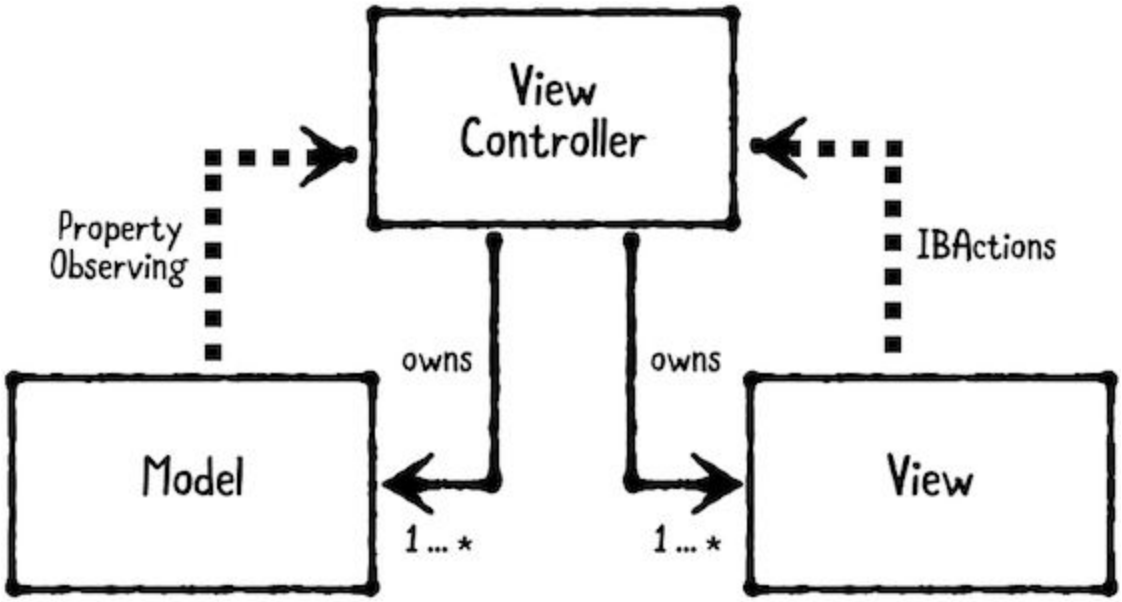
Fundamental Design Patterns

Patterns are frequently used throughout iOS development

Model-View-Controller Pattern
Delegation Pattern
Strategy Pattern
Singleton Pattern
Memento Pattern
Observer Pattern
Builder Pattern

Model-view-controller (MVC) [Structural pattern]

MVC pattern separates objects into three distinct types. are models, views and controllers!



Models hold application data. They are usually structs or simple classes.

Views display visual elements and controls on screen. They are usually subclasses of UIView.

Controllers coordinate between models and views. They are usually subclasses of UIViewController.

MVC is very common in iOS programming because it's the design pattern that Apple chose to adopt in UIKit.

Controllers are allowed to have strong properties for their model and view so they can be accessed directly. Controllers may also have more than one model and/or View.

Conversely, models and views **should not hold a strong** reference to their owning controller. This would cause a retain cycle.

Instead, models communicate to their controller via property observing (KVO), and views communicate to their controller via IBActions.

This lets you reuse models and views between several controllers

Note: Views may have a weak reference to their owning controller through a delegate (see “Delegation Pattern”).

Limitations of MVC
Controllers are much harder to reuse since their logic is often very specific to whatever task they are doing. Consequently, MVC doesn’t try to reuse them.

MVC is a good starting point, but it has limitations. Not every object will neatly fit into the category of model, view or controller. Consequently, applications that only use MVC tend to have a lot of logic in the controllers. This can result in view controllers getting very big! There's a rather quaint term for when this happens, called "**Massive View Controller**."

To solve this issue, you should introduce other design patterns as your app requires them.

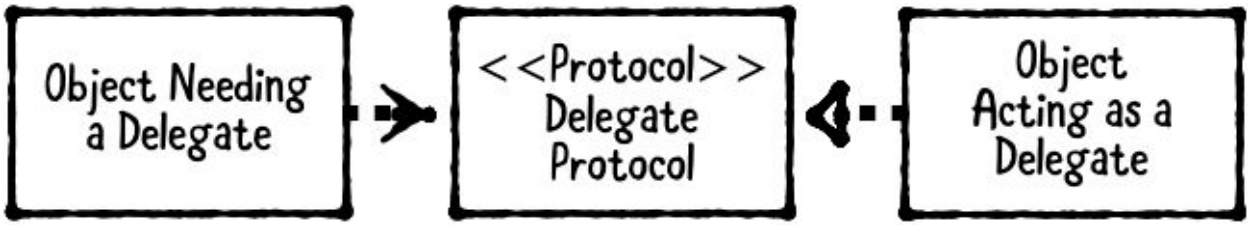
When should you use it?
Use this pattern as a starting point for creating iOS apps. In nearly every app, you’ll likely need additional patterns besides MVC, but it’s okay to introduce more patterns as your app requires them.

Tutorial project

- Models we have: Question and QuestionGroup. View we have: QuestionView. Controller we have: QuestionViewController.
- QuestionViewController owns QuestionGroup (Model). QuestionViewController update QuestionView as per model QuestionGroup in method showQuestion()
- QuestionViewController update ui and update model in IBAction handleCorrect(_), handleIncorrect(_) and showNextQuestion()

Delegation Pattern[Behavioral Pattern]

The delegation pattern enables an object to use another “helper” object to provide data or perform a task rather than do the task itself. This pattern has three parts:



An object needing a delegate , also known as the delegating object. It’s the object that has a delegate. The delegate is usually held as a weak property to avoid a retain cycle where the delegating object retains the delegate, which retains the delegating object.(Ex: UITableView)

A delegate protocol , which defines the methods a delegate may or should implement. (ex. UITableViewDelegate)

A delegate , which is the helper object that implements the delegate protocol. (ex Your Custom view controller which conforms UITableviewDatasource and UITableViewDelegate)

By relying on a delegate protocol instead of a concrete object, the implementation is much more flexible: any object that implements the protocol can be used as the delegate!

When should you use it?
Delegate relationships are common throughout Apple frameworks, especially UIKit . Both DataSource - and Delegate -named objects actually follow the delegation pattern, as each involves one object asking another to provide data or do something/action.

Why isn’t there just one protocol, instead of two, in Apple frameworks?
Apple frameworks commonly use the term DataSource to group delegate methods that provide data. For example, UITableViewDataSource is expected to provide UITableViewCells to display.
Apple frameworks typically use protocols named Delegate to group methods that receive data or events. For example, UITableViewDelegate is notified whenever a row is selected.

It’s common for the dataSource and delegate to be set to the same object, such as the view controller that owns a UITableView . However, they don’t have to be, and it can be very beneficial at times to have them set to different objects.

What should you be careful about?
You should also be careful about creating retain cycles. Most often, delegate properties should be weak.

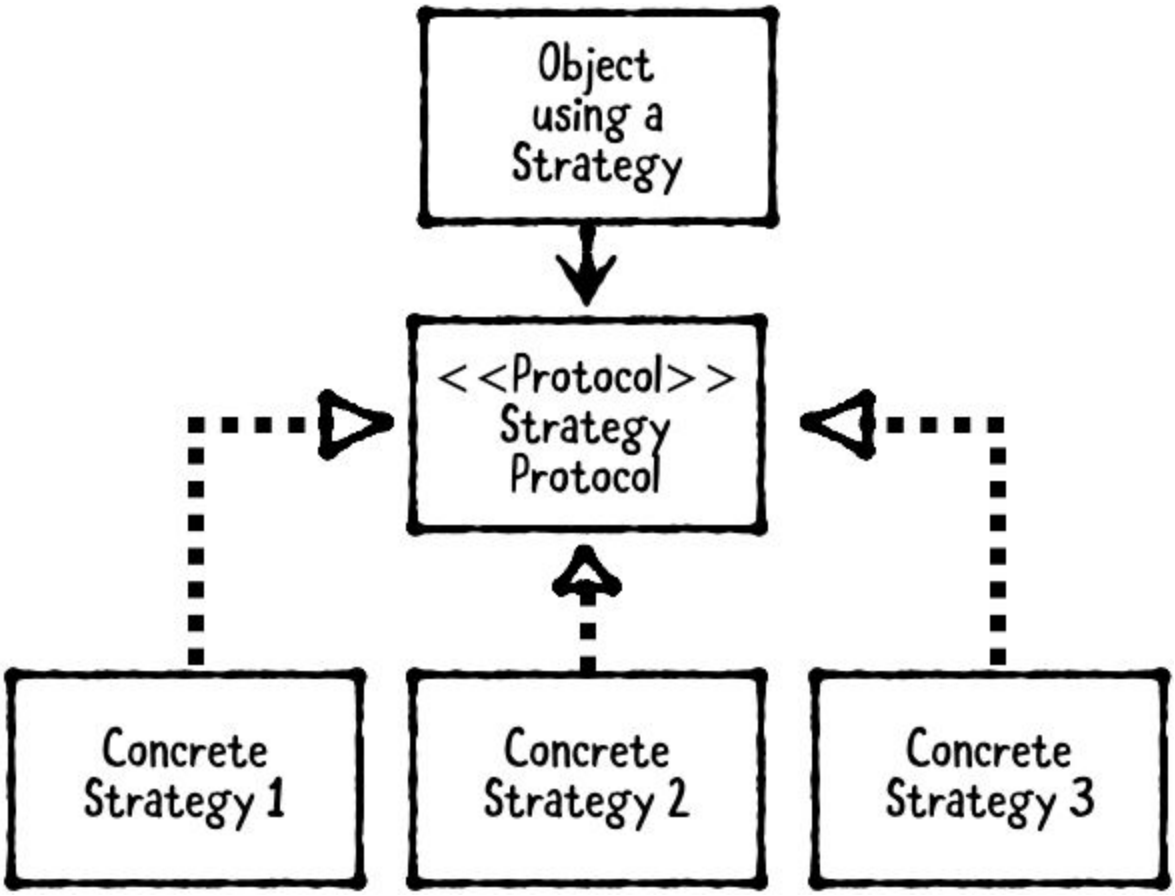
Tip: If an object must absolutely have a delegate set, consider adding the delegate as an input to the object's initializer and marking its type as forced unwrapped using ! instead of optional via ? . This will force consumers to set the delegate before using the object.

Tutorial project

- SelectQuestionGroupViewController has acted as a delegate object for apple defined delegate UITableViewDataSource and UITableViewDelegate.
- QuestionViewController implemented custom delegate pattern; QuestionViewController (An object needing a delegate); QuestionViewControllerDelegate(A delegate protocol) and SelectQuestionGroupViewController (A delegate)

Strategy Pattern[Behavioral Pattern]

The strategy pattern defines a family of interchangeable objects that can be set or switched at runtime.



The **object using a strategy**. it can technically be any kind of object that needs interchangeable behavior.

The **strategy protocol** defines methods that every strategy must implement.

The **strategies** are objects that conform to the strategy protocol.

When should you use it?
Use the strategy pattern when you have two or more different behaviors that are interchangeable.

Note:
This pattern is similar to the delegation pattern: both patterns rely on a protocol instead of concrete objects for increased flexibility.
Unlike delegation, the strategy pattern uses a family of objects.
Delegates are often fixed at runtime. it's rare for these to change during runtime.
Strategies, however, are intended to be easily interchangeable at runtime.

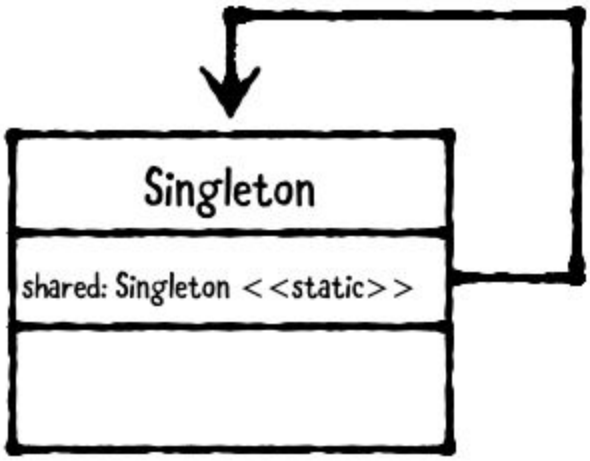
Tutorial project

- QuestionViewController has a questionGroups model but we changed with random and sequence questionGroup. To achieve that we used a strategy pattern.
- Object using a strategy:- QuestionViewController; strategy protocol:- QuestionStrategy and strategy1:- SequentialQuestionStrategy; strategy2:- RandomQuestionStrategy
- QuestionViewController can interchange behavior using QuestionStrategy. It holds QuestionStrategy instance which can be interchangeable with SequentialQuestionStrategy or RandomQuestionStrategy

Singleton Pattern [Creational Pattern]

The singleton pattern restricts a class to only one instance. Every reference to the class refers to the same underlying instance.

This pattern is extremely common in iOS app development, as Apple makes extensive use of it.



The “**singleton plus**” pattern is also common, which provides a shared singleton instance and that allows other instances to be created, too.

When should you use it?
Use the singleton pattern when having more than one instance of a class would cause problems, or when it just wouldn’t be logical.

```
public class MySingleton {
   static let shared = MySingleton()
   //declare init  as private
   private init() { }
}
let mySingleton = MySingleton.shared
// You can't use custom instance
// let mySingleton2 = MySingleton()
```

Use the singleton plus pattern if a shared instance is useful most of the time, but you also want to allow custom instances to be created.

```
public class MySingletonPlus {
   static let shared = MySingletonPlus()
   // declare init  as public
   public init() { }
}
let singletonPlus = MySingletonPlus.shared
// You can use custom instance
let singletonPlus2 = MySingletonPlus()
```

Tutorial Projects:

- Created AppSettings singleton class; use this to manage app-wide settings. (it doesn’t make sense to have multiple, app-wide settings, so you make this a true singleton, instead of a singleton plus.)

Memento Pattern [Behavioral Pattern]

The memento pattern allows an object to be saved and restored.

The **originator** is the object to be saved or restored.
The **memento** represents a stored state.
The **caretaker** requests a save from the originator and receives a memento in response.
The **caretaker** is responsible for persisting the memento and, later on, providing the memento back to the originator to restore the originator's state.

iOS apps typically use an **`Encoder`** to encode an originator's state into a memento, and a **`Decoder`** to decode a memento back to an originator. This allows encoding and decoding logic to be reused across originators.
For example, JSONEncoder and JSONDecoder allow an object to be encoded into and decoded from JSON data respectively.

When should you use it?
Use the memento pattern whenever you want to save and later restore an object's state.
For example, you can use this pattern to implement a save game system, where the originator is the game state (such as level, health, number of lives, etc), the memento is saved data, and the caretaker is the gaming system.

```
// MARK: - Originator
public class Game: Codable {
	public class State: Codable {
		public var attemptsRemaining: Int = 3
		public var level: Int = 1
		public var score: Int = 0
	}

	public var state = State()

	public func rackUpMassivePoints() {
		state.score += 9002
	}

	public func monstersEatPlayer() {
		state.attemptsRemaining -= 1
	}
}
```

```
// MARK: - Memento
typealias GameMemento = Data

// MARK: - CareTaker
public class GameSystem {

	private let decoder = JSONDecoder()
	private let encoder = JSONEncoder()
	private let userDefaults = UserDefaults.standard

	public func save(_ game: Game, title: String) throws {
		let data = try encoder.encode(game)
		userDefaults.set(data, forKey: title)
	}

	public func load(title: String) throws -> Game {
		guard let data = userDefaults.data(forKey: title),
			let game = try? decoder.decode(Game.self, from: data)
		else {
			throw Error.gameNotFound
		}
		return game
	}

	public enum Error: String, Swift.Error {
		case gameNotFound
	}
}

// MARK: - Example
var game = Game()
game.monstersEatPlayer()
game.rackUpMassivePoints()

// Save Game
let gameSystem = GameSystem()
try gameSystem.save(game, title: "Best Game Ever")

// Load Game
game = try! gameSytem.load(title: "Best Game Ever")
print("Loaded Game Score: \(game.state.score)")
```

Output: Loaded Game Score: 9002

The originator is the object to be saved; the memento is a saved state; and the caretaker handles, persists and retrieves mementos.

Tutorial Project:

- QuestionGroup is an object to be saved or restored(originator), Data (Type) is state (memento).
- QuestionGroup is stored in a Data form with help of JSONEncoder. Data is restored in a QuestionGroup object with help of JSONDecoder.
- QuestionGroupCaretaker(Caretaker) is responsible to persist/store QuestionGroup in Data type and is responsible to restore Data type in a QuestionGroup object.

Observer Pattern [Behavioral Pattern]

The observer pattern lets one object observe changes on another object.

This pattern involves three types:

1. The **subscriber** is the "observer" object and receives updates.
2. The **publisher** is the "observable" object and sends updates.
3. The **value** is the underlying object that's changed.

When should you use it?
Use the observer pattern whenever you want to receive changes made on another object.

This pattern is often used with MVC, where the view controller has subscriber(s) and the model has publisher(s). This allows the model to communicate changes back to the view controller without needing to know anything about the view controller's type. Thereby, different view controllers can use and observe changes on the same model type.

Apple added language-level support for this pattern in Swift 5.1 with the addition of Publisher in the **Combine** framework.
Swift 5.1 makes it easy to implement the observer pattern using @Published properties.

```
// Import Combine, which includes the @Published annotations and Publisher and Subscriber types.
import Combine
// Need to declare class; @Published property cannot be used on structs.
public class User {
      // @Published; this tells xcode to automatically generate a Publisher for this property. You cannot use @Published for let properties.
```

```
        @Published var name: String
        public init(name: String) {
                self.name = name
        }
}
```

Example:

```
let user = User(name: "Ray")
// Access the publisher for broadcasting changes to the user's name via
user.$name.
//This returns an object of type Publisher<String>.Publisher; This object is
what can be listened to for update.
let publisher = user.$name
// Create a Subscriber by calling sink on the publisher. This takes a closure
for which is called for the initial value and anything the value changes.
var subscriber: AnyCancellable? = publisher.sink() {
print("User's name is \($0)")
}
user.name = "Vicki"

// By setting the subscriber to nil, it will not longer receive updates from
the publisher.
subscriber = nil
user.name = "Ray has left the building"
```

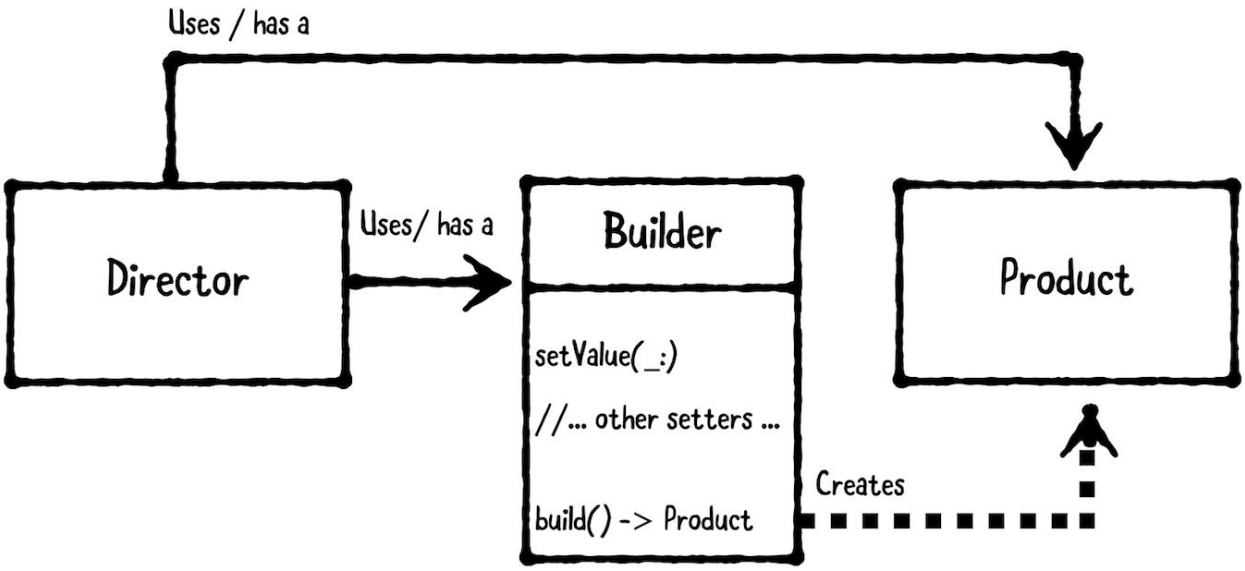
Output:
User's name is Ray
User's name is Vicki

Tutorial Project:

- QuestionGroup is publisher and QuestionGroupCell is subscriber
- We have @Published annotation for runningPercentage property in QuestionGroup. percentageSubscriber is property in QuestionGroupCell
- Combine is functional programming.

Builder Pattern[Creational Pattern]

The builder pattern allows you to create complex objects by **providing inputs step-by-step**, instead of requiring all inputs upfront via an initializer.



The **director** accepts inputs and coordinates with the builder. (This is usually a view controller or a helper class that's used by a view controller.)
The **product** is the complex object to be created. (It's usually a model, but it can be any type depending on your use case.)
The **builder** accepts step-by-step inputs and handles the creation of the product. This is often a class, so it can be reused by reference.

When should you use it?
Use the builder pattern when you want to create a complex object **using a series of steps**.

This pattern works especially well when a product requires multiple inputs . The builder abstracts how these inputs are used to create the product, and it accepts them in whatever order the director wants to provide them.

For example, you can use this pattern to implement a "hamburger builder." The product could be a hamburger model, which has inputs such as meat selection, toppings and sauces. The director could be an employee object, which knows how to build hamburgers, or it could be a view controller that accepts inputs from the user. The "hamburger builder" can thereby accept meat selection, toppings and sauces in any order and create a hamburger upon request.

```
// MARK: - Product
public struct Hamburger {
      public let meat: Meat
      public let sauce: Sauces
      public let toppings: Toppings
}

extension Hamburger: CustomStringConvertible {
```

```
	public var description: String {
		return meat.rawValue + " burger"
	}
}

public enum Meat: String {
	case beef
	case chicken
	case kitten
	case tofu
}

public struct Sauces: OptionSet {
	public static let mayonnaise = Sauces(rawValue: 1 << 0)
	public static let mustard = Sauces(rawValue: 1 << 1)
	public static let ketchup = Sauces(rawValue: 1 << 2)
	public static let secret = Sauces(rawValue: 1 << 3)
	public let rawValue: Int
	public init(rawValue: Int) {
		self.rawValue = rawValue
	}
}

public struct Toppings: OptionSet {
	public static let cheese = Toppings(rawValue: 1 << 0)
	public static let lettuce = Toppings(rawValue: 1 << 1)
	public static let pickles = Toppings(rawValue: 1 << 2)
	public static let tomatoes = Toppings(rawValue: 1 << 3)
	public let rawValue: Int
	public init(rawValue: Int) {
		self.rawValue = rawValue
	}
}

// MARK: - Builder
public class HamburgerBuilder {
	public private(set) var meat: Meat = .beef
	public private(set) var sauces: Sauces = []
	public private(set) var toppings: Toppings = []
	private var soldOutMeats: [Meat] = [.kitten]

	public func addSauces(_ sauce: Sauces) {
		sauces.insert(sauce)
	}

	public func removeSauces(_ sauce: Sauces) {
		sauces.remove(sauce)
```

```
        }
        public func addToppings(_ topping: Toppings) {
                toppings.insert(topping)
        }

        public func removeToppings(_ topping: Toppings) {
                toppings.remove(topping)
        }

        public func setMeat(_ meat: Meat) {
                guard isAvailable(meat) else { throw Error.soldOut }
                self.meat = meat
        }

        public func isAvailable(_ meat: Meat) -> Bool {
                return !soldOutMeats.contains(meat)
        }

        public func build() -> Hamburger {
                return Hamburger(meat: meat, sauce: sauces, toppings: toppings)
        }
}

public enum Error: Swift.Error {
        case soldOut
}

// MARK: - Director
public class Employee {
        public func createCombo1() throws -> Hamburger {
                let builder = HamburgerBuilder()
                try builder.setMeat(.beef)
                builder.addSauces(.secret)
                builder.addToppings([.lettuce, .tomatoes, .pickles])
                return builder.build()
        }

        public func createKittenSpecial() throws -> Hamburger {
                let builder = HamburgerBuilder()
                try builder.setMeat(.kitten)
                builder.addSauces(.mustard)
                builder.addToppings([.lettuce, .tomatoes])
                return builder.build()
        }
}
```

What should you be careful about?

The builder pattern works best for creating complex products that require multiple inputs using a series of steps. If your product doesn't have several inputs or can't be created step by step, the builder pattern may be more trouble than it's worth.

Instead, consider providing convenience initializers to create the product.

Tutorial Project:

- CreateQuestionGroupViewController is the director, and QuestionGroup is the product.
- QuestionGroupBuilder is build QuestionGroup step-by-step.