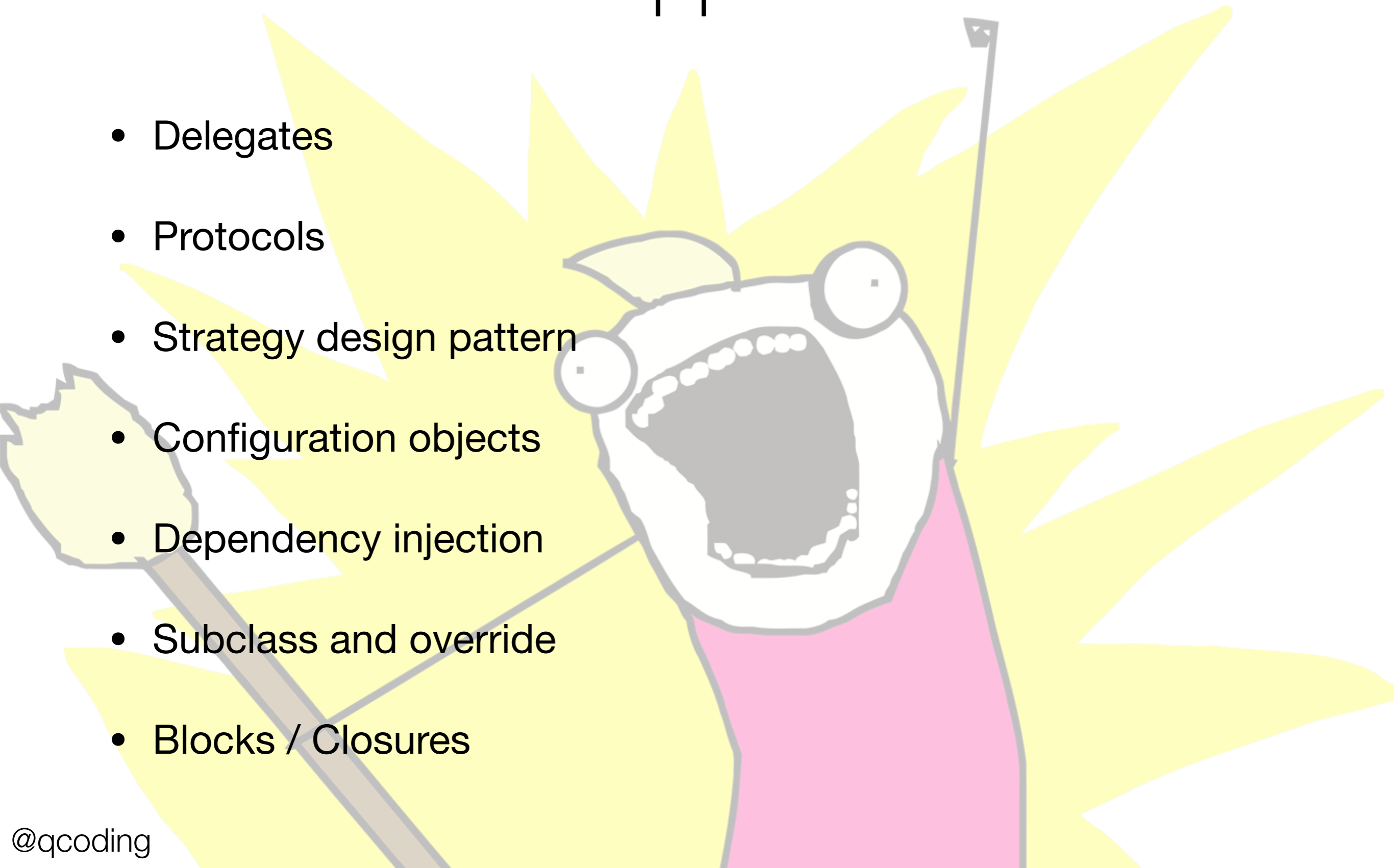
- Have to change the guts of a thing
- Server URL: Staging vs. production
- URL versioning

OCP techniques:

- Delegates
- Protocols
- Strategy design pattern
- Configuration objects
- Dependency injection
- Subclass and override
- Blocks / Closures

Over-application

- Delegates
- Protocols
- Strategy design pattern
- Configuration objects
- Dependency injection
- Subclass and override
- Blocks / Closures





OPEN CLOSED PRINCIPLE

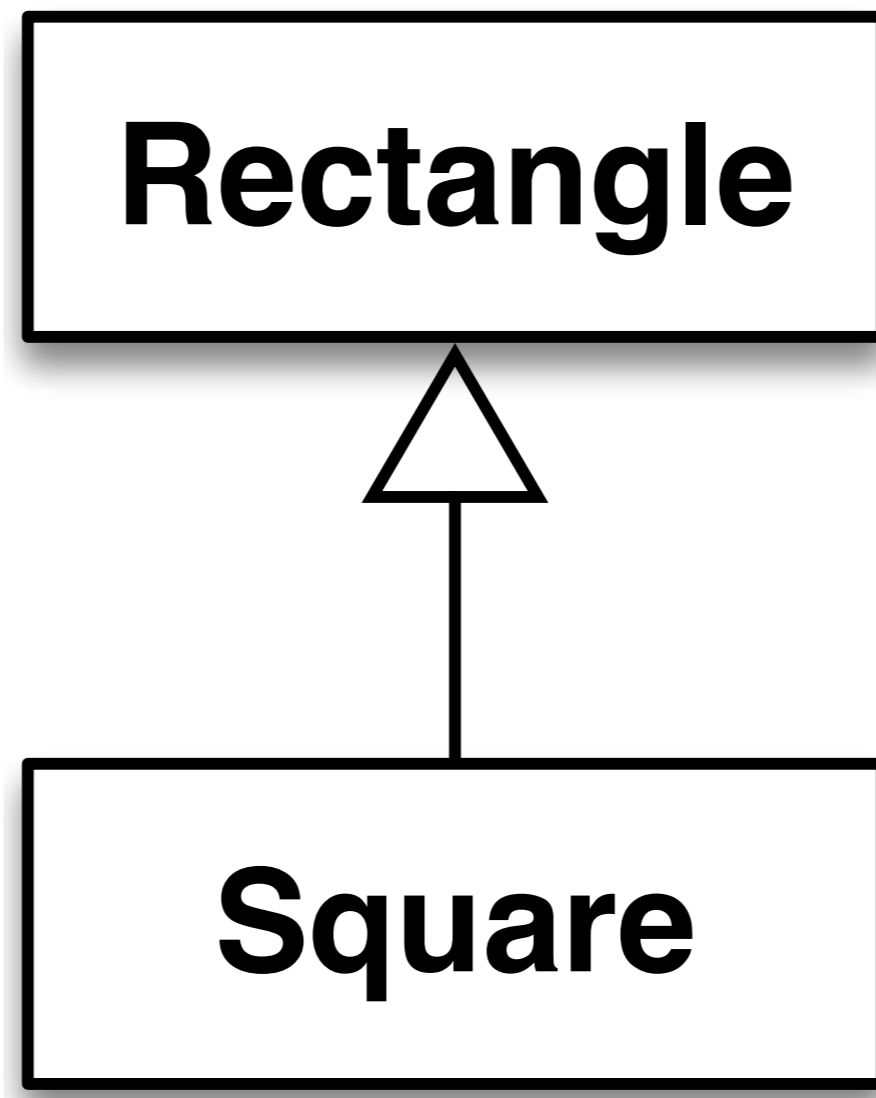
Brain surgery is not necessary when putting on a hat.

Liskov Substitution Principle

Subclasses should be substitutable for their base classes.



Liskov Substitution Principle



Examples of LSP violations:

- Single table inheritance & battling notifications
- Radio button: UIControl clear
- Sequences, collections
- NSMutableArray: NSArray
- All related to mutability

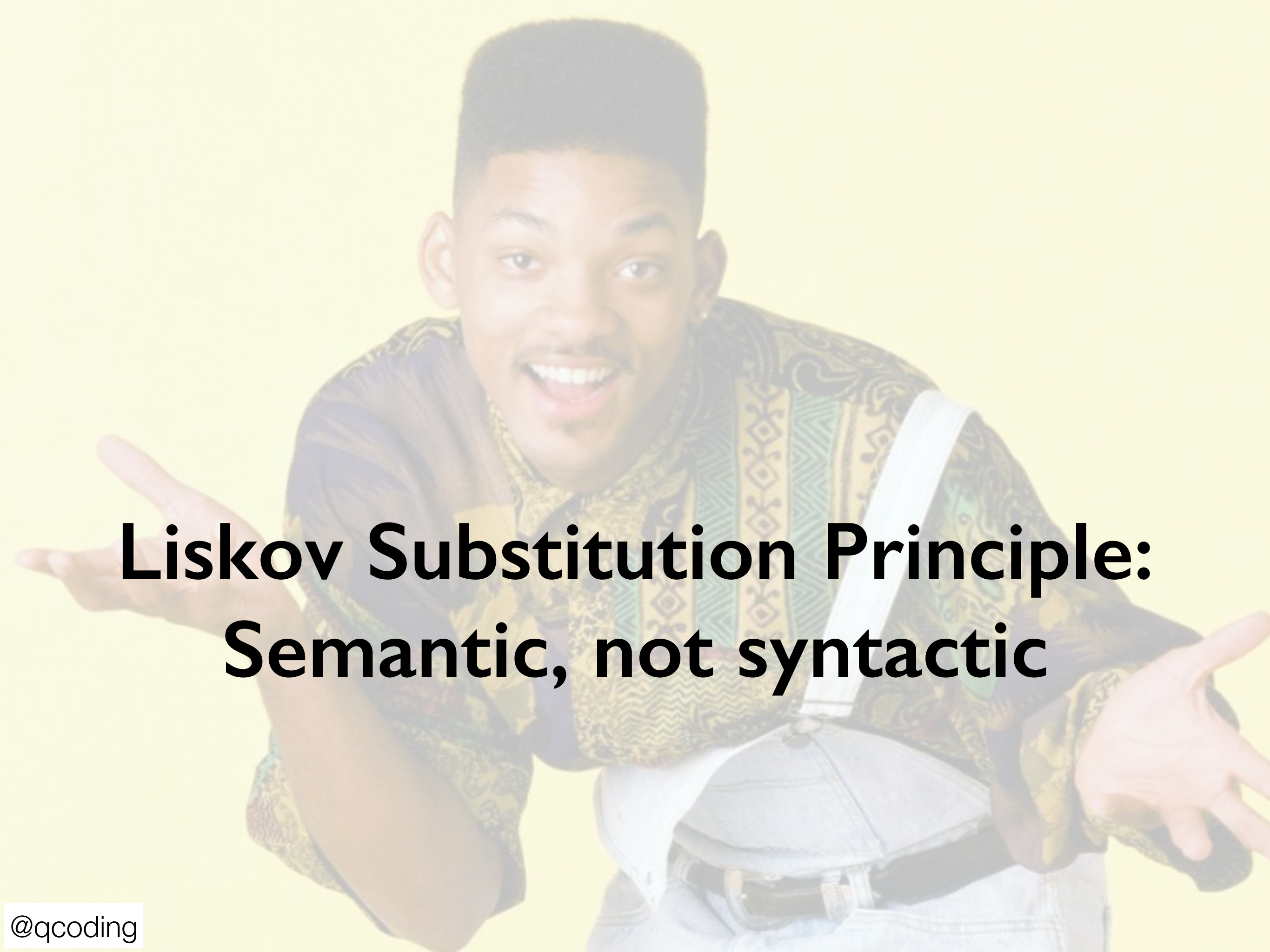
Examples of LSP violations:

```
required init?(coder _: NSCoder) {  
    fatalError("init(coder:) has not been implemented")  
}
```

```
required init?(coder _: NSCoder) {  
    fatalError("(◡ ◯ ◻ ◯)◡ ◡")  
}
```

LSP: Only applies to subclasses?

- I lied: It's not subclassing, it's sub-typing
- Implement a protocol
- "I'm going to implement this protocol, but leave these blank"

A background image of Will Smith from the movie 'The Matrix', wearing a patterned shirt and a white strap over his shoulder, with his hands outstretched in a 'bullet time' pose. The image is semi-transparent, allowing the text to be overlaid.

Liskov Substitution Principle: Semantic, not syntactic



LISKOV SUBSTITUTION

If it looks like a duck, quacks like a duck, but needs batteries —
you probably have the wrong abstraction.

Interface Segregation Principle

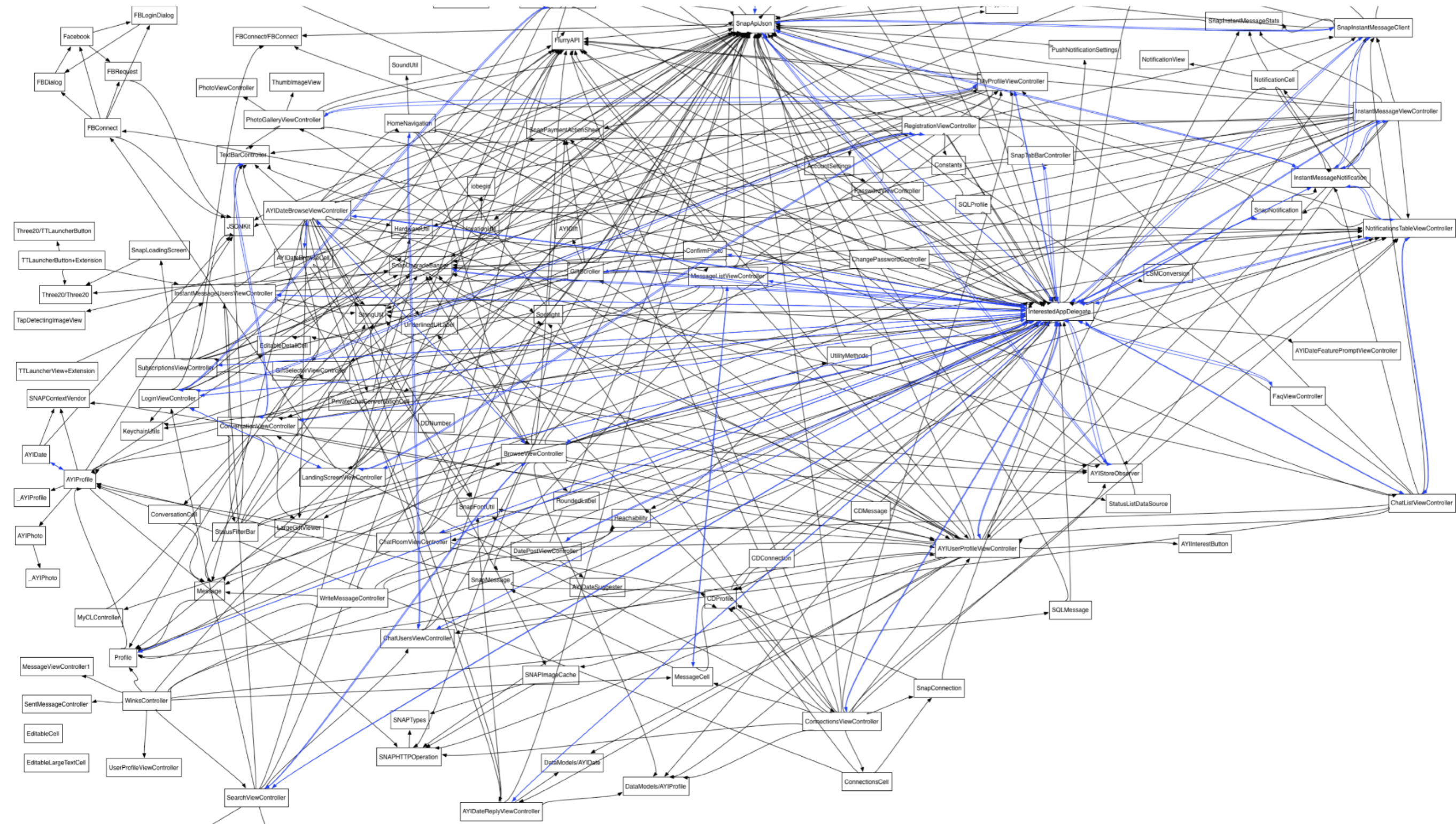
Many client-specific interfaces are better than one general-purpose interface.



Consequences of “fat” classes:

- Adaptability
- Mock ALL THE THINGS?
- Coupling too much stuffs
- Hard to read
- Hard to share

Fat classes affect build times



DATA STRUCTURE SEGREGATION

It matters, too!



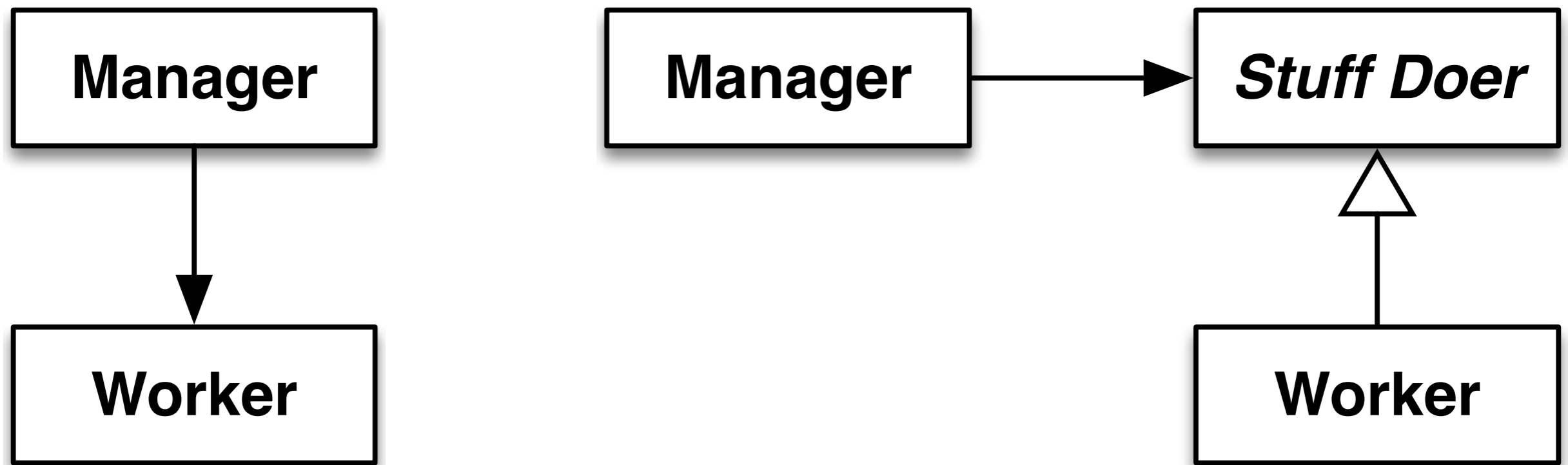


INTERFACE SEGREGATION

Tailor interfaces to individual clients' needs.

Dependency Inversion Principle

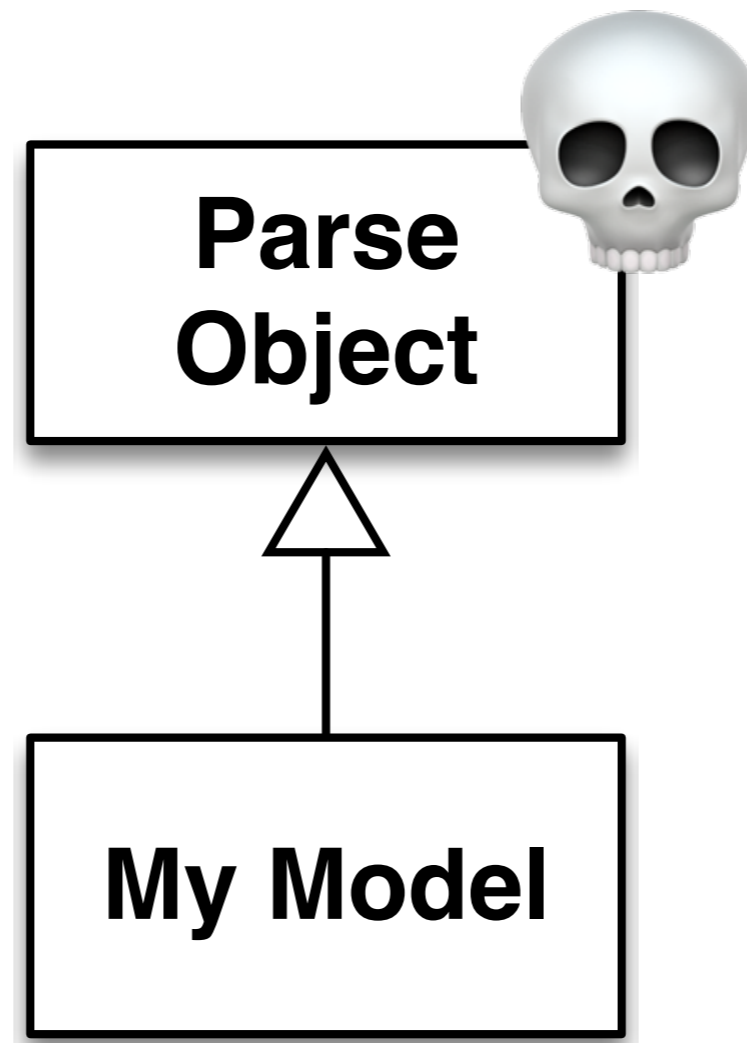
High-level modules should not depend on low-level modules.
Both should depend on abstractions.



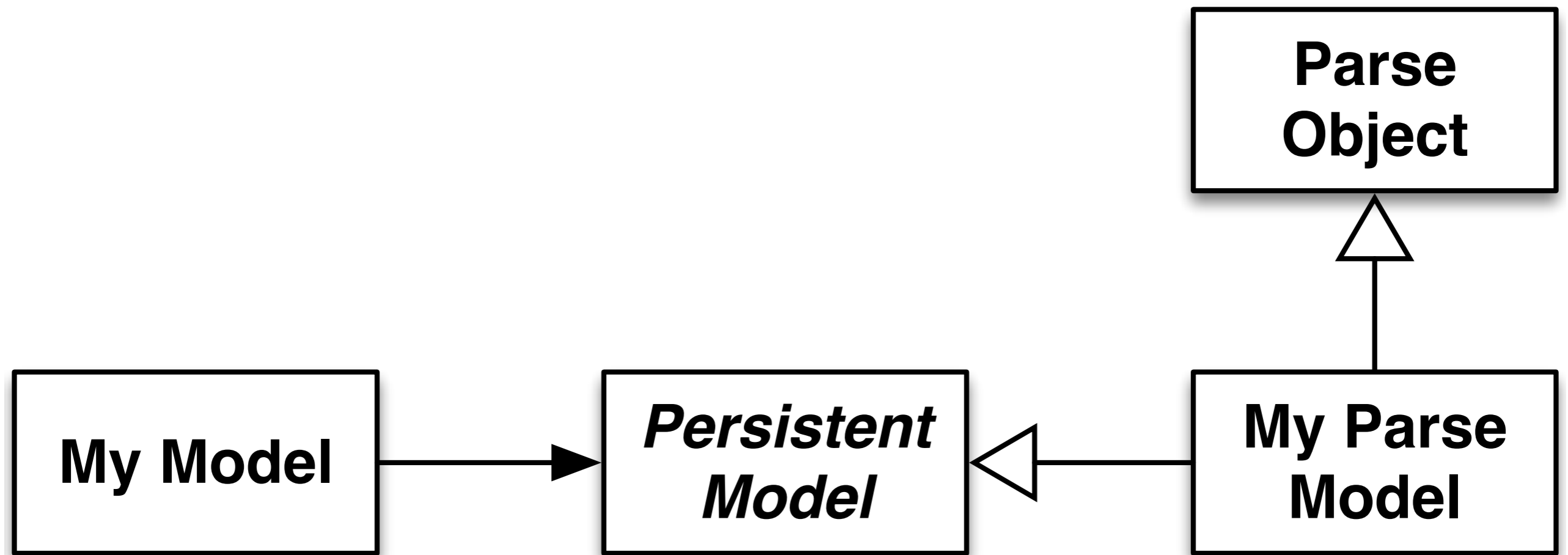
Examples of DIP violations:

- View talk directly to model
- Swift 2
- Storyboards
- CocoaPod

High-impact DIP violation:



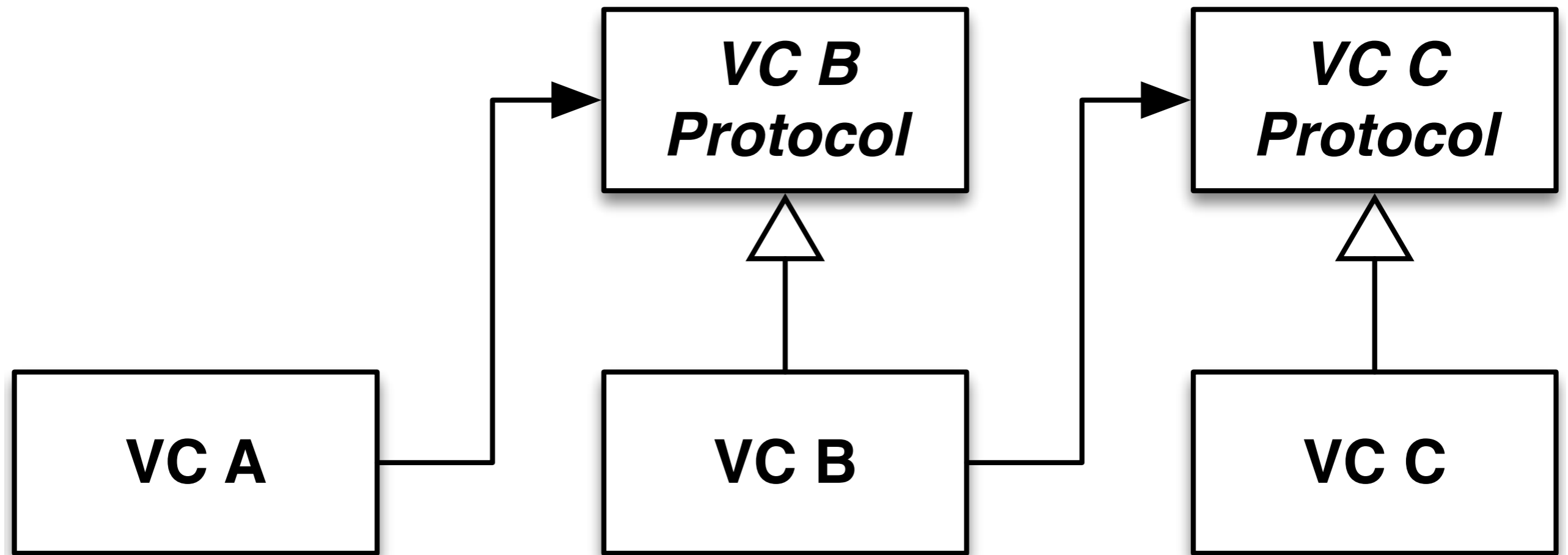
Introducing an abstraction in the middle



Routing dependencies: before



Routing dependencies: after





DEPENDENCY INVERSION

Would you solder a lamp directly to the electrical source?

Single Responsibility Principle

A module should have one and only one reason to change.



Examples of “technical” reasons to change

- Need to improve performance
- Swift
- New framework
- Testability

Examples of “business” reasons to change

- Requirements change
- Accessibility
- Analytics
- Localization
- Security vulnerabilities



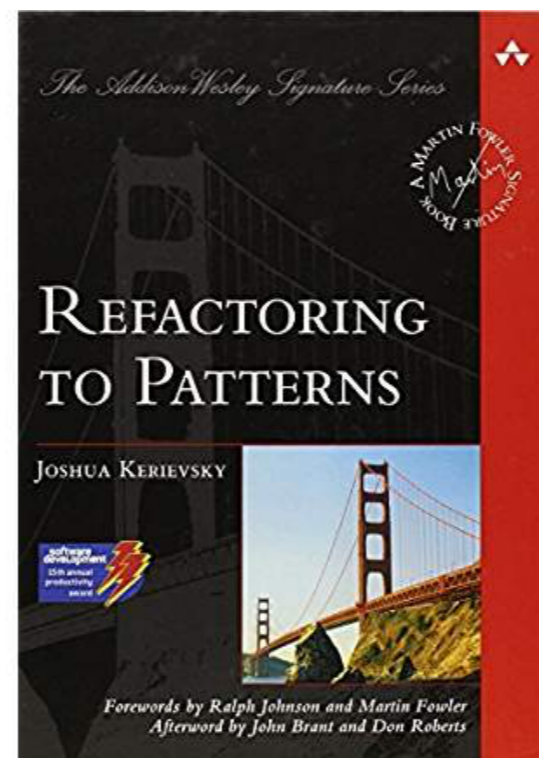
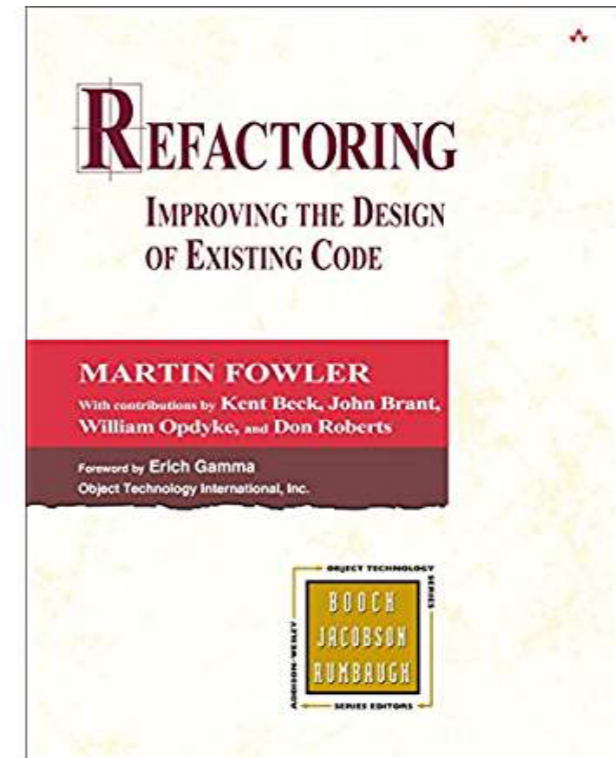
SINGLE RESPONSIBILITY
Avoid tightly coupling your tools together.



WHAT IS SOLID FOR?

Keep the end in mind.

EVOLUTIONARY DESIGN: “Responding to change”

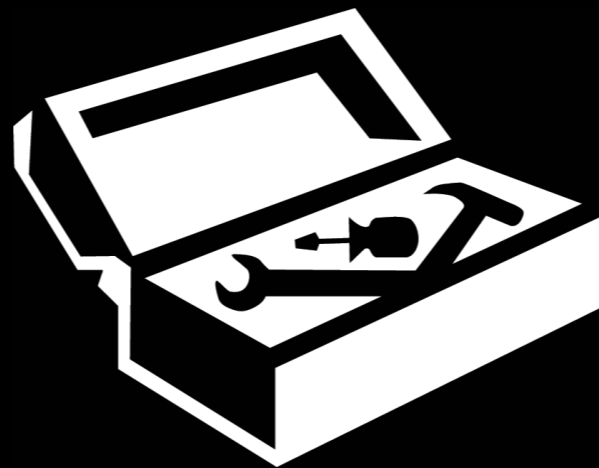




SWIFT PROTOCOLS!

A single language construct has many purposes.

***Go make faster-building
SOLID (but soft) Swift!***



Slides and show notes:
qualitycoding.org/talk/swiftnw2017