

iOS Design Patterns in Swift

BJ Miller
@bjmillerltd
iOS developer, DXY
March 22, 2016



Head First Design Patterns

O'Reilly Publishing

Agenda

- There are lots of patterns, not covering them all
- What are design patterns?
- Why care?
- Cocoa/CocoaTouch favor some patterns more than others
- Show how the book implements **some** patterns
- Show how I think they're implemented in Swift

What Are Design Patterns?

- A **Pattern** is a **solution** to a **problem** in a **context**
- The **context** is the situation in which the pattern applies
- The **problem** refers to the goal you are trying to achieve in this context, or constraints occurring in the context
- The **solution** is what you're after, a general design to apply which resolves the goal and constraints

Why Care?

- Recognize problems better
- Recognize what tools could solve certain problems
- Help structure your code for better reuse or efficiency
- Help structure your code for better refactorability
- Help avoid making up words like refactorability

MVC

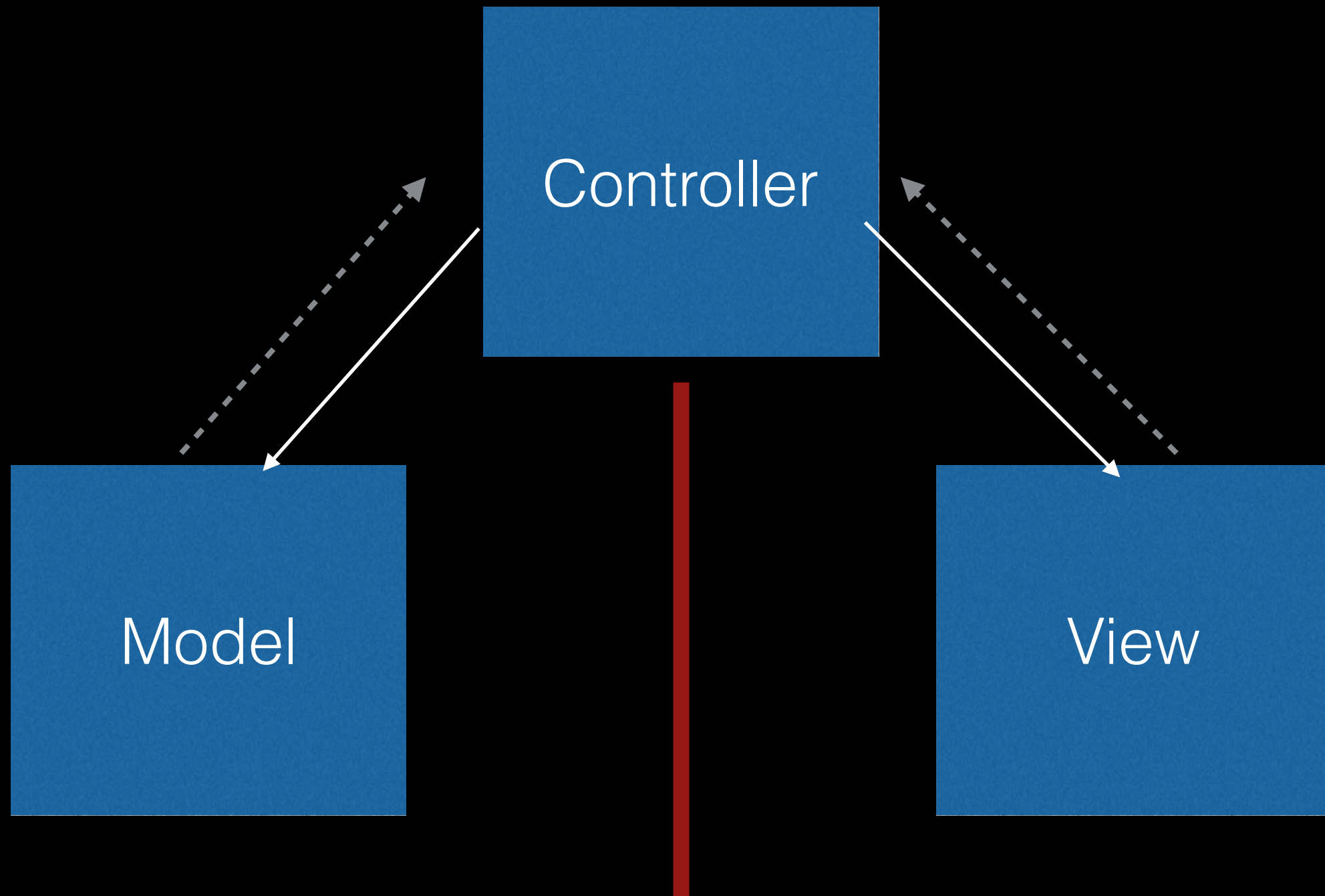
MVC Pattern Defined

“The MVC Pattern is a software architectural pattern... [which] divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.”

MVC Pattern Defined

“The MVC Pattern is a software architectural pattern... [which] divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.”

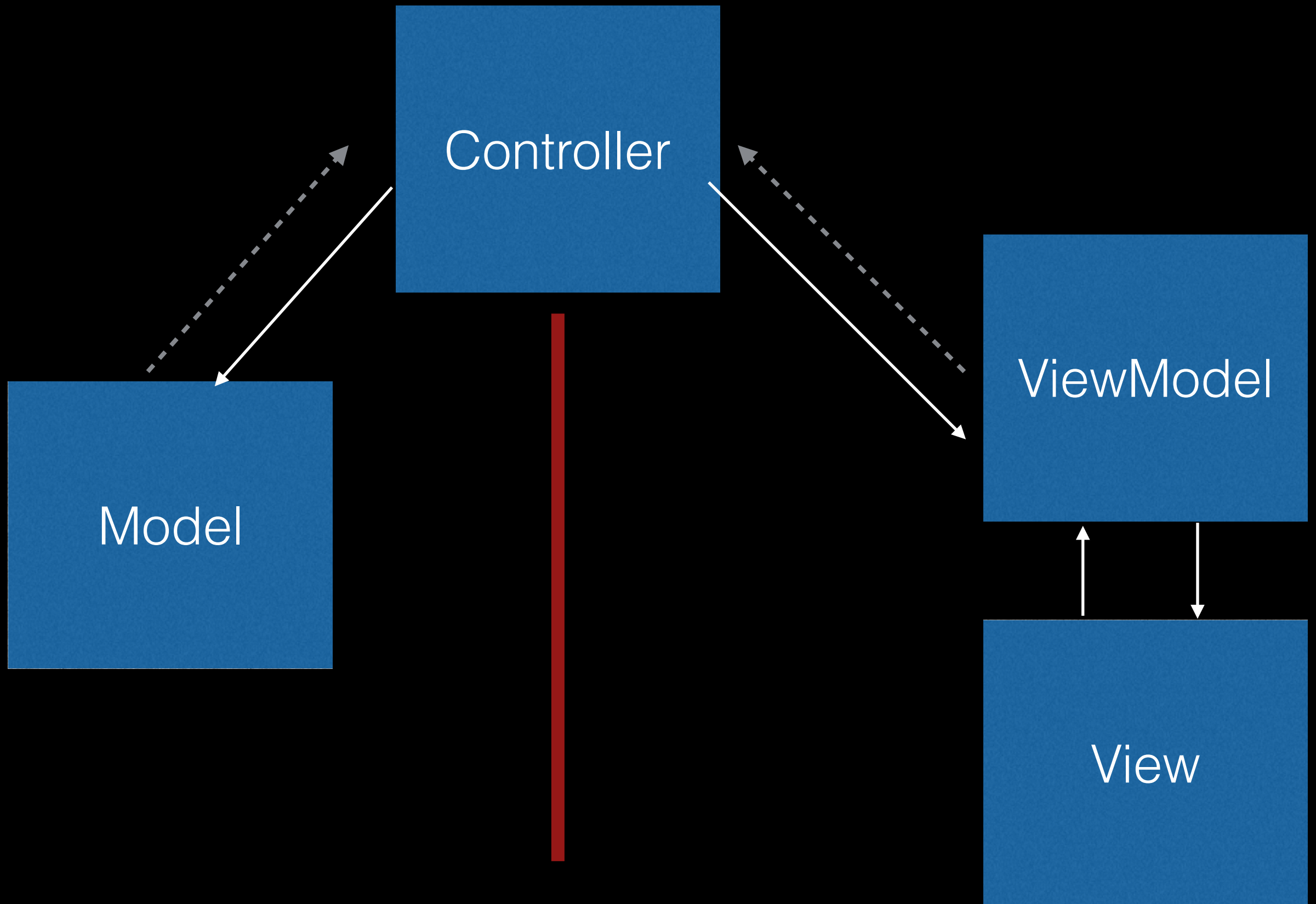
Wat?



MVVM

MVVM Pattern Defined

“The MVVM Pattern is a variation of Martin Fowler’s Presentation Model design pattern...[which] abstracts a view’s state and behavior.”



The Delegate Pattern

The Delegate Pattern

Yep.

Delegate Pattern Defined

“The Delegate Pattern is a pattern where an object, instead of performing one of its stated tasks, delegates that task to an associated helper object.”

The Delegate Pattern in iOS

- You see this a lot. Like, a really lot lot.
- UITableView & UICollectionView
- Bluetooth for discovery and service finding/identifying
- NSURLSessionDataTask
- UIPickerView
- CNContactPickerViewController
- UINosePicker

The Delegate Pattern in iOS

- Nothing too out of the ordinary here
- If you use reference types as delegates, pay attention to memory semantics
 - weak: may be nil during its lifetime (must use ?)
 - unowned: should be alive as long as its owner
 - unowned is preferred when possible, but requires non-optionality and init injection

The Delegate Pattern in iOS

```
protocol ButtonDelegateProtocol : class {  
    func didTapButton(button: Button)  
}  
  
class ButtonDelegate : ButtonDelegateProtocol {  
    func didTapButton(button: Button) {  
        print("You tapped the button labeled \(button.title)")  
    }  
}
```

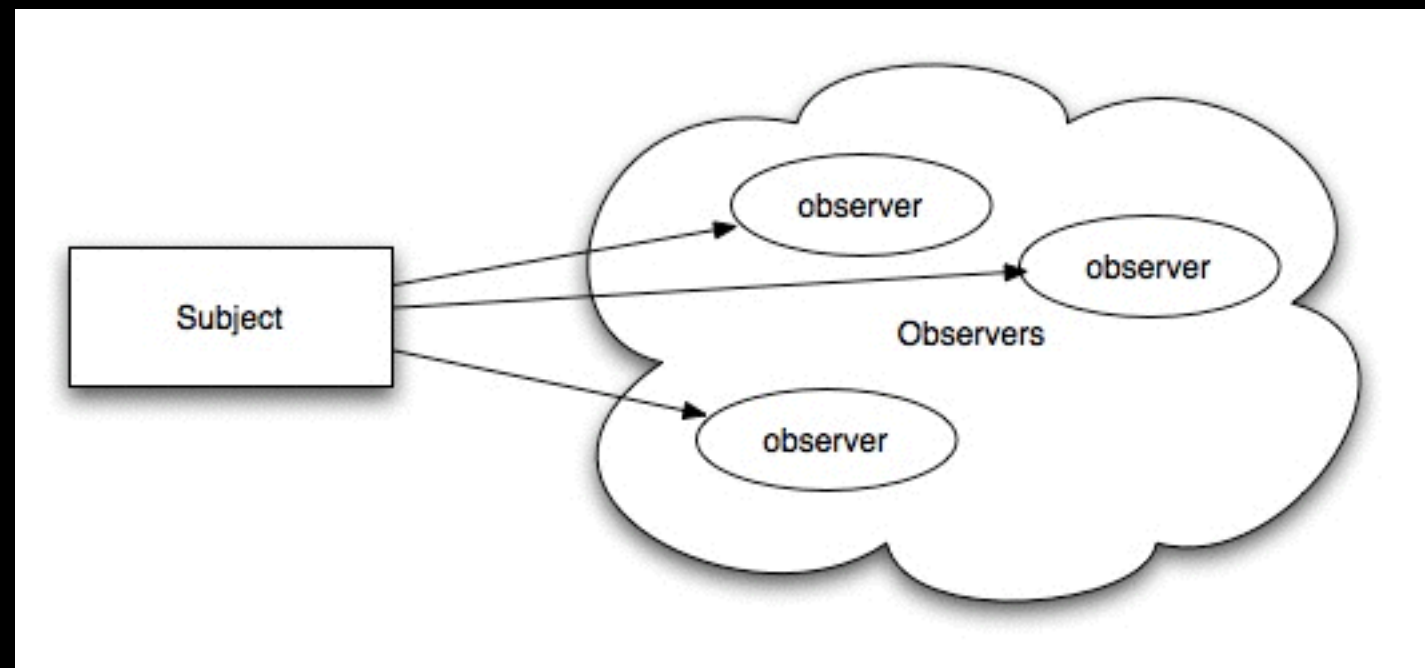
The Delegate Pattern in iOS

```
class Button {  
    let title: String  
    unowned let delegate: ButtonDelegateProtocol  
  
    init(title: String, delegate: ButtonDelegateProtocol) {  
        self.title = title  
        self.delegate = delegate  
    }  
  
    func buttonPressed() { delegate.didTapButton(self) }  
}  
  
let delegate = ButtonDelegate()  
let button = Button(title: "Tap Me!", delegate: delegate)  
button.buttonPressed()  
// You tapped the button labeled Tap Me!
```

The Observer Pattern

Observer Pattern Defined

“The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.”



The Observer Pattern in iOS

- Key-Value Observation (KVO)
 - Observes a property in a class such that you can inspect when the ivar gets accessed
 - NSHipster: “KVO has the *worst* API in all of Cocoa. It’s awkward, verbose, and confusing.”
- NotificationCenter
 - Uses a defaultCenter singleton
 - Basically a key-value store for a notification name and a list of selectors to call
 - Frequent use case: UIKeyboardWill[Show/Hide]Notification

The Observer Pattern in iOS

```
import Foundation

let DataStoreDidUpdateNotification = "DataStoreDidUpdateNotification"

class MainClass {
    func updateDataStore() {
        // just imagine something worthwhile happened here
        NotificationCenter.defaultCenter()
            .postNotificationName(DataStoreDidUpdateNotification,
                                object: nil,
                                userInfo: nil)
    }
}
```

The Observer Pattern in iOS

```
class SomeOtherClass {
    init() {
        NotificationCenter.defaultCenter()
            .addObserverForName(DataStoreDidUpdateNotification,
                                object: nil,
                                queue: NSOperationQueue.mainQueue()) { [unowned self] notification in
                self.dataStoreUpdated(notification)
            }
    }

    func dataStoreUpdated(notification: NSNotification) {
        print("I was notified: \(notification.userInfo)")
    }

    deinit {
        NotificationCenter.defaultCenter().
            removeObserver(self, name: DataStoreDidUpdateNotification, object: nil)
    }
}
```


The Observer Pattern in iOS

```
let someOtherClass = SomeOtherClass()  
let mainClass = MainClass()  
mainClass.updateDataStore()  
  
// I was notified: nil
```

The Singleton Pattern

Singleton Pattern Defined

“The Singleton Pattern ensures a class has only one instance, and provides a global point of access to it.”



The Singleton Pattern in Objective-C

```
@implementation MyClass
+ (instancetype)sharedInstance {
    static MyClass *sharedInstance = nil;
    static dispatch_once_t onceToken;

    dispatch_once(&onceToken, ^{
        sharedInstance = [[MyClass alloc] init];
    });
    return sharedInstance;
}
@end
```

The Singleton Pattern in Swift

```
class MyClass {  
    static let sharedInstance = MyClass()  
    private init() {}  
}
```

- The private `init` is optional, but helps prevent accidental initialization outside of using singleton object
- `sharedInstance` is a static property, not a method like ObjC

Using The Singleton Pattern

```
import Foundation
```

```
let defaults = UserDefaults.standardUserDefaults()  
defaults.setBool(true, forKey: "agreedToEula")  
defaults.synchronize()  
let agreedToEula = defaults.boolForKey("agreedToEula")
```

```
// or with MyClass
```

```
let myClass = MyClass.sharedInstance
```

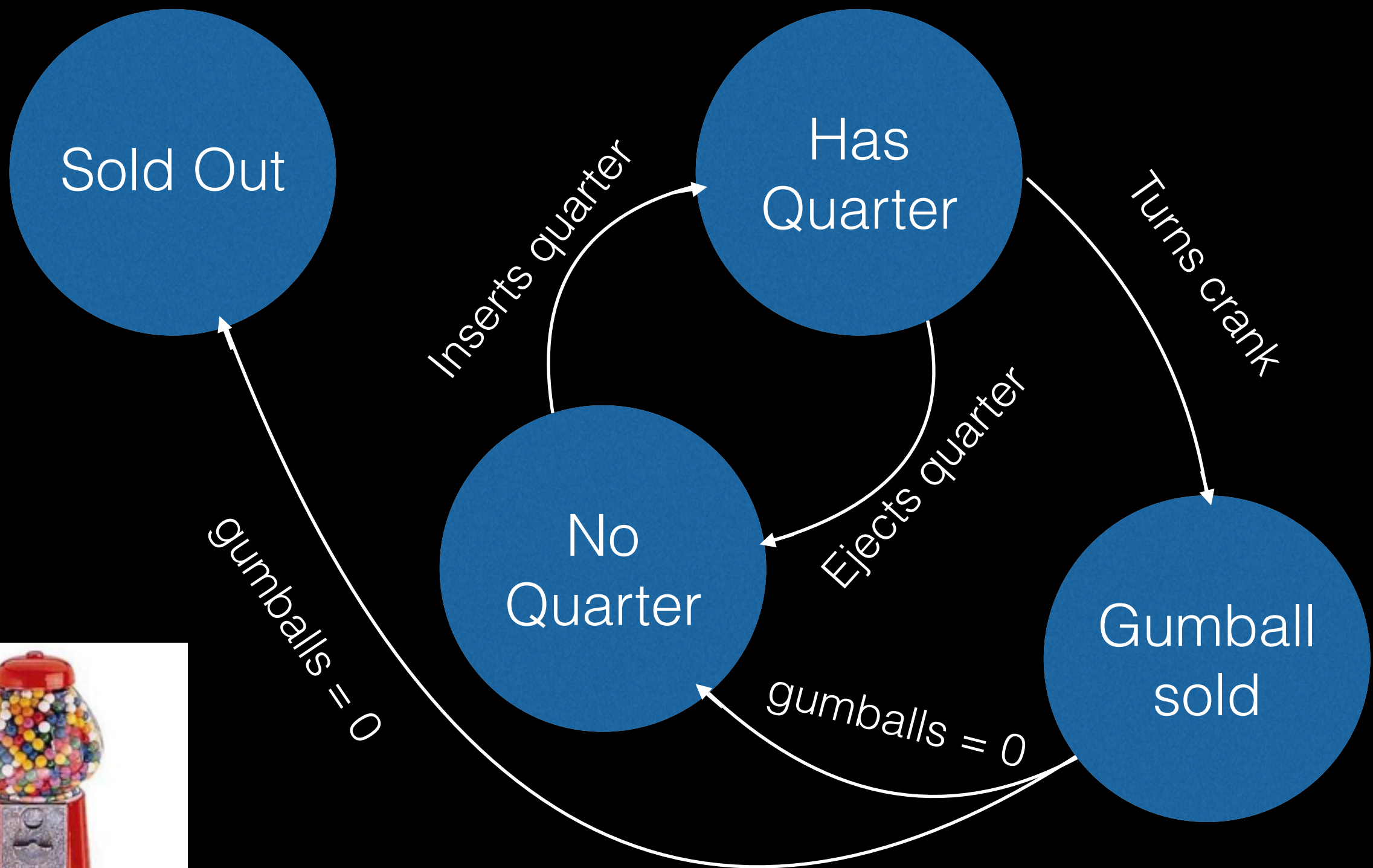
A thin, light gray outline of the state of Ohio is centered on the slide. The outline shows the state's irregular borders, including the Lake Erie islands in the north and the jagged southern border.

The State Pattern

State Pattern Defined

“The State Pattern allows an object to alter its behavior when its internal state changes. the object will appear to change its class.”

The State Pattern



The State Pattern

The Gumball Machine's state changes when different actions happen ("state machine")

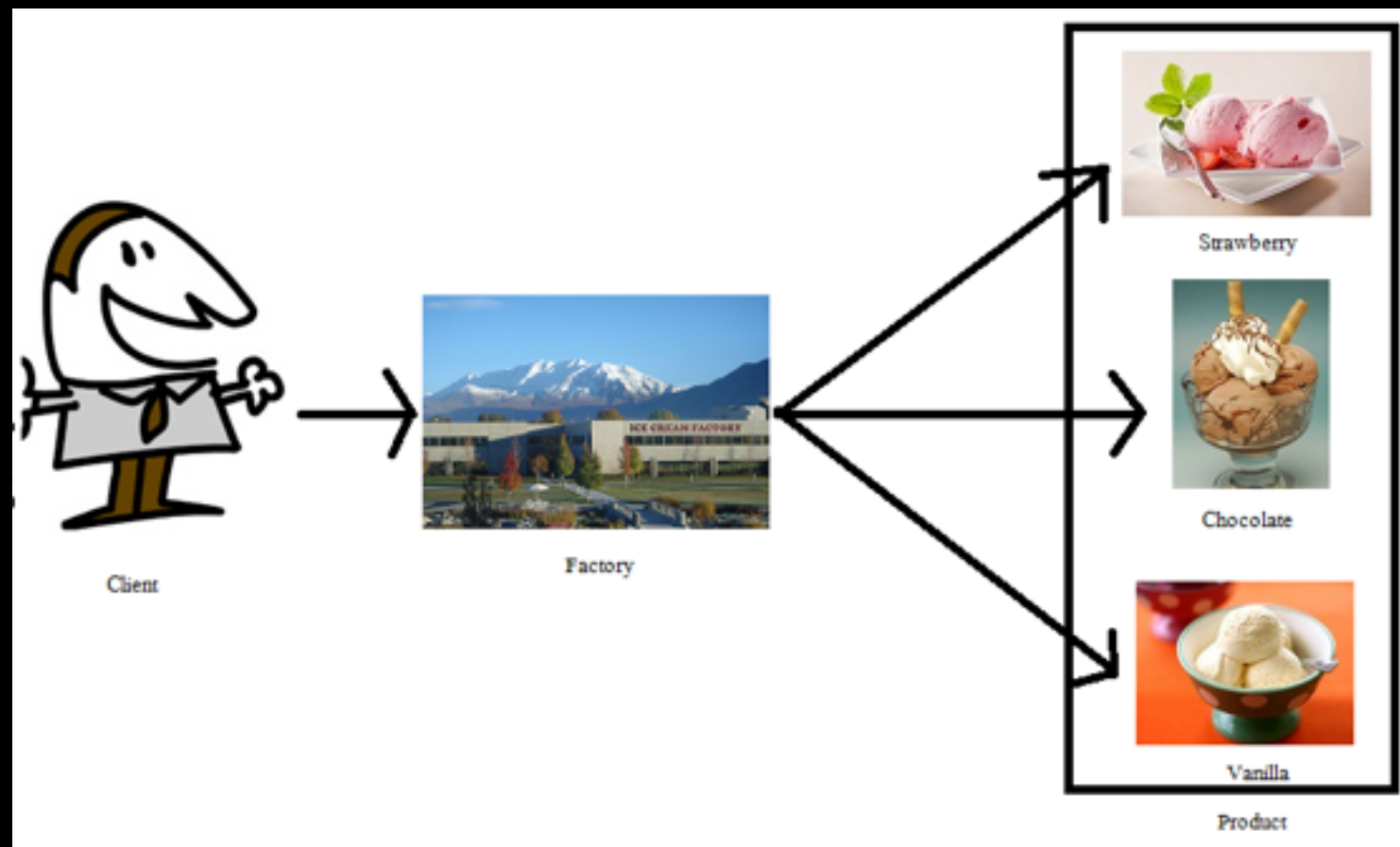
The State Pattern

Demo

The Factory Pattern

Factory Pattern Defined

“The Factory Pattern encapsulates object creation by letting subclasses decide what objects to create.”



The Factory Pattern in iOS

- Useful for things like creating heterogeneous cells in a table view
- Useful for creating objects that have common protocol but different implementations
- Book example: pizza shops
 - Chicago style vs California style
 - But both use dough, ingredients, bake, cut, etc

The Factory Pattern in iOS

```
func tableView(tableView: UITableView,  
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {  
  
    // use factory to create the cell and return it here  
    let cell = cellFactory.configuredCell(  
        cellContents: editableCellContents,  
        inTableView: tableView, atIndexPath: indexPath,  
        sortOrderElement: sortOrderElements[indexPath.section],  
        forEditing: self.isEditing  
    )  
  
    return cell  
}
```

The Factory Pattern in iOS

```
func configuredCell(cellContents cellContents: [String : [ContactModelData]],
    inTableView tableView: UITableView, atIndexPath path: NSIndexPath,
    sortOrderElement element: String, nameAndPhotoInfo: ProfileNameAndPhotoInfo,
    forEditing editing: Bool) -> UITableViewCell {

    guard !editing else {
        // return an editing cell from factory
    }

    let cell = tableView.dequeueReusableCellWithIdentifier(.ContactInfoCell,
        forIndexPath: path) as! ProfileContactInfoTableViewCell

    switch element {
    case kPropertyAddress:
        // address specific stuff
        cell.setContactInfo(someValue, withTitle: someTitle,
            multilineContactInfo: true)
    default:
        cell.setContactInfo(someValue, withTitle: someTitle,
            multilineContactInfo: false)
    }
    return cell
}
```


The Builder Pattern

Builder Pattern Defined

“The Builder Pattern encapsulates the construction of a product to allow it to be constructed in steps.”



Builder Pattern Defined

- Beneficial for building many subparts of an aggregate object
- Abstracts object creation into a single object
- Utilize protocol extensions with default behavior for overarching parser
- Pass a function to overarching parser to tell it how to parse a single object
- Aggregate is returned to you

Builder Pattern in iOS

Demo

The Adapter Pattern

Adapter Pattern Defined

“The Adapter Pattern converts the interface of a class into another interface the clients expect.”



Adapter Pattern Defined

- Allows your application or modules to interface with a standard API
- Adapter API can interchange its underlying API
- Changing Adapter API should not affect your app or module's code
- Helpful for 3rd party code that may change or be replaced

Adapter Pattern in Swift

For example, adapting the Lockbox framework:

```
let LoginTokenID = "LoginTokenID"
struct TokenManager {
    static var loginToken: String? {
        return Lockbox.stringForKey(LoginTokenID)
    }
    static func setLoginToken(token: String) {
        Lockbox.setString(token, forKey: LoginTokenID)
    }
    static func removeLoginToken() {
        Lockbox.setString(nil, forKey: LoginTokenID)
    }
    static func clearAll() { self.removeLoginToken() }
}
```



The Template Pattern

Template Pattern Defined

“The Template Pattern defines a skeleton of an algorithm in a method, called template method, which defers some steps to subclasses. It lets one redefine certain steps of an algorithm without changing the algorithm’s structure.”

Hipster Ipsum

Artisanal filler text for your site or project.



Do you need some text for your website or whatever? *sigh* Okay...

Paragraphs:

Type: ☒ Hipster w/ a shot of Latin.
☐ Hipster, neat.

Template Pattern Defined

- Use superclass/subclass relationship
- Or...
- Use protocol with extension for default behavior
- Protocol extension calls other methods defined in protocol since they have a definition and type
- Protocol methods must be implemented by your conforming type

Template Pattern in Swift

```
protocol AnimalType {  
    func eat()  
    func poop()  
    func liveLife()  
}
```

```
extension AnimalType {  
    func liveLife() {  
        eat()  
        poop()  
    }  
}
```

Template Pattern in Swift

```
struct Dog : AnimalType {  
    func eat() { print("Dog eating dog food...nom nom") }  
    func poop() { print("Dog pooping") }  
}
```

```
Dog().liveLife()  
// "Dog eating dog food...nom nom"  
// "Dog pooping"
```

```
struct Deer : AnimalType {  
    func eat() { print("Deer eating plants, yum!") }  
    func poop() { print("Deer poop everywhere!") }  
}
```

```
Deer().liveLife()  
// "Deer eating plants, yum!"  
// "Deer poop everywhere!"
```

The Iterator Pattern

Iterator Pattern Defined

"The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation."



Sebastian Thul, diplomat and iterator of the 591st expedition during the Great Crusade

Iterator Pattern in Swift

- Conform to `GeneratorType` protocol and provide `next()` method in your type
- Or, conform to `SequenceType`, includes benefits
- A **sequence** (of type `SequenceType`) represents a series of values in a sequence
- A **generator** (of type `GeneratorType`) provides a way to use the values in a sequence, one at a time, sequentially

How To Use A Generator

```
let nums = [1,2,3]
var generator = nums.generate()
while let num = generator.next() {
    print(num)
}
```

```
// prints
```

```
1
```

```
2
```

```
3
```

How To Create A Generator

```
struct Mailman<T : Comparable> {  
    private var children: [T]  
    init<S : SequenceType where S.Generator.Element == T>(_ sequence: S) {  
        children = sequence.sort()  
    }  
  
    func indexOf(element: T) -> Int? { return children.indexOf(element) }  
  
    private func insertionIndexForElement(element: T) -> Int { // returns index }  
  
    mutating func insert(element: T) {  
        let index = insertionIndexForElement(element)  
        children.insert(element, atIndex: index)  
    }  
  
    mutating func remove(element: T) -> T? {  
        let index = indexOf(element)  
        return index != nil ? children.removeAtIndex(index!) : nil }  
}
```

How To Create A Generator

```
extension Mailman : SequenceType {  
    typealias Generator = AnyGenerator<T>  
  
    func generate() -> AnyGenerator<T> {  
        var index = 0  
        return anyGenerator {  
            guard index < self.children.count else { return nil }  
            let child = self.children[index]  
            index += 1  
            return child  
        }  
    }  
}
```

How To Create A Generator

```
var tennesseeMailman = Mailman(["cletus", "mary jane", "buford"])
tennesseeMailman.children
tennesseeMailman.insert("shuga")
tennesseeMailman.children

var tennesseeMailmanGenerator = tennesseeMailman.generate()
while let child = tennesseeMailmanGenerator.next() {
    print(child)
}
```

SequenceType Benefits

```
let uppercaseChildren = tennesseeMailman.map { $0.uppercaseString }  
uppercaseChildren  
// ["BUFORD", "CLETUS", "MARY JANE", "SHUGA"]
```

```
let joinedUcaseChildren = tennesseeMailman.map { $0.uppercaseString }  
                                     .joinWithSeparator("!")  
joinedUcaseChildren  
// "BUFORD!CLETUS!MARY JANE!SHUGA"
```

```
let lazyChildren = tennesseeMailman.lazy.map { $0.uppercaseString }  
lazyChildren  
// LazyMapSequence<Mailman<String>, String>  
lazyChildren.maxElement()  
// "SHUGA"
```

Wrapping Up

- Knowing Design Patterns helps identify problems much easier
- Allows for better tools to solve problems
- Enables better structured code

Thank you

- @bjmillerltd on Twitter
- Concepts in Code podcast
- Sams Teach Yourself Swift in 24 Hours (<http://informit.com/bjmiller>)

