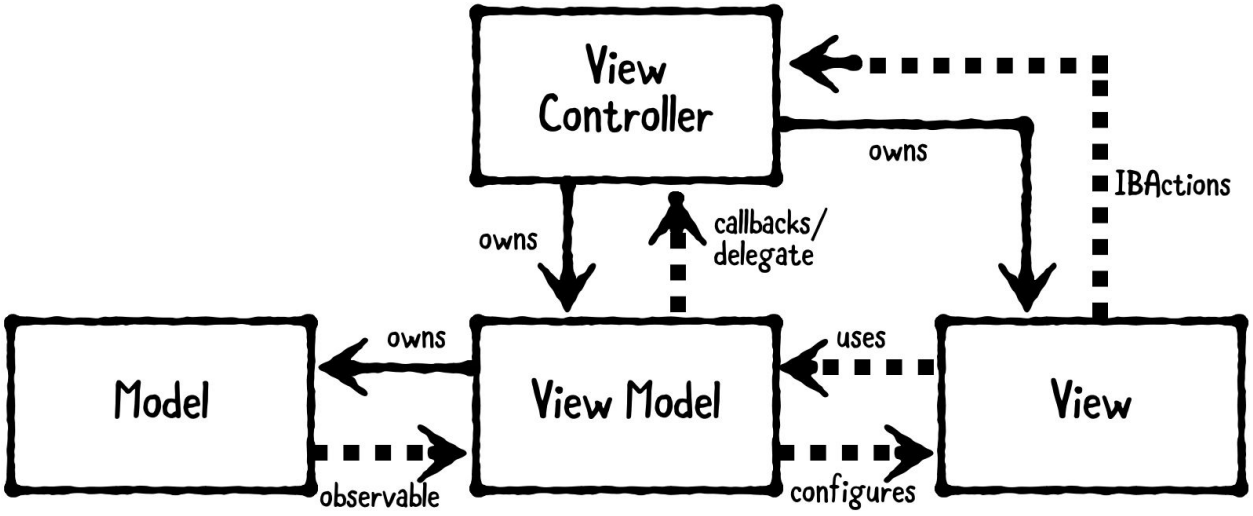
Intermediate Design Patterns

Patterns are used less frequently than the fundamental design patterns

Model-View-ViewModel Pattern
Factory Pattern
Adapter Pattern
Iterator Pattern
Prototype Pattern
State Pattern
Multicast Delegate Pattern
Facade Pattern

Model-View-ViewModel (MVVM) [Structural pattern]



Models hold app data. They're usually structs or simple classes
Views display visual elements and controls on the screen. They're typically subclasses of UIView.
View models transform model information into values that can be displayed on a view. They're usually classes, so they can be passed around as references.

view controllers do exist in MVVM, but their role is minimized.

When should you use it?
Use this pattern when you need to transform models into another representation for a view.

For example, you can use a view model to transform a Date into a date-formatted String, a Decimal into a currency-formatted String, or many other useful transformations.

This pattern compliments MVC especially well. Without view models, you'd likely put model-to-view transformation code in your view controller. However, view controllers are already doing quite a bit: handling viewDidLoad and other view lifecycle events, handling view callbacks via IBActions and several other tasks as well.
This leads to what developers jokingly refer to as "MVC: Massive View Controller".

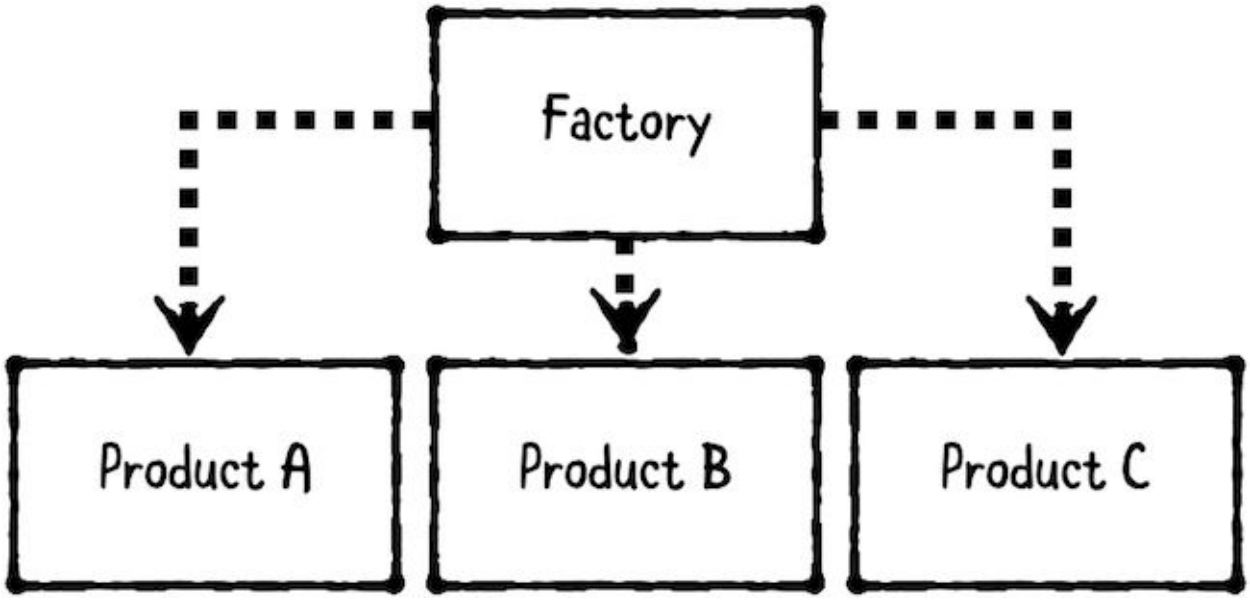
MVVM is a great way to slim down massive view controllers that require several model-to-view transformations.

What should you be careful about?
MVVM works well if your app requires many model-to-view transformations. However, not every object will neatly fit into the categories of model, view or view-model. Instead, you should use MVVM in combination with other design patterns.

Furthermore, MVVM may not be very useful when you first create your application. MVC may be a better starting point. As your app's requirements change, you'll likely need to choose different design patterns based on your changing requirements. It's okay to introduce MVVM later in an app's lifetime when you really need it.

Factory pattern [Creational Pattern]

The factory pattern is a creational pattern that provides a way to make objects without exposing creation logic



The factory creates objects. (isolate object creation logic within its own construct.)
The **products** are the objects that are created.

Technically, there are multiple “flavors” of this pattern, including simple factory, abstract factory and others. However, each of these share a common goal: to isolate object creation logic within its own construct.

When should you use it?
Use the factory pattern whenever you want to separate out product creation logic, instead of having consumers create products directly.

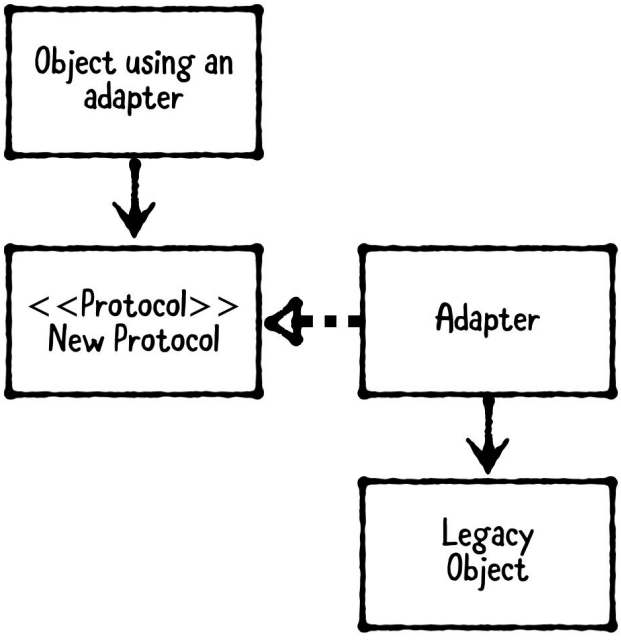
A factory is very useful when you have a group of related products, such as polymorphic subclasses or several objects that implement the same protocol.
For example, you can use a factory to inspect a network response and turn it into a concrete model subtype.

What should you be careful about?
Not all polymorphic objects require a factory. If your objects are very simple, you can always put the creation logic directly in the consumer, such as a view controller itself.

Alternatively, if your object requires a series of steps to build it, you may be better off using the builder pattern or another pattern instead.

Adapter pattern [Behavioral pattern]

The adapter pattern is a behavioral pattern that allows incompatible types to work together.



An **object using an adapter** is the object that depends on the new protocol.
The **new protocol** is the desired protocol for use.
A **legacy object** existed before the protocol was made and cannot be modified directly to conform to it.
An **adapter** is created to conform to the protocol and passes calls onto the legacy object.

The **adapter** pattern is useful when working with classes from third-party libraries that cannot be modified. You can use **protocols** to have them work with the project's custom classes.

To use an **adapter** , you can either extend the legacy object , or make a new adapter class.

When should you use it?
Classes, modules, and functions can't always be modified, especially if they're from a third-party library. Sometimes you have to adapt instead!

For this example, you'll adapt a third-party authentication service to work with an app's internal authentication protocol

```
// MARK: - Legacy Object
public class GoogleAuthenticator {
      public func login(email: String, password: String,
      completion: @escaping (GoogleUser?, Error?) -> Void) {
            // Make networking calls that return a token string
            let token = "special-token-value"
            let user = GoogleUser(email: email, password: password,
                                  token: token)
            completion(user, nil)
      }
}

public struct GoogleUser {
      public var email: String
      public var password: String
      public var token: String
}
```

Imagine GoogleAuthenticator is a third-party class that cannot be modified. Thereby, it is the legacy object .

```
// MARK: - New Protocol
public protocol AuthenticationService {
      func login(email: String, password: String,
            success: @escaping (User, Token) -> Void, failure: @escaping
            (Error?) -> Void)
}
```

```
public struct User {
	public let email: String
	public let password: String
}

public struct Token {
	public let value: String
}
```

The app will use this protocol instead of GoogleAuthenticator directly, and it gains many benefits by doing so. For example, you can easily support multiple authentication mechanisms – Google, Facebook and others – simply by having them all conform to the same protocol.

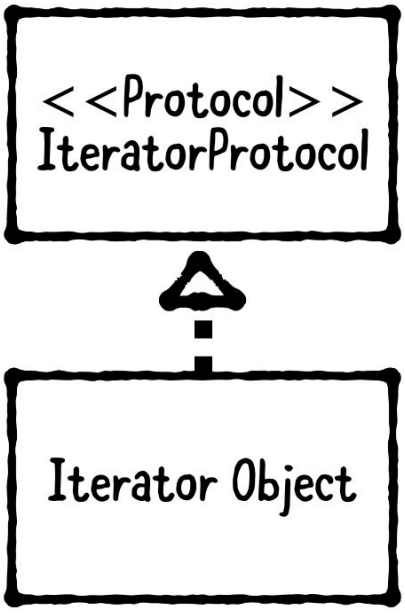
While you could extend GoogleAuthenticator to make it conform to AuthenticationService — which is also a form of the adapter pattern! — you can also create an Adapter class

```
// MARK: - Adapter
//GoogleAuthenticationAdapter as the adapter between GoogleAuthenticator and
AuthenticationService.
public class GoogleAuthenticatorAdapter: AuthenticationService {
	//You declare a private reference to GoogleAuthenticator,
	so it's hidden from end consumers.
	private var authenticator = GoogleAuthenticator()
	//
	public func login(email: String,
				password: String,
				success: @escaping (User, Token) -> Void,
				failure: @escaping (Error?) -> Void) {
		authenticator.login(email: email, password: password) {
			(googleUser, error) in
			//
			guard let googleUser = googleUser else {
				failure(error)
				Return
			}
			//
			let user = User(email: googleUser.email,
						password: googleUser.password)
			let token = Token(value: googleUser.token)
			success(user, token)
		}
	}
}

// MARK: - Object Using an Adapter
public class LoginViewController: UIViewController {
```

```
        // MARK: - Properties
        public var authService: AuthenticationService!
        // MARK: - Views
        var emailTextField = UITextField()
        var passwordTextField = UITextField()

        // MARK: - Class Constructors
        public class func instance( with authService: AuthenticationService)
        -> LoginViewController {
              let viewController = LoginViewController()
              viewController.authService = authService
              return viewController
        }


        public func login() {
              guard let email = emailTextField.text,
              let password = passwordTextField.text else {
                    print("Email and password are required inputs!")
                    Return
              }
              authService.login(email: email, password: password,
              success: { user, token in
                    print("Auth succeeded: \(user.email), \(token.value)")
              },
              failure: { error in
                    print("Auth failed with error: no error provided")
              })
        }
}

// MARK: - Example
let viewController = LoginViewController.instance(
                        with: GoogleAuthenticatorAdapter())
viewController.emailTextField.text = "user@example.com"
viewController.passwordTextField.text = "password"
viewController.login()
```

Iterator pattern [Behavioral pattern]

The iterator pattern is a behavioral pattern that provides a standard way to loop through a collection.



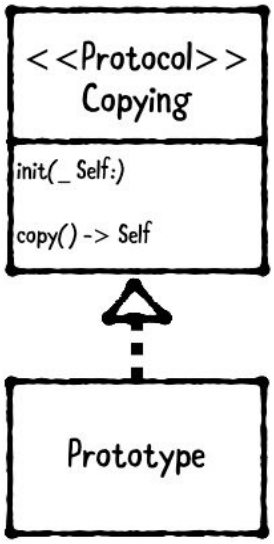
The **Swift IteratorProtocol** defines a type that can be iterated using a for in loop.

The **iterator object** is the type you want to make iterable. Instead of conforming to `IteratorProtocol` directly, however, you can conform to `Sequence` , which itself conforms to `IteratorProtocol` . By doing so, you'll get many higher-order functions, including `map` , `filter` and more, for free.

When should you use it?
Use the iterator pattern when you have a type that holds onto a group of objects, and you want to make them iterable using a standard for in syntax.

Prototype pattern [Creational pattern]

The prototype pattern is a creational pattern that allows an object to copy itself.



A **copying** protocol that declares copy methods.
A **prototype** class that conforms to the copying protocol.

There are actually two different types of copies: **shallow** and **deep** .
A shallow copy creates a new object instance, but doesn't copy its properties. Any properties that are reference types still point to the same original objects.

A deep copy creates a new object instance and duplicates each property as well.

When should you use it?
Use this pattern to enable an object to copy itself.

For example, Foundation defines the NSCopying protocol. However, this protocol was designed for Objective-C, and unfortunately, it doesn't work that well in Swift. You can still use it, but you'll wind up writing more boilerplate code yourself. Instead, you'll implement your own Copying protocol

Copying protocol

```
public protocol Copying: class {
	//This is called a copy initializer as its purpose is to create a new class instance using an existing instance.
	init(_ prototype: Self)
}

extension Copying {
	//  You normally won't call the copy initializer directly. Instead, you'll simply call copy() on a conforming Copying class instance that you want to copy.
	public func copy() -> Self {
		return type(of: self).init(self)
	}
}
```

Prototype

```
//conforms to Copying
public class Monster: Copying {

	public var health: Int
	public var level: Int

	public init(health: Int, level: Int) {
		self.health = health
		self.level = level
	}
```

```
        //In order to satisfy Copying, you must declare init(_ prototype:)
        as required.
        public required convenience init(_ monster: Monster) {
              self.init(health: monster.health, level: monster.level)
        }
}
```

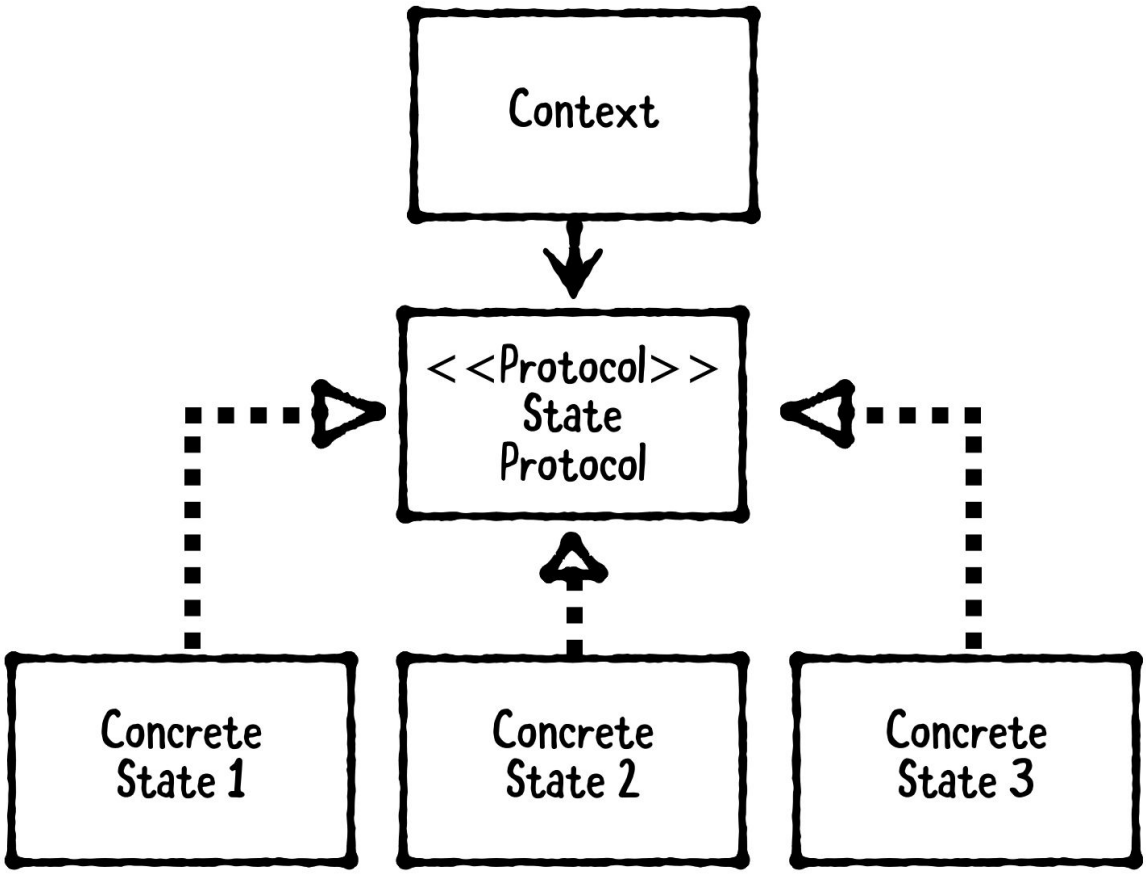
Example:

```
let monster = Monster(health: 700, level: 37)
let monster2 = monster.copy()
print("Watch out! That monster's level is \(monster2.level)!")
```

Output:
Watch out! That monster's level is 37!

State Pattern [Behavioral pattern]

The state pattern is a behavioral pattern that allows an object to change its behavior at runtime. It does so by changing its current state. Here, “state” means the set of data that describes how a given object should behave at a given time.



The **context** is the object that has a current state and whose behavior changes.

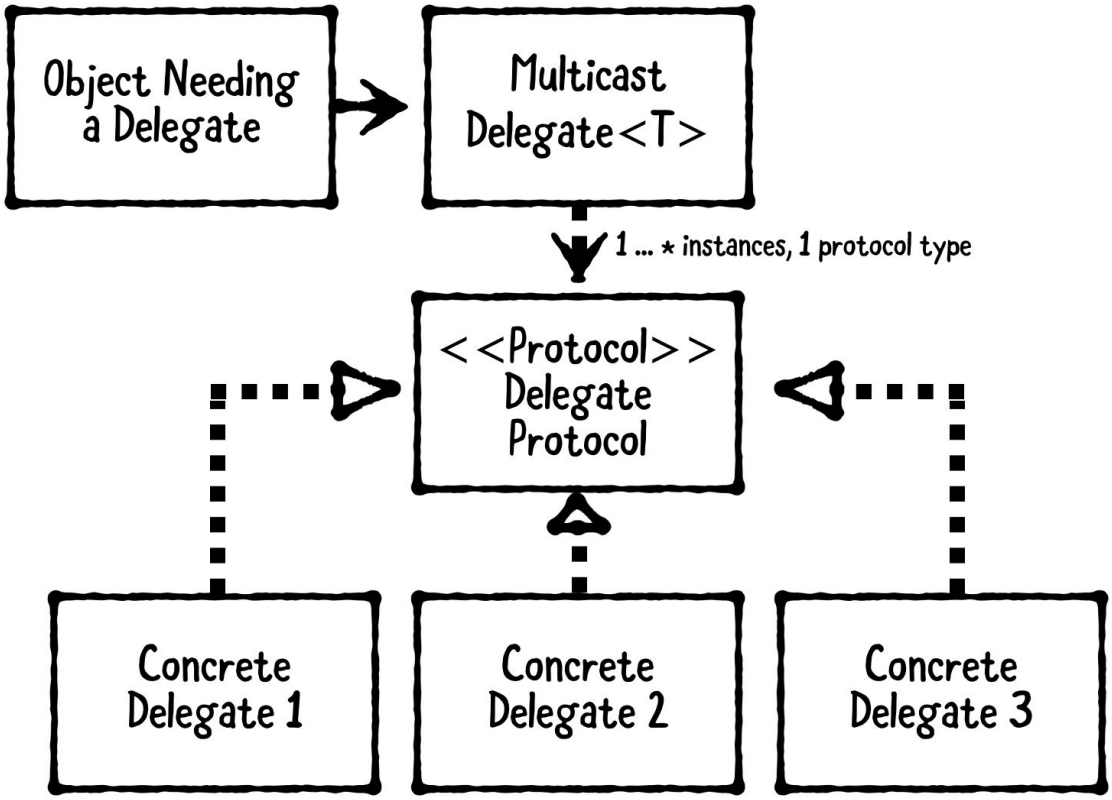
The **state protocol** defines required methods and properties.
Developers commonly substitute a base state class in place of a protocol. By doing so, they can define stored properties in the base, which isn't possible using a protocol.
Even if a base class is used, it's not intended to be instantiated directly. Rather, it's defined for the sole purpose of being subclassed. In other languages, this would be an abstract class .

Concrete states conform to the state protocol, or if a base class is used instead, they subclass the base.
The context holds onto its current state, but it doesn't know its concrete state type. Instead, the context changes behavior using polymorphism: concrete states define how the context should act. If you ever need a new behavior, you define a new concrete state.

The state pattern doesn't actually tell you where to put transition logic between states. Rather, this is left for you to decide: you can put this logic either within the context or within the concrete states.

Multicast Delegate Pattern [Behavioral pattern]

The multicast delegate pattern is a behavioral pattern that's a variation on the delegate pattern. It allows you to create one-to-many delegate relationships, instead of one-to-one relationships in a simple delegate.



An **object needing a delegate** , also known as the **delegating object** , is the object that has one or more delegates.

The **delegate protocol** defines the methods a delegate may or should implement.

The **delegate(s)** are objects that implement the delegate protocol.

The **multicast delegate** is a helper class that holds onto delegates and allows you to notify each whenever a delegate-worthy event happens.

The main difference between the multicast delegate pattern and the delegate pattern is the presence of a multicast delegate helper class.

When should you use it?
Use this pattern to create one-to-many delegate relationships.
For example, you can use this pattern to inform multiple objects whenever a change has happened to another object. Each delegate can then update its own state or perform relevant actions in response.

```
//Generic class
public class MulticastDelegate<ProtocolType> {

	private class DelegateWrapper {

		weak var delegate: AnyObject?

		init(_ delegate: AnyObject) {
			self.delegate = delegate
		}
	}

	private var delegateWrappers: [DelegateWrapper]

	public var delegates: [ProtocolType] {
		delegateWrappers = delegateWrappers.filter { $0.delegate != nil }
		return delegateWrappers.map { $0.delegate! } as! [ProtocolType]
	}

	public init(delegates: [ProtocolType] = []) {
		delegateWrappers = delegates.map {
			DelegateWrapper($0 as AnyObject)
		}
	}
```

```
	// MARK: - Delegate Management
	public func addDelegate(_ delegate: ProtocolType) {
		let wrapper = DelegateWrapper(delegate as AnyObject)
		delegateWrappers.append(wrapper)
	}

	public func removeDelegate(_ delegate: ProtocolType) {
		guard let index = delegateWrappers.firstIndex(where: {
		$0.delegate === (delegate as AnyObject)
		}) else {
			Return
		}
		delegateWrappers.remove(at: index)
	}

	public func invokeDelegates(_ closure: (ProtocolType) -> ()) {
		delegates.forEach { closure($0) }
	}
}

// MARK: - Delegate Protocol
public protocol EmergencyResponding {
	func notifyFire(at location: String)
	func notifyCarCrash(at location: String)
}

// MARK: - Delegates
public class FireStation: EmergencyResponding {
	public func notifyFire(at location: String) {
		print("Firefighters were notified about a fire at " + location)
	}

	public func notifyCarCrash(at location: String) {
		print("Firefighters were notified about a car crash at " +
		location)
	}
}

public class PoliceStation: EmergencyResponding {
	public func notifyFire(at location: String) {
		print("Police were notified about a fire at " + location)
	}
	public func notifyCarCrash(at location: String) {
		print("Police were notified about a car crash at " + location)
	}
}
```

```
// MARK: - Delegating Object
public class DispatchSystem {
      let multicastDelegate = MulticastDelegate<EmergencyResponding>()
}

// MARK: - Example
let dispatch = DispatchSystem()
var policeStation: PoliceStation! = PoliceStation()
var fireStation: FireStation! = FireStation()
dispatch.multicastDelegate.addDelegate(policeStation)
dispatch.multicastDelegate.addDelegate(fireStation)

dispatch.multicastDelegate.invokeDelegates {
      $0.notifyFire(at: "Ray's house!")
}
```

Police were notified about a fire at Ray's house!
Firefighters were notified about a fire at Ray's house!

```
fireStation = nil
dispatch.multicastDelegate.invokeDelegates {
      $0.notifyCarCrash(at: "Ray's garage!")
}
```

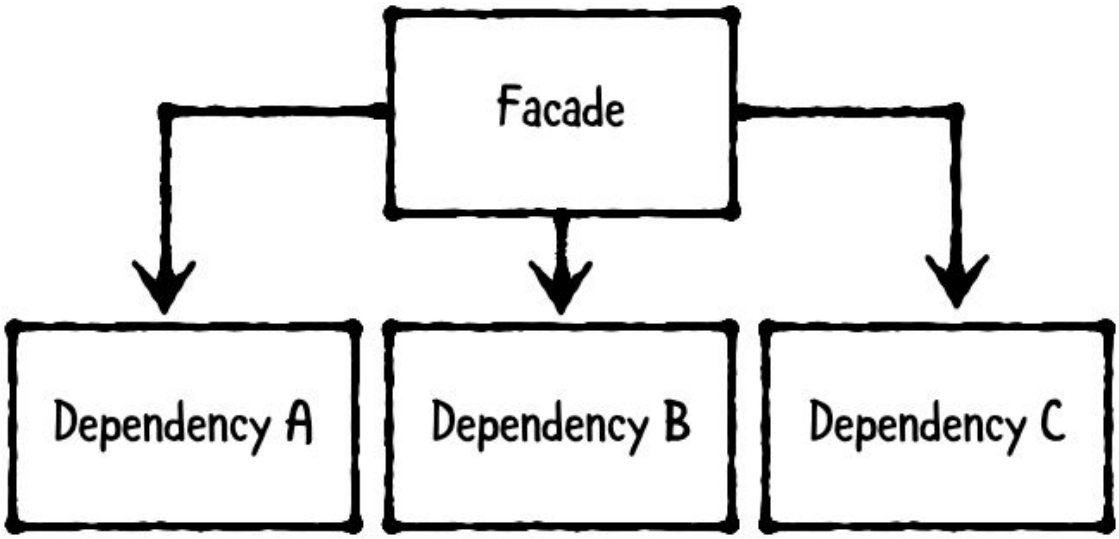
Police were notified about a car crash at Ray's garage!

What should you be careful about?
This pattern works best for “information only” delegate calls.
If delegates need to provide data (Datasource), this pattern doesn’t work well. That’s because multiple delegates would be asked to provide the data, which could result in duplicated information or wasted processing.

Facade Pattern [Structural Pattern]

The facade pattern is a structural pattern that provides a simple interface to a complex system.

The facade provides simple methods to interact with the system. Behind the scenes, it owns and interacts with its dependencies , each of which performs a small part of a complex task.



The **facade** provides simple methods to interact with the system. This allows consumers to use the facade instead of knowing about and interacting with multiple classes in the system.

The dependencies are objects owned by the facade. Each dependency performs a small part of a complex task.

When should you use it?
Use this pattern whenever you have a system made up of multiple components and want to provide a simple way for users to perform complex tasks.

What should you be careful about?
Be careful about creating a “god” facade that knows about every class in your app. It’s okay to create more than one facade for different use cases. For example, if you notice a facade has functionality that some classes use and other functionality that other classes use, consider splitting it into two or more facades.