Engenharia de Software Moderna

Marco Tulio Valente
https://engsoftmoderna.info/

Capítulo 6: Padrões de Projeto

Entendimento do Capítulo

O capítulo 6 do livro "Engenharia de Software Moderna" apresenta os padrões de projeto, que são soluções reutilizáveis para problemas recorrentes no desenvolvimento de software. Esses padrões foram introduzidos no famoso livro da Gang of Four (GoF), que descreve 23 padrões divididos em três categorias: criacionais, estruturais e comportamentais. O objetivo principal dos padrões é tornar o design de sistemas mais flexível, extensível e preparado para mudanças futuras.

Uma coisa que me chamou atenção logo no início foi a ideia de "design for change", ou seja, projetar pensando nas mudanças inevitáveis que virão. Isso faz muito sentido já que frequentemente surgem novos requisitos ou alterações no meio do caminho. Se o sistema não estiver bem projetado, qualquer mudança pode se transformar em um caos.

Outro ponto interessante é como os padrões de projeto ajudam a criar um vocabulário comum entre os desenvolvedores. Por exemplo, quando alguém menciona que usou um "Singleton" ou um "Decorator", todos que conhecem esses padrões já entendem a solução adotada sem precisar de explicações detalhadas facilitando muito a comunicação dentro de uma equipe.

Aplicação no Mercado

No meu estágio de desenvolvimento iOS, muitos dos padrões apresentados no capítulo podem ser aplicados diretamente. Aqui estão alguns exemplos práticos:

1. **Padrão Singleton**:
- No desenvolvimento iOS, o padrão Singleton é amplamente utilizado para gerenciar recursos globais, como configurações de usuário ou serviços compartilhados. Um exemplo clássico é o uso de `UserDefaults` para armazenar preferências do usuário. Podemos encapsular o acesso ao `UserDefaults` em uma classe Singleton chamada `SettingsManager`, garantindo que haja apenas uma instância centralizada para gerenciar essas configurações.

Exemplo prático:
```
 class SettingsManager {
    static let shared = SettingsManager()
    private init() {}

    func saveThemePreference(_ theme: String) {
        UserDefaults.standard.set(theme, forKey: "theme")
    }

    func getThemePreference() -> String? {
        return UserDefaults.standard.string(forKey: "theme")
    }
}
```

- Dessa forma, sempre que precisarmos salvar ou recuperar as preferências de tema, usamos `SettingsManager.shared`.

2. **Padrão Fábrica**:

- . Em vez de instanciar cada tipo de notificação diretamente, criamos uma fábrica que decidia qual classe de notificação usar com base no tipo recebido.

Exemplo prático:

```
protocol Notification {
    func send(message: String)
}

class PushNotification: Notification {
    func send(message: String) {
        print("Enviando push: \(message)")
    }
}

class EmailNotification: Notification {
    func send(message: String) {
        print("Enviando email: \(message)")
    }
}

class NotificationFactory {
    static func create(type: String) -> Notification {
        switch type {
        case "push":
            return PushNotification()
        case "email":
            return EmailNotification()
        default:
            fatalError("Tipo de notificação desconhecido")
        }
    }
}

// Uso
let notification = NotificationFactory.create(type: "push")
notification.send(message: "Olá, mundo!")
```

3. **Padrão Observador**:

- Esse padrão é muito útil em aplicações iOS, especialmente ao trabalhar com o padrão arquitetural MVVM (Model-View-ViewModel) que estou mais familiarizado. Usamos o padrão Observador para atualizar a interface do usuário automaticamente quando os dados no modelo mudam. Por exemplo, ao monitorar mudanças em um valor usando `Combine` ou `KVO` (Key-Value Observing).

Exemplo prático:

```
import Combine

class TemperatureModel {
    @Published var temperature: Double = 0.0
}

class TemperatureViewModel {
```

```
    private var model: TemperatureModel
    private var cancellables = Set<AnyCancellable>()

    init(model: TemperatureModel) {
        self.model = model
        self.model.$temperature
            .sink { newTemperature in
                print("Temperatura atualizada: \(newTemperature)")
            }
            .store(in: &cancellables)
    }
}

// Uso
let model = TemperatureModel()
let viewModel = TemperatureViewModel(model: model)
model.temperature = 25.0
```

4. **Padrão Decorador**:

- Em vez de modificar diretamente a classe base, usamos o padrão Decorador para adicionar opções como log e formatação das mensagens.

Exemplo prático:

```
 protocol Message {
    func display()
}

class SimpleMessage: Message {
    func display() {
        print("Mensagem simples")
    }
}

class MessageDecorator: Message {
    private let wrapped: Message

    init(wrapped: Message) {
        self.wrapped = wrapped
    }

    func display() {
        wrapped.display()
    }
}

class LogMessageDecorator: MessageDecorator {
    override func display() {
        print("Log: Antes de exibir a mensagem")
        super.display()
        print("Log: Depois de exibir a mensagem")
    }
}

// Uso
let message = LogMessageDecorator(wrapped: SimpleMessage())
```

```
message.display()
```

Exemplos Práticos

Exemplo 1: Singleton no Desenvolvimento iOS

No capítulo, o autor explica como o padrão Singleton garante que uma classe tenha apenas uma instância. No meu caso, usei esse padrão para criar um gerenciador de sessão de usuário no app. A classe `SessionManager` armazena informações sobre o usuário logado e é acessada globalmente durante toda a execução do app.

```
class SessionManager {
 static let shared = SessionManager()
 private init() {}

 var currentUser: String?

 func login(user: String) {
    currentUser = user
    print("\(user) logado com sucesso.")
 }

 func logout() {
    currentUser = nil
    print("Usuário deslogado.")
 }
}

// Uso
SessionManager.shared.login(user: "João")
print(SessionManager.shared.currentUser ?? "Nenhum usuário logado")
```

Exemplo 2: Observador no MVVM

O padrão Observador, como mostrado no capítulo, é ideal para desacoplar classes de modelo e interface. Monitoraramos mudanças no estado de carregamento de dados em uma tela.

```
import Combine

class LoadingState {
 @Published var isLoading: Bool = false
}

class LoadingViewModel {
 private var state: LoadingState
 private var cancellables = Set<AnyCancellable>()

 init(state: LoadingState) {
    self.state = state
    self.state.$isLoading
        .sink { isLoading in
            print(isLoading ? "Carregando..." : "Pronto!")
        }
        .store(in: &cancellables)
 }
```

```
}

// Uso
let state = LoadingState()
let viewModel = LoadingViewModel(state: state)
state.isLoading = true
state.isLoading = false
```

Exemplo 3: Fábrica para Criar ViewControllers

Muitas vezes precisamos criar várias telas (ViewControllers) dinamicamente. Usar uma fábrica para isso ajuda a centralizar a lógica de criação.

```
import UIKit

class ViewControllerFactory {
 static func create(type: String) -> UIViewController {
    switch type {
    case "home":
       return HomeViewController()
    case "profile":
       return ProfileViewController()
    default:
       fatalError("Tipo de ViewController desconhecido")
    }
 }
}

// Uso
let homeVC = ViewControllerFactory.create(type: "home")
```