Delegate Design Pattern in Swift

ddas.tech/delegate-design-pattern-in-swift/

July 28, 2023

In this post we will discuss the Delegate Design Pattern and implement sample code in Swift.

What is Delegate Design Pattern ?

- The delegate design pattern in Swift is a common way to establish communication between objects in an application. It is a behavioral pattern, which means it focuses on how objects collaborate with each other to perform tasks.
- In the delegate pattern, one object (the delegating object) delegates a task or responsibility to another object (the delegate) to handle it on its behalf. The delegating object retains a reference to its delegate and calls its methods when it needs to communicate information or ask for a response.
- In Swift, the delegate pattern is implemented using **protocols**. The delegating object defines a protocol that declares the methods that the delegate can implement to handle the delegated tasks. The delegate object conforms to the protocol by implementing the required methods.

Lets start with a simple example in Swift

```
// Protocol defining the delegate methods
protocol MyViewDelegate {
    func didDoSomething()
}

// Class that will use the delegate
class MyView {
    var delegate: MyViewDelegate?

    func doSomething() {
        // Do some task here...
        print("MyView does some work.")
        print("Once the work is completed it notifies the delegate (A class which has
implemented the protocol (MyViewDelegate) that the task has been completed")
        // Notify the delegate that the task has been completed
        delegate?.didDoSomething()
    }
}

// Class that will act as the delegate
class ControllerClass: MyViewDelegate {
    func didDoSomething() {
        print("ControllerClass - The task is complete!")
    }
}
```

Example Usage for the above

```
let view1 = MyView()
let controller1 = ControllerClass()

view1.delegate = controller1
view1.doSomething() // This will trigger the delegate method and print "The task is
complete!"

//In the above example view1 is the delegating object and controller 1 is the
delegate.

//Output
//MyView does some work.
//Once the work is completed it notifies the delegate (A class which has implemented
the protocol (MyViewDelegate) that the task has been completed
//ControllerClass - The task is complete!
```

Lets try with another example

```
//Example Two
//let's define the delegate protocol:
protocol MessageDeliveryDelegate: AnyObject {
    func messageDelivered(message: String, to recipient: String)
    func messageFailed(message: String, to recipient: String, error: Error)
}

//Next, we'll create a class that sends messages and notifies the delegate about the
message delivery status:
class MessageSender {
    weak var delegate: MessageDeliveryDelegate?

    func sendMessage(_ message: String, to recipient: String) {
        // Simulate sending the message (e.g., through a network request)
        let isMessageDelivered = true // Replace this with the actual result of
sending the message

        if isMessageDelivered {
            delegate?.messageDelivered(message: message, to: recipient)
        } else {
            let error = NSError(domain: "com.example.messagingapp", code: 1001,
userInfo: [NSLocalizedDescriptionKey: "Failed to deliver the message."])
            delegate?.messageFailed(message: message, to: recipient, error: error)
        }
    }
}

//Now, we'll create a class that acts as the delegate and receives the message
delivery status notifications:
class MessageStatusDelegate: MessageDeliveryDelegate {
    func messageDelivered(message: String, to recipient: String) {
        print("Message '\(message)' delivered successfully to \(recipient).")
    }

    func messageFailed(message: String, to recipient: String, error: Error) {
        print("Failed to deliver message '\(message)' to \(recipient). Error: \
(error.localizedDescription)")
    }
}
```

Example Usage

```
    func delegateExampleTwo(){
        let messageSender = MessageSender()
        let messageStatusDelegate = MessageStatusDelegate()
        messageSender.delegate = messageStatusDelegate
        messageSender.sendMessage("Hello, how are you?", to: "John")
    }
//Message 'Hello, how are you?' delivered successfully to John.
```

In the above example When the sendMessage method is called, it simulates sending the message and then notifies the delegate about the delivery status. If the message is delivered successfully, the messageDelivered method of the delegate will be called. If there is an error in delivering the message, the messageFailed method will be called with the corresponding error information.

Using the delegate design pattern in this messaging app example allows for loose coupling between the sender and the receiver of the message delivery status notifications. It provides a way to handle message delivery status asynchronously and keeps the messaging logic separate from the notification handling logic.

Delegate Design Pattern Benefits

The Delegate design pattern offers several benefits that make it a valuable approach to establishing communication and passing data between objects. Some of the key advantages includes:

- **Decoupling of Components**: The Delegate pattern allows us to separate the responsibilities and concerns of different components. The delegating class doesn't need to know the specifics of how a task is handled; it only knows that the delegate will take care of it. This reduces the interdependence between classes, making the code more modular and maintainable.
- **Flexibility and Extensibility:** Delegates provide a flexible way to extend the behavior of a class without modifying its code. We can introduce new delegate classes that conform to the same protocol, allowing for additional functionality to be added without changing the existing codebase.
- **Promotes Single Responsibility Principle:** The Delegate pattern encourages the Single Responsibility Principle, which states that a class should have only one reason to change. By delegating tasks to separate delegate classes, each class has a well-defined responsibility, making it easier to manage and understand the codebase.
- **Protocol-Oriented Programming (POP):** Swift's Delegate pattern is typically implemented using protocols, which are a core feature of Protocol-Oriented Programming. POP encourages code reuse, testability, and composability by focusing on defining protocols and their implementations rather than inheriting from specific classes.
- **Easy Customization and Configuration:** Delegates enable customization by allowing us to replace or modify the default behavior of a class. For example, UI components in iOS and macOS often have delegates that allows us to customize their appearance, behavior, or handle user interactions.
- **Loose Coupling:** The Delegate pattern leads to loose coupling between objects. Since the delegating object holds a weak reference to the delegate, there is no strong ownership, preventing strong retain cycles and memory leaks.

- **Event Handling:** The Delegate pattern is commonly used for event handling scenarios. It allows one object to notify other objects when an event occurs, such as button taps, data updates, or network responses.
- **Clear Communication Channel:** By using delegate methods with descriptive names, the code becomes more self-documenting. It's easier for developers to understand what's happening in the system and how different components are interacting.
- **Unit Testing:** Delegates make it easier to mock or stub dependencies during unit testing. By creating custom delegate implementations, we can control and observe the interactions between objects, making testing more manageable.
- **Reduces Callback**: In situations where a lot of callbacks or closures might be required, using a delegate can help organize and manage these interactions in a structured manner, reducing the complexity often associated with callback chains.
- ***Overall, the Delegate design pattern is a powerful tool that enhances code organization, promotes reusability, and fosters good software design principles like loose coupling and single responsibility. It is commonly used in iOS and macOS development, but its benefits extend to other programming domains as well.***

Summary of other design patterns https://ddas.tech/introduction-to-design-patterns/