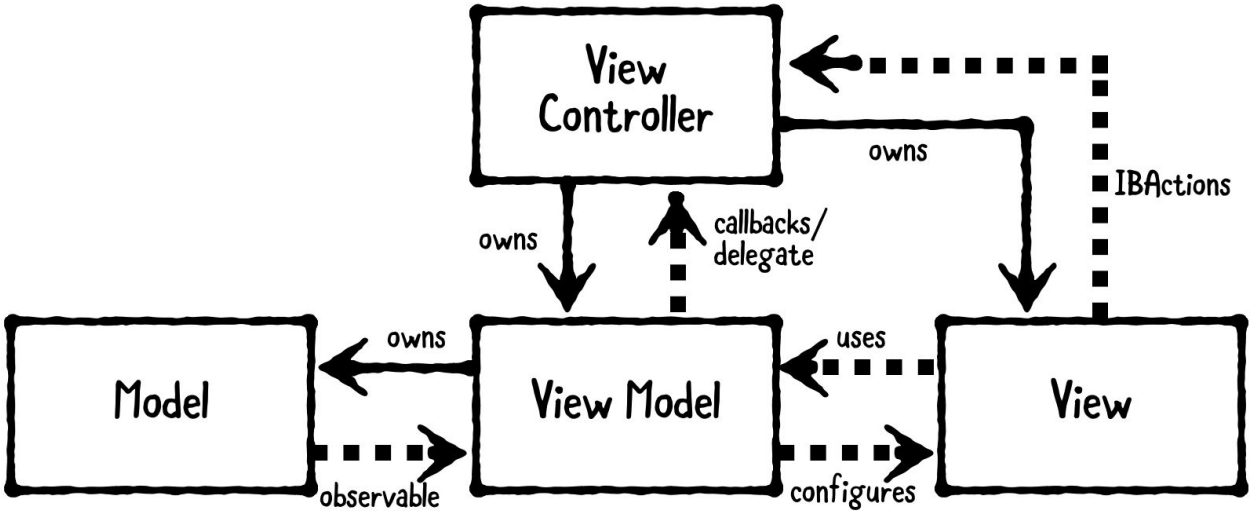
Intermediate Design Patterns

Patterns are frequently used throughout iOS development

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".

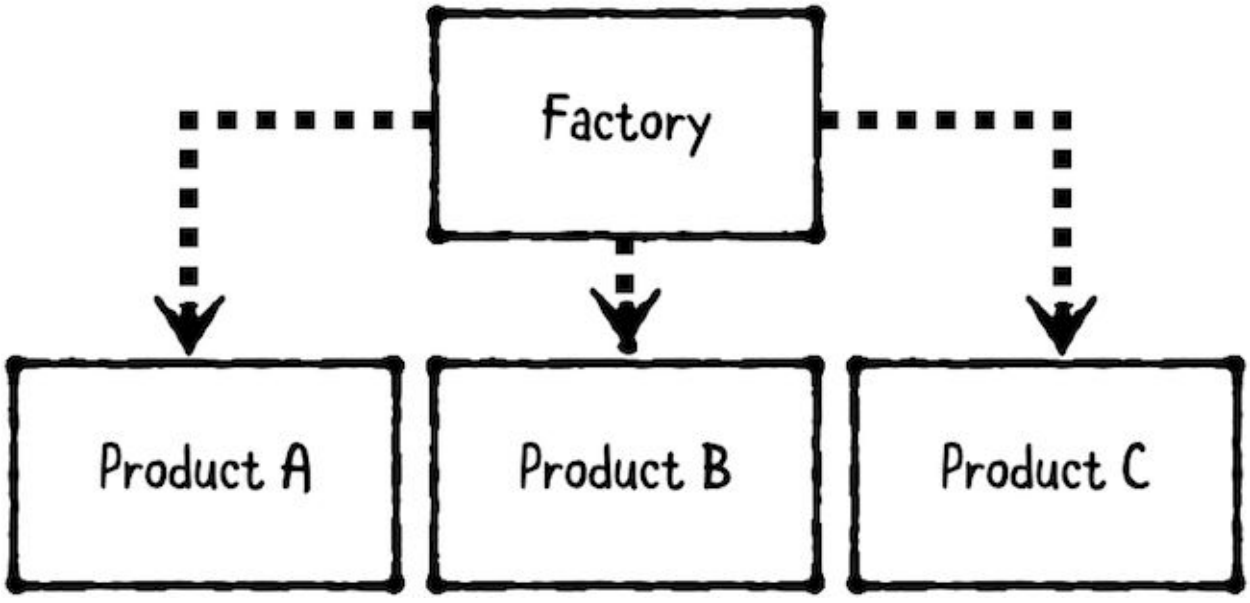
MVVM is a great way to slim down massive view controllers that require several model-to-view transformations.

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.

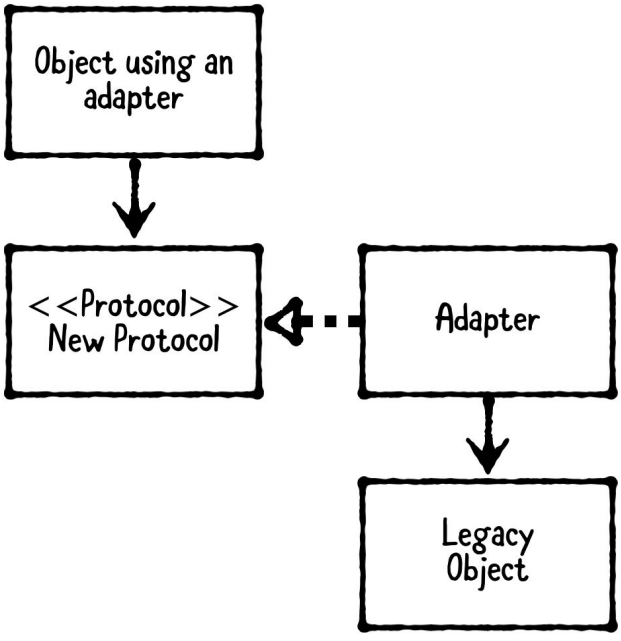
A factory is very useful when you have a group of related products, such as polymorphic subclasses or several objects that implement the same protocol.
For example, you can use a factory to inspect a network response and turn it into a concrete model subtype.

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.

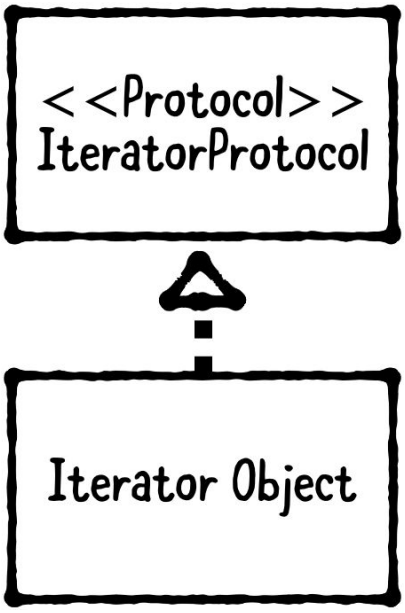
While you could extend GoogleAuthenticator to make it conform to AuthenticationService — which is also a form of the adapter pattern! — you can also create an Adapter class

```
// 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.