KVC KVO Design Pattern in Swift

ddas.tech/kvc-kvo-design-pattern-in-swift/

July 28, 2023

KVC KVO Design Pattern in Swift – **KVC (Key-Value Coding)** and **KVO (Key-Value Observing)** are related design patterns in Cocoa and Cocoa Touch frameworks that allow for efficient and dynamic access to object properties and enable objects to observe changes to those properties, respectively.

KVC (Key-Value Coding) is a design pattern that allows access to an object's properties using a string key instead of using the traditional dot notation. This can be useful when working with dynamic or unknown property names, such as when parsing JSON data. Using KVC, we can easily access and set property values by key rather than writing custom code for each property.

Here's an example of using KVC in Swift:

In this example, we have a Person class with name and age properties. Using KVC, we can access and set the value of the age property using the string "age" as the key.

```
class Person: NSObject {
    @objc dynamic var name: String
    @objc dynamic var age: Int

    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}

let person = Person(name: "John", age: 30)
person.setValue(35, forKey: "age")
print(person.age) // Output: 35
```

KVO (Key-Value Observing) is another design pattern that allows objects to observe changes to an object's property. When an observed property is changed, the observing object is automatically notified so it can update its own state accordingly. KVO can be useful in situations where we need to know when a property value changes, but you don't want to manually monitor it.

In the below example, we have a Employee class with name and age properties. We also have an Observer class that observes changes to the age property of an Employee object. When the age property of the Employee object changes, the

observeValue(forKeyPath:of:change:context:) method in the Observer class is automatically called with information about the change, allowing the Observer object to update its own state accordingly.

Note that in order for KVO to work, the observed property must be declared with the **@objc dynamic** attribute to enable dynamic dispatch. Additionally, the observing object should call **addObserver(_:forKeyPath:options:context:)** on the observed object to register for notifications, and call **removeObserver(_:forKeyPath:)** when it no longer needs

```
class Employee: NSObject {
    @objc dynamic var name: String
    @objc dynamic var age: Int

    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}

class Observer: NSObject {
    @objc var employee: Employee

    init(employee: Employee) {
        self.employee = employee
        super.init()
        self.employee.addObserver(self, forKeyPath: "age", options: .new, context:
nil)
    }

    deinit {
        self.employee.removeObserver(self, forKeyPath: "age")
    }

    override func observeValue(forKeyPath keyPath: String?,
                               of object: Any?,
                               change: [NSKeyValueChangeKey : Any]?,
                               context: UnsafeMutableRawPointer?) {
        if keyPath == "age" {
            print("Employee's age changed to \(employee.age)")
        }
    }
}

let employee = Employee(name: "John", age: 30)
let observer = Observer(employee: employee)
employee.age = 38 // Output: "Employee's age changed to 38"
```

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