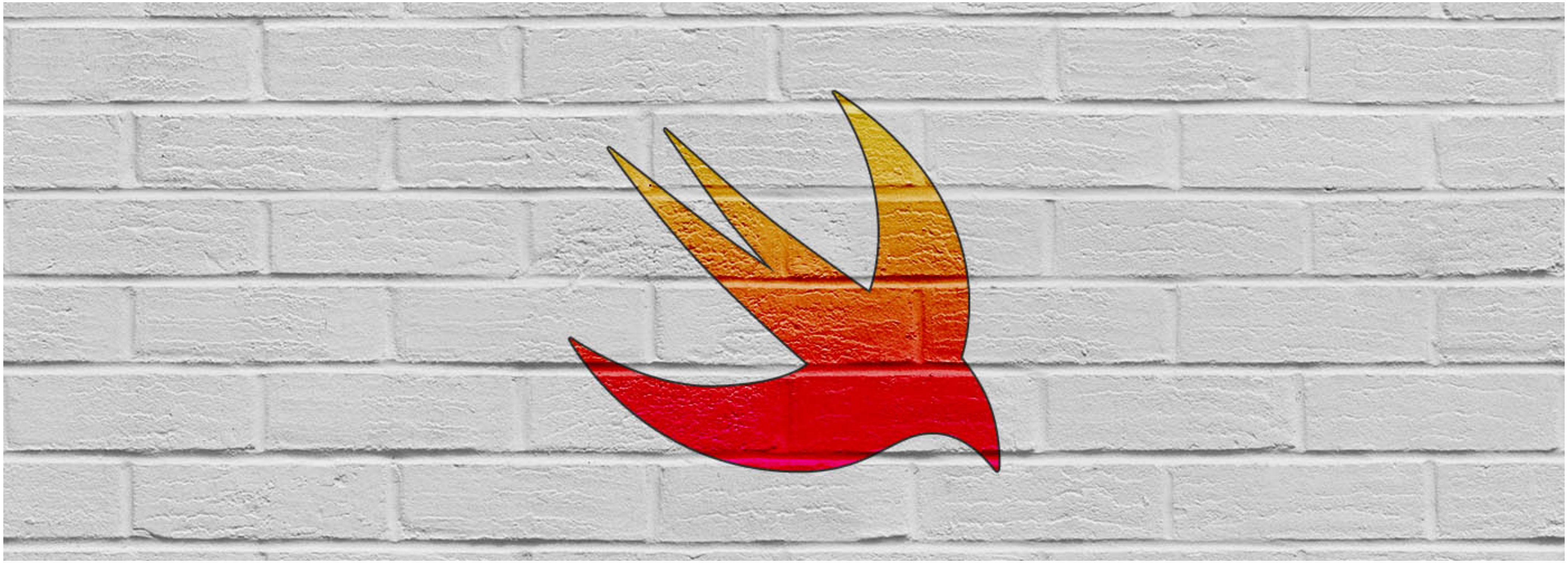

SOLID

with Swift

Before Start…

- The SOLID principles are not rules. They are not laws. They are not perfect truths.
- Good principle is *good advice*, but it's not a pure truth. Like the statements on the order of "*An apple a day keeps the doctor away.*"
- They are common-sense solutions to common problems. They are common-sense disciplines that can *help you stay out of trouble*.
- They have been observed to work in many cases; but there is no proof that they always work, nor any proof that they should always be followed.
- Knowledge of the principles and patterns gives you the justification to decide **when** and **where** to apply them.

Principles will not turn a bad programmer into a good programmer.

Principles have to be <u>applied with judgement</u>. If they are applied by rote it is just as bad as if they are not applied at all.

You must be smart enough to understand when apply these principles.

Practice, practice, practice, practice…

Be prepared to make *lots* of mistakes.

— Robert C. Martin (Uncle Bob) —

SOLID

- SOLID is an acronym named by Robert C. Martin (Uncle Bob).
- With these principles, you can solve the main problems of a bad architecture:
  - Fragility: A change may break unexpected parts—it is very difficult to detect if you don’t have a good test coverage.
  - Immobility: A component is difficult to reuse in another project—or in multiple places of the same project—because it has too many coupled dependencies.
  - Rigidity: A change requires a lot of efforts because affects several parts of the project.

Topics

- **S**ingle Responsibility Principle
- **O**pen-Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

The Single Responsibility Principle (SRP)

THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A CLASS TO CHANGE.

- Every time you create/change a class, you should ask yourself: How many responsibilities does this class have

- This principle helps you to keep your classes as clean as possible.

The Single Responsibility Principle (SRP)

- Example :-

```swift
class Handler {

    func handle() {
        let data = requestDataToAPI()
        let array = parse(data: data)
        saveToDB(array: array)
    }

    private func requestDataToAPI() -> Data {
        // send API request and wait the response
    }

    private func parse(data: Data) -> [String] {
        // parse the data and create the array
    }

    private func saveToDB(array: [String]) {
        // save the array in a database (CoreData/Realm/...)
    }
}
```

How many responsibilities does this class have?

The Single Responsibility Principle (SRP)

- Apply SRS :-

```swift
class Handler {

    let apiHandler: APIHandler
    let parseHandler: ParseHandler
    let dbHandler: DBHandler

    init(apiHandler: APIHandler,
         parseHandler: ParseHandler,
         dbHandler: DBHandler)
    {
        self.apiHandler = apiHandler
        self.parseHandler = parseHandler
        self.dbHandler = dbHandler
    }

    func handle() {
        let data = apiHandler.requestDataToAPI()
        let array = parseHandler.parse(data: data)
        dbHandler.saveToDB(array: array)
    }
}
```

```swift
class APIHandler {

    func requestDataToAPI() -> Data {
        // send API request
        // and wait the response
    }
}

class ParseHandler {

    func parse(data: Data) -> [String] {
        // parse the data and
        // create the array
    }
}

class DBHandler {

    func saveToDB(array: [String]) {
        // save the array in a database...)
    }
}
```

The Open-Closed Principle (OCP)

SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS, ETC.) SHOULD BE **OPEN** FOR EXTENSION, BUT **CLOSED** FOR MODIFICATION.

- If you want to create a class easy to maintain, it must have two important characteristics:
  - **Open for extension**: You should be able to extend or change the behaviour of a class without efforts.
  - **Closed for modification**: You must extend a class without changing the implementation.

The Open-Closed Principle (OCP)

- Example :-

```swift
class Logger {

    func printData() {
        let cars = [
            Car(name: "Batmobile", color: "Black"),
            Car(name: "SuperCar", color: "Gold"),
            Car(name: "FamilyCar", color: "Grey")
        ]

        cars.forEach { car in
            print(car.printDetails())
        }
    }
}
```

```swift
class Car {
    let name: String
    let color: String

    init(name: String, color: String) {
        self.name = name
        self.color = color
    }

    func printDetails() -> String {
        return "I'm \(name) and my color is \(color)"
    }
}
```

The Open-Closed Principle (OCP)

- Example :-

```
class Logger {

    func printData() {
        let cars = [
            Car(name: "Batmobile", color: "Black"),
            Car(name: "SuperCar", color: "Gold"),
            Car(name: "FamilyCar", color: "Grey")
        ]

        cars.forEach { car in
            print(car.printDetails())
        }

        let bicycles = [
             Bicycle(type: "BMX"),
             Bicycle(type: "Tandem")
         ]

         bicycles.forEach { bicycles in
             print(bicycles.printDetails())
         }
    }
}
```

Logger change every time we add new class.

```
class Car {
    let name: String
    let color: String

    init(name: String, color: String) {
        self.name = name
        self.color = color
    }

    func printDetails() -> String {
        return "I'm \(name) and my color is \(color)"
    }
}
```

```
class Bicycle {
    let type: String

    init(type: String) {
        self.type = type
    }

    func printDetails() -> String {
        return "I'm a \(type)"
    }
}
```

The Open-Closed Principle (OCP)

- Apply OCP :-

```
protocol Printable {
    func printDetails() -> String
}


class Logger {

    func printData(p: Printable) {
       cars.forEach { car in
           print(car.printDetails())
        }
    }
}
```

open to extend

closed to modify

```
let vehicles: [Printable] = [
        Car(name: "Batmobile", color: "Black"),
        Car(name: "SuperCar", color: "Gold"),
        Car(name: "FamilyCar", color: "Grey"),
        Bicycle(type: "BMX"),
        Bicycle(type: "Tandem")
]

Logger().printData(p: vehicles)
```

```
class Car: Printable {
    let name: String
    let color: String

    init(name: String, color: String) {
        self.name = name
        self.color = color
    }

    func printDetails() -> String {
        return "I'm \(name) and my color is \(color)"
    }
}

class Bicycle: Printable {
    let type: String

    init(type: String) {
        self.type = type
    }

    func printDetails() -> String {
        return "I'm a \(type)"
    }
}
```

The Liskov Substitution Principle (LSP)

FUNCTIONS THAT USE POINTERS OR REFERENCES TO BASE CLASSES *MUST BE ABLE TO USE* OBJECTS OF DERIVED CLASSES *WITHOUT KNOWING IT*.

- Inheritance may be dangerous and you should use *composition over inheritance* to avoid a messy codebase. Even more if you use inheritance in an improper way.

- This principle can help you to use inheritance without messing it up.

The Liskov Substitution Principle (LSP)

- Preconditions Changes Example :-

```swift
class Handler {

    func save(string: String) {
        // save string to cloud
    }
}

class FilterHandler: Handler {
    override func save(string: String) {
        guard string.count > 5 else {
            return
        }
    }
}
```

This example breaks LSP because, in the subclass, we add the precondition that `string` must have a length greater than 5.

A client of `Handler` doesn't expect that `FilteredHandler` has a different precondition, since it should be the same for `Handler` and all its subclasses.

The Liskov Substitution Principle (LSP)

- Preconditions Changes Example :-

```swift
class Handler {

    func save(string: String, minChars: Int = 0) {
        guard string.characters.count >= minChars else {
            return
        }

        // Save string in the Cloud
    }
}
```

We can solve this problem getting rid of `FilteredHandler` and adding a new parameter to inject the minimum length of characters to filter:

The Liskov Substitution Principle (LSP)

- Postconditions Changes Example :-

```
class Rectangle {

    var width: Float = 0
    var length: Float = 0

    var area: Float {
        return width * length
    }
}

class Square: Rectangle {

    override var width: Float {
        didSet {
            length = width
        }
    }
}
```

This approach break LSP because if the client has the current method:

```
func printArea(of rectangle: Rectangle) {
    rectangle.length = 5
    rectangle.width = 2
    print(rectangle.area)
}


let rectangle = Rectangle()
printArea(of: rectangle) // 10

let square = Square()
printArea(of: square) // 4
```

The Liskov Substitution Principle (LSP)

- Postconditions Changes Example :-

```swift
import Foundation

protocol Polygon {
    var area: Float { get }
}

class Rectangle: Polygon {

    private let width: Float
    private let length: Float

    init(width: Float, length: Float) {
        self.width = width
        self.length = length
    }

    var area: Float {
        return width * length
    }
}

class Square: Polygon {

    private let side: Float

    init(side: Float) {
        self.side = side
    }

    var area: Float {
        return pow(side, 2)
    }
}
```

```swift
// Client Method

func printArea(of polygon: Polygon) {
    print(polygon.area)
}


// Usage

let rectangle = Rectangle(width: 2, length: 5)
printArea(of: rectangle) // 10

let square = Square(side: 2)
printArea(of: square) // 4
```

We can solve it using a protocol with a method `area`, implemented by `Rectangle` and `Square` in different ways.

Finally, we change the `printArea` parameter type to accept an object which implement `Polygon` protocol

The Interface Segregation Principle (ISP)

CLIENTS SHOULD NOT BE FORCED TO DEPEND UPON INTERFACES THAT THEY DO NOT USE.

- This principle introduces one of the problems of object-oriented programming: the fat interface.

- An interface is called “fat” when has too many members/ methods, which are not cohesive and contains more information than we really want. This problem can affect both classes and protocols.

The Interface Segregation Principle (ISP)

- Fat interface (Protocol)

```
protocol GestureProtocol {
    func didTap()
    func didDoubleTap()
    func didLongPress()
}
```

```
class JustTapButton: GestureProtocol {

    func didTap() {
        // send tap action
    }

    func didDoubleTap() {
        // send double tap action
    }

    func didLongPress() {
        // send long press action
    }
}
```

```
class JustTapButton: GestureProtocol {

  func didTap() {
      // send tap action
  }

  func didDoubleTap() {

  }

  func didLongPress() {

  }
}
```

The Interface Segregation Principle (ISP)

- Fat interface (Protocol) breakdown

```swift
protocol TapProtocol {
    func didTap()
}
```

```swift
protocol DoubleProtocol {
    func didDoubleTap()
}
```

```swift
protocol LongPressProtocol {
    func didLongPress()
}
```

```swift
class GestureButton: TapProtocol,
   DoubleProtocol, LongPressProtocol {

    func didTap() {
        // send tap action
    }

    func didDoubleTap() {
        // send double tap action
    }

    func didLongPress() {
        // send long press action
    }
}
```

```swift
class JustTapButton: TapProtocol {

    func didTap() {
        // send tap action
    }

    func didDoubleTap() {

    }

    func didLongPress() {

    }
}
```

The Interface Segregation Principle (ISP)

- Fat interface (Class)

```swift
class Video {
    var title: String = "My Video"
    var description: String = "This is a beautiful video"
    var author: String = "Marco Santarossa"
    var url: String = "https://marcosantadev.com/my_video"
    var duration: Int = 60
    var created: Date = Date()
    var update: Date = Date()
}


func play(video: Video) {
    // load the player UI
    // load the content at video.url
    // add video.title to the player UI title
    // update the player scrubber with video.duration
}
```

We are injecting too many information in the method `play`, since it needs just `url`, `title` and `duration`.

The Interface Segregation Principle (ISP)

- Fat interface (Class) breakdown (1/2)

```swift
protocol Playable {
    var title: String { get }
    var url: String { get }
    var duration: Int { get }
}

class Video: Playable {
    var title: String = "My Video"
    var description: String = "This is a beautiful video"
    var author: String = "Marco Santarossa"
    var url: String = "https://marcosantadev.com/my_video"
    var duration: Int = 60
    var created: Date = Date()
    var update: Date = Date()
}

func play(video: Playable) {
    // load the player UI
    // load the content at video.url
    // add video.title to the player UI title
    // update the player scrubber with video.duration
}
```

You can solve this problem using a protocol `Playable` *with just the information …*

…which the player needs:

The Interface Segregation Principle (ISP)

- Fat interface (Class) breakdown (2/2)

```
class StubPlayable: Playable {
    var isTitleRead = false

    var title: String {
        self.isTitleRead = true
        return "My Video"
    }

    var duration = 60
    var url: String = "https://marcosantadev.com/my_video"
}

func test_Play_IsUrlRead() {
    let stub = StubPlayable()

    play(video: stub)

    XCTAssertTrue(stub.isTitleRead)
}
```

This approach is very useful also for the unit test. We can create a stub class which implements the protocol `Playable`:

The Dependency Inversion Principle (DIP)

A. HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.

B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.

- DIP is very similar to Open-Closed Principle: the approach to use, to have a clean architecture, is decoupling the dependencies. You can achieve it thanks too abstract layers.

The Dependency Inversion Principle (DIP)

- Tight coupling

```
class FilesystemManager {

    func save(string: String) {
        // Open a file
        // Save the string in this file
        // Close the file
    }
}

class Handler {

    let fm = FilesystemManager()

    func handle(string: String) {
        fm.save(string: string)
    }
}
```

The Dependency Inversion Principle (DIP)

- Loosely coupling

```
protocol Storage {
   func save(string: String)
}



class FilesystemManager: Storage {

    func save(string: String) {
        // Open a file in read-mode
        // Save the string in this file
        // Close the file
    }
}
```

```
class Handler {

    let storage: Storage

    init(storage: Storage) {
        self.storage = storage
    }

    func handle(string: String) {
        storage.save(string: string)
    }
}
```

The Dependency Inversion Principle (DIP)

- Stub in Unit Test

```
protocol Storage {
    func save(string: String)
}



class Handler {

    let storage: Storage

    init(storage: Storage) {
        self.storage = storage
    }

    func handle(string: String) {
        storage.save(string: string)
    }
}
```

```
class StubStorage: Storage {

    var isSavedCalled = false

    func save(string: String) {
        isSavedCalled = true
    }
}


class HandlerTests {

    func test_Handle_IsSaveCalled() {

        let stubStorage = StubStorage()
        let handler = Handler(storage: stubStorage)

        handler.handle(string: "test")

        XCTAssertTrue(stubStorage.isSavedCalled)
    }
}
```

Manifesto for Software Craftsmanship

As aspiring Software Craftsmen we are raising the bar of professional software development by practicing it and helping others learn the craft. Through this work we have come to value:

Not only working software, but also well-crafted software

Not only responding to change, but also steadily adding value

Not only individuals and interactions, but also a community of professionals

Not only customer collaboration, but also productive partnerships

That is, in pursuit of the items on the left we have found the items on the right to be indispensable.

Conclusion

- You have 3 enemies to defeat: Fragility, Immobility and Rigidity. SOLID principles are your weapons.
- If you follow SOLID principles judiciously, you can increase the quality of your code. Moreover, your components can become more maintainable and reusable.
- The mastering of these principles is not the last step to become a perfect developer, actually, it’s just the beginning.
- You will have to deal with different problems in your projects, understand the best approach and, finally, check if you’re breaking some principles.

Reference

- SOLID Principles Applied to Swift :- https://marcosantadev.com/solid-principles-applied-swift/

- Clean Coder (Uncle Bob) :- https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start

- Software Craftsmanship Manifesto :- http://manifesto.softwarecraftsmanship.org