Unidade 5

Aula 1

# MVC, Decorator, Adapter e Facade

Durante todos os módulos deste curso de desenvolvimento de apps, você foi apresentado a diversos padrões de projeto. Nesta unidade você reencontrará alguns deles, adaptado para a Swift, e conhecerá alguns novos. Diante de tantos padrões de projeto, iremos subir para um nível macro e mostrar as categorias de padrões de projeto. Antes de tudo isto, vamos relembrar o MVC.

## MVC

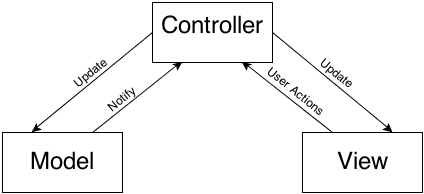
O Model-View-Controller (MVC) é uma arquitetura de software que esta presente no iOS e é, sem dúvida, o padrão de projeto mais utilizado de todos. Ele classifica os objetos de acordo com seu papel geral na sua aplicação e incentiva a separação limpa de código, baseada no seu papel. Até então nenhuma novidade sobre o que aprendemos no curso de POO.

As três funções das camadas do MVC são:

* **Model**: São objetos que contém os dados do aplicativo e definem como manipulá-los.
* **View**: São objetos que estão a cargo da representação visual do modelo e os controles que o usuário pode interagir. Basicamente, todos os objetos UIView e suas subclasses estão classificados nesta camada.
* **Controller**: O Controller (controlador) é o mediador que coordena todo o trabalho. Ele acessa os dados do modelo e mostra-os com as views, escuta eventos e manipula os dados, conforme necessário. Você consegue adivinhar qual classe é o seu controlador? É isso mesmo: ViewController.

DICA: A classe UIViewController tem este nome composto por View e Controller, mas não significa que ela está presente na camada View e Controller, significa que ela é uma controladora de views, e está presente apenas na camada Controller.

A comunicação entre a View e o Model através do Controller pode ser melhor descrita com o seguinte esquema:



O model notifica o controller de quaisquer alterações de dados, que por sua vez, (o controller) atualiza a exibição dos dados nas views. A view, por sua vez, pode notificar ao controller sobre ações e eventos de interações do usuário, e neste ponto de vista o controller irá atualizar o modelo de dados, caso seja necessário.

Tudo se resume a separação de código e reutilização. Idealmente, a view deve ser completamente separada do model. Se a view não depende de uma implementação específica do model, então ela pode ser reutilizada com um modelo diferente e apresentar alguns outros dados.

Por exemplo, se no futuro você também gostaria de adicionar filmes ou livros para o nosso aplicativo de música, você ainda pode usar a mesma view usada para mostrar a música para exibir seu filme ou livro e trocaríamos apenas o modelo de dados. Outro ponto é exibir o mesmo modelo de dados de diferentes formas, trocando apenas as views e mantendo o modelo de dados. Essa é a força do MVC!

Agora que você sabe que uma aplicação deve estar totalmente regrada no MVC, vamos conhecer as categorias dos padrões de projeto.

## Categorias dos Padrões de Projeto

Estamos mais uma vez falando dos design patterns e insistindo na sua importância. Toda estes padrões mostrados realmente irão fazer a diferença como profissional programador de software e irá te garantir destaque entre os demais programadores que não conhecem este assunto. Agora que você já teve mais contato suficiente com os padrões na prática, vamos então terminar este assunto com alguns conceitos adicionais, desta aula.

Design Patterns podem agilizar o processo de desenvolvimento provendo técnicas de desenvolvimento já testadas e consagradas por muitos programadores. Uma projeção de software efetiva requer que consideremos questões que não são tão explícitas durante o desenvolvimento mas, futuramente, se tornará um problema. **Reutilizar** padrões de design (projeto) ajuda a evitar problemas sutis que podem causar grandes falhas no sistema, além disso melhora a legibilidade do código para programadores e arquitetos de software que estão familiarizados com os padrões.

Os padrões mais conhecidos e utilizados foram criados pelo grupo **GoF** (Gang of Four), que é um grupo de quatro pessoas que resolveram colocar os “pingos nos is” da programação orientada a objetos. Os padrões de projeto concebidos pelo GoF ficaram mundialmente conhecidos como Padrões de Projeto Gof. Estes padrões estão organizados em três categorias: **criacionais, estruturais** e **comportamentais.**

### Padrões de projeto criacionais

Estes padrões de design são todos sobre instanciação de objetos. Estes padrões podem ainda ser divididos em padrões de criação de classes e padrões de criação de objetos. Enquanto os padrões de criação de classe se preocupam em utilizar herança de forma eficaz no processo de instanciação, os padrões de criação de objetos usam técnicas de delegação de atividades para realizar as instanciações.

Exemplos de padrões criacionais (resumido)

* **Abstract Factory:** Cria uma instância de várias famílias de classes
* **Builder:** Separa construção do objeto de sua representação
* **Factory Method:** Cria uma instância de várias classes derivadas
* **Object Pool:** Evita aquisições caras e libera recursos através da reciclagem de objetos que não estão mais em uso.
* **Prototype:** Uma instância totalmente inicializada pronta para ser copiada ou clonada
* **Singleton:** Uma classe no qual apenas uma única instância pode existir.

Como o foco do nosso curso não é padrões de projeto, não será possível explicar cada um deles, mas explore e pesquise cada um deles.

### Padrões de projeto estruturais

Estes padrões de design são todos sobre a classe e composição de objetos. Utilizam a estratégia de compor objetos de diferentes formas de modo a definir novas funcionalidades para estes objetos em tempo de execução.

Exemplos de padrões estruturais:

* **Adapter**: Combina interfaces de diferentes classes
* **Bridge**: Separa a interface de um objeto a partir de sua implementação
* **Composite**: Uma estrutura em árvore de objetos simples e compostos.
* **Decorator**: Adiciona responsabilidades aos objetos dinamicamente
* **Facade**: Uma única classe representa um subsistema inteiro
* **Flyweight**: Uma instância fina (leve) usada para compartilhamento eficiente
* **Private Class Data**: Acesso/modificação são restringidos
* **Proxy**: Um objeto que representa um outro objeto

### Padrões de projeto comportamentais

Estes padrões de design são todos sobre a comunicação entre objetos. Padrões de comportamento são os padrões que são mais especificamente preocupados com a comunicação entre objetos.

Exemplos de padrões comportamentais:

* **Chain of Responsibility**: Uma maneira de passar um pedido entre uma cadeia de objetos
* **Command**: Encapsula um pedido de comando como um objeto
* **Interpreter**: Uma maneira de incluir elementos de linguagem em um programa
* **Iterator**: Acessa sequencialmente os elementos de uma coleção
* **Mediador**: Define uma comunicação simplificada entre as classes
* **Memento**: Captura e restaura o estado interno de um objeto
* **Null Object**: Concebido para funcionar como um valor padrão de um objeto
* **Observer**: Uma maneira de notificar a mudança para um número de classes
* **State**: Altera o comportamento de um objeto quando ele altera de estado
* **Strategy**: Encapsula um algoritmo dentro de uma classe
* **Template Method**: Adia as etapas exatas de um algoritmo para uma subclasse
* **Visitor**: Define uma nova operação a uma classe sem alterações

Aula 2

# Closures

Closures são blocos independentes de funcionalidade que podem ser transferidos de um lugar para o outro e utilizados em seu código. Com eles podemos implementar o tratamento de um objeto em tempo de execução. Em outras palavras, podemos criar métodos e passa-los via parâmetro em algum método ou atribuí-lo em alguma variável/constante. Os closures são utilizados principalmente para realizar operações assíncronas.

## Expreções Closure

Para dar continuidade nos exemplos desta aula, abra novamente o seu Playground e execute os códigos mostrados aqui.

Em Swift podemos criar funções aninhadas, ou seja, é possível definir funções dentro de outra função. Isto é feito para organizar melhor o código ou alguma outra aplicação específica que se faz necessário este uso. Mas convenientemente não é muito aconselhável pela orientação a objetos pois diminui a legibilidade do código e quanto mais aninhamos funções dentro de outras funções aninhadas, mais complexo se torna nosso código.

Uma alternativa as funções aninhadas é o uso de closures. Closures são utilizadas para escrever blocos de código inline em uma sintaxe breve e focada. Closures são utilizados para otimizar a escrita de certos algoritmos sem diminuir sua clareza ou intenção. Os exemplos abaixo ilustram estas otimizações refinando o método de ordenação de arrays **sorted(by:)** com diversas maneiras para realizar a mesma funcionalidade.

### O método de ordenação

O Swift apresenta um método chamado **sorted(by:)**, que classifica um array de valores de um tipo conhecido com base na produção de uma saída ordenada por um closure que você fornece. Quando este método completa o processo de classificação, é retornado um novo array de mesmo tipo que o antigo, com seus elementos em uma ordem especificada pelo closure. O array original não é modificado depois deste processo.

O exemplo abaixo usa o método usa o método **sorted(by:)** para ordenar um array de Strings em ordem alfabética reversa (de Z a A). Este é o array inicial a ser classificado:

let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]

O método **sorted(by:)** aceita um bloco closure que recebe dois argumentos com mesmo tipo do conteúdo do array e retorna um valor Bool que indica se o primeiro argumento deve ser posicionado no novo array antes do segundo argumento. O closure de classificação deve retornar **true** se o primeiro valor deve estar antes do segundo, e **false** caso contrário.

Este exemplo está classificando um array de String, e este closure de classificação deve possuir o tipo **(String, String) -> Bool**.

Uma maneira de providenciar uma closure de classificação é escrever uma função normal com o tipo que corresponde ao tipo que o método **sorted(by:)** aceita. O tipo de uma função é definido pela sua assinatura (conjunto de argumentos) e tipo de retorno. Ao definir esta função, basta passarmos esta via parâmetro do método **sorted(by:).**

func backward(\_ s1: String, \_ s2: String) -> Bool {

  return s1 > s2

}

var reversedNames = names.sorted(by: backward)

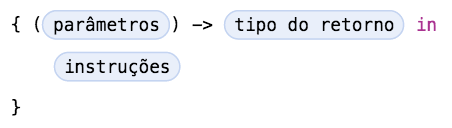
// reversedNames é igual a ["Ewa", "Daniella", "Chris", "Barry", "Alex"]

Veja que não invocamos **backward()** mas passamos apenas o **nome** do método via parâmetro de **sorted(by:)**, isto porque este método de classificação não espera um valor ou retorno, mas sim a definição de como ele irá classificar o array.

Se a primeira string (s1) é maior do que a segunda (s2), a função **backward()**  irá retornar **true**, indicando que s1 deve aparecer antes de s2 no array classificado. Para caracteres em strings, “maior que” significa “aparece antes no alfabeto que”. Isto significa que “B” é maior do que “A”, e a string “Tom” é maior que a string “Tim”. Como este exemplo ordena as strings na ordem alfabética inversa então “Bruno” será apresentado antes de “Alex”.

### Sintaxe de um closure

Closure expression syntax has the following general form:



Veja que esta sintaxe se assemelha a declaração de uma função, porém não precisamos definir um nome a ela (e nem utilizar a palavra reservada **func**).Para definir um closure, especificamos um conjunto de parâmetros, o tipo de retorno e as instruções no interior do bloco.

O exemplo abaixo mostra o exemplo do **sorted(by:)** com o método **backward()** sendo substituído por um closure inline.

reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in

return s1 > s2

})

Note que os parâmetros e retorno deste closure inline é idêntico ao da função **backward()**. Em ambos os casos, eles são escrito com **(s1: String, s2: String) -> Bool.** Entretanto, para o caso inline, o parâmetro e o tipo de retorno estão escritos dentro de chaves ({ }), e não fora delas.

O inicio do corpo do closure é introduzido pela palavra-chave **in**. Esta palavra chave indica que a definição dos parâmetros e tipo de retorno do closure terminou, e daí em diante o corpo do closure começa.

Como o corpo do closure é muito curto, ele pode ser escrito em uma única linha:

reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in return s1 > s2 } )

Isto ilustra que em geral, a chamada de **sorted(by:)** permance a mesma. Um par de parênteses envolve inteiramente o argumento do método. Entretanto, este argumento agora é um closure inline.

### Inferring Type From Context