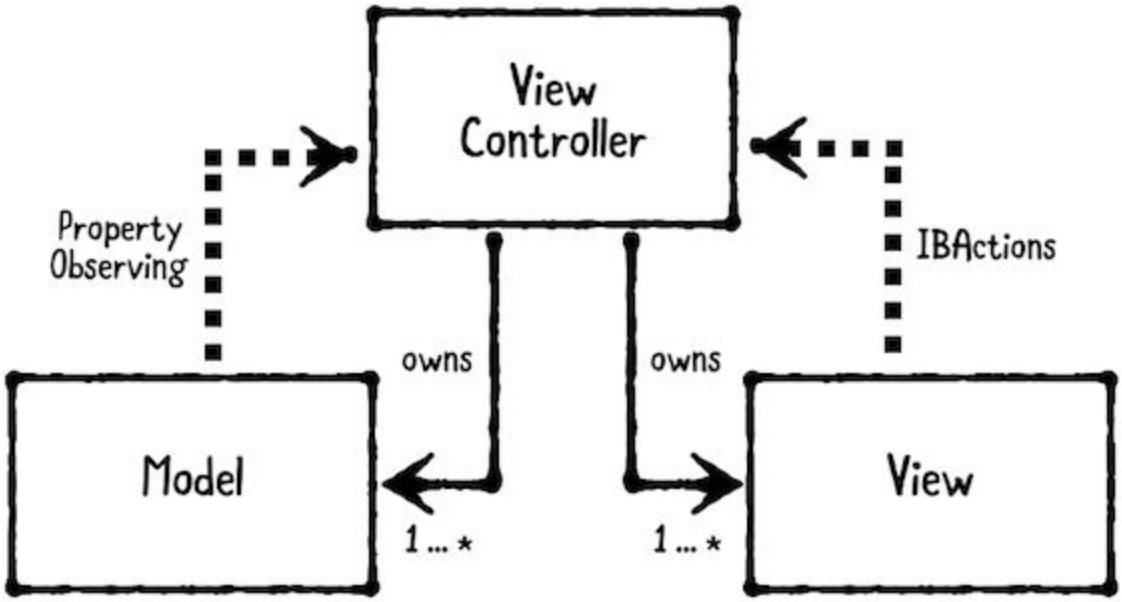
Fundamental Design Patterns

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.

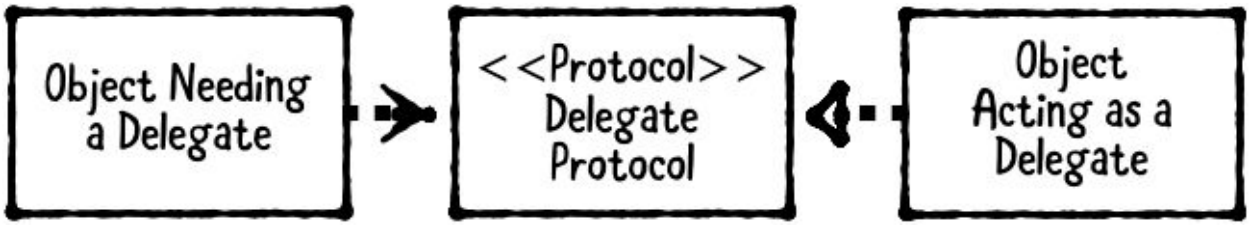
When should you use it?
Use this pattern as a starting point for creating iOS apps. In nearly every app, you’ll likely need additional patterns besides MVC, but it’s okay to introduce more patterns as your app requires them.

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.

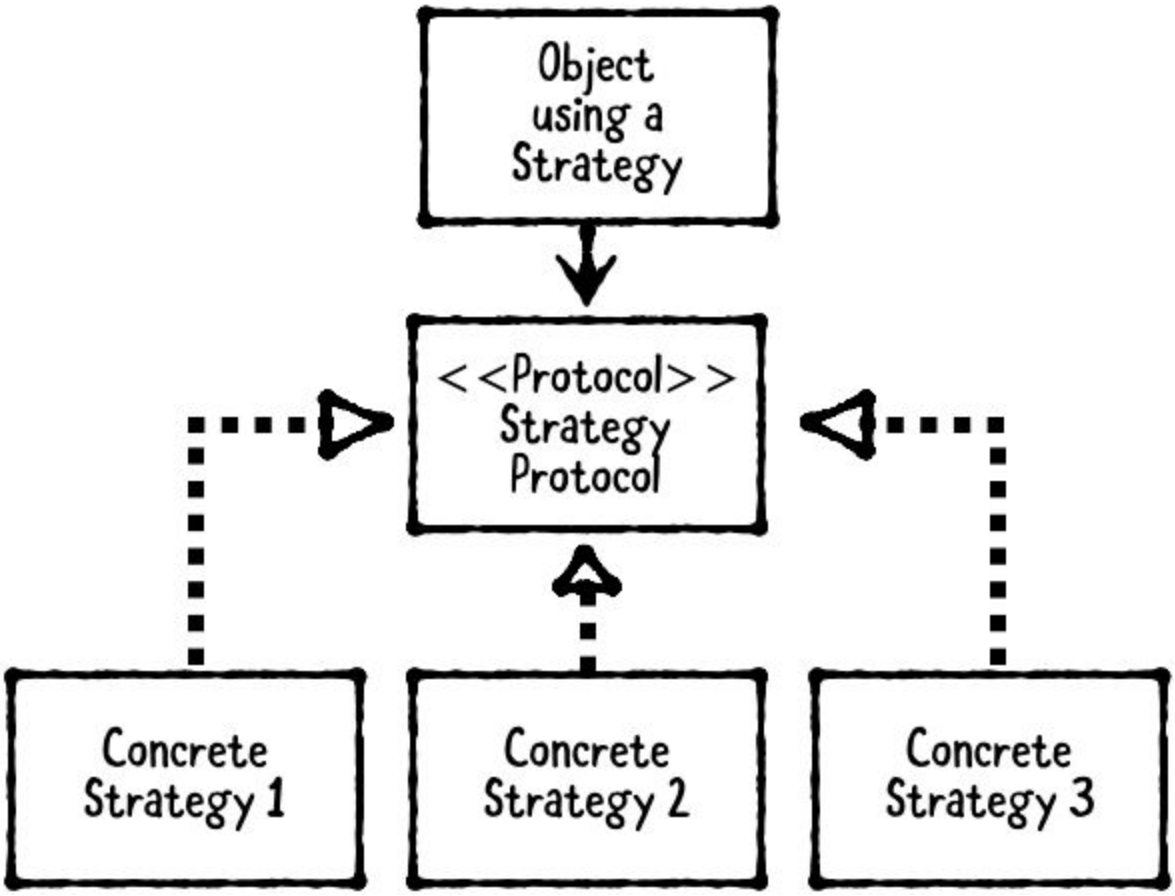
Tip: If an object must absolutely have a delegate set, consider adding the delegate as an input to the object's initializer and marking its type as forced unwrapped using ! instead of optional via ? . This will force consumers to set the delegate before using the object.

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.

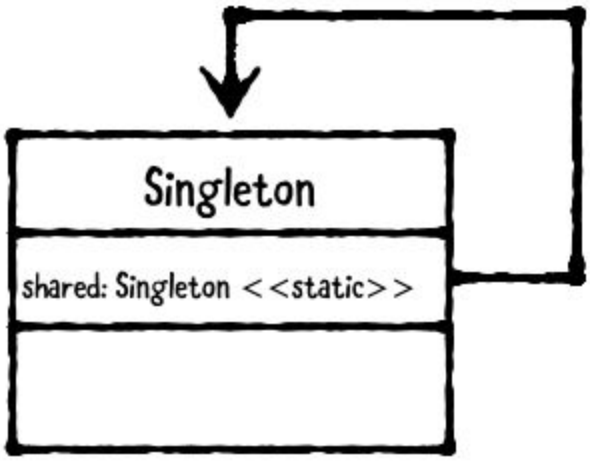
Tutorial project

- 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.)