Blog: mvvm
Arquitetura iOS MVC, MVP, MVVM

24-11-2020 por ederson

Dando continuidade ao aprendizado sobre Padrões e Arquiteturas, me mantenho nos padrões mais usados para Mobile seja para Android ou iOS.

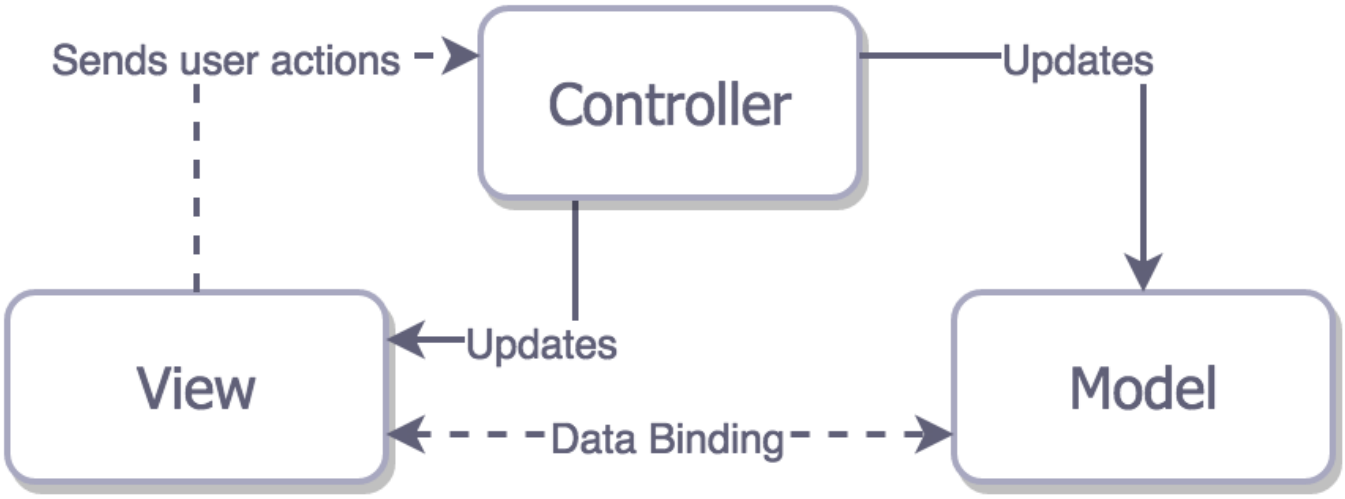
Fundamentos de MV (X) Existem muitas opções quando se trata de padrões de design de arquitetura: MVC ,MVP, MVVM

Os 3 MV's pressupõem colocar as entidades do aplicativo em uma das 3 categorias:

Modelos - responsáveis pelos dados de domínio ou uma camada de acesso a dados que manipula os dados.

Visualizações - responsável pela camada de apresentação, para o ambiente iOS pense em tudo começando com o prefixo ' UI '. Como o mediador entre o Model e a View, responsável por alterar o Model , reagindo às ações do usuário realizadas na View e atualizando a View com as mudanças do Model.

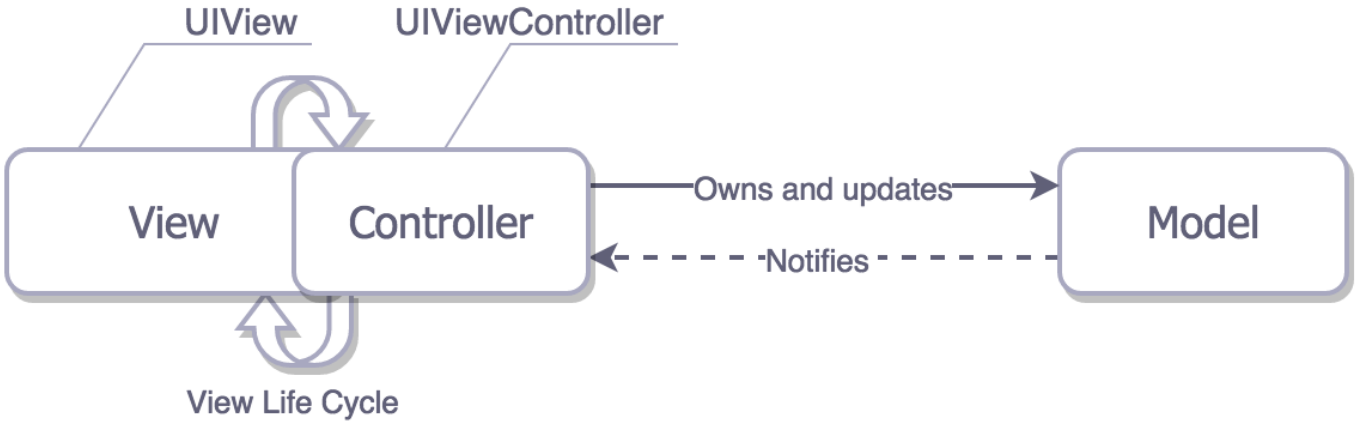
MVC Tradicional por assim dizer, é um pouco diferente do MVC da Apple.



Cacau MVC O Controlador é um mediador entre a Visualização e o Modelo para que eles não se conheçam. O menos reutilizável é o Controlador, já que devemos ter um lugar para toda aquela lógica de negócios complicada que não se encaixa no Modelo .

MVC da Apple

Realistic Cocoa MVC Cocoa MVC incentiva você a escrever controladores de visualização massiva , porque eles estão tão envolvidos no ciclo de vida de visualização que é difícil dizer que são separados. Embora você ainda tenha a capacidade de descarregar parte da lógica de negócios e da transformação de dados para o Modelo, você não tem muita escolha quando se trata de descarregar trabalho para a Visualização, na maioria das vezes, toda a responsabilidade da Visualização é enviar ações para o controlador . O controlador de visualização acaba sendo um delegado e uma fonte de dados de tudo, e você pode deixa-los responsável por despachar e cancelar as solicitações de rede.



A View configurada diretamente com o Model , então as diretrizes do MVC são violadas, mas isso acontece o tempo todo, e geralmente as pessoas não acham que está errado. Se você seguir estritamente o MVC, deverá configurar a célula a partir do controlador e não passar o modelo para a visualização , o que aumentará ainda mais o tamanho do seu controlador .

Cocoa MVC não é abreviado como o Controlador Massive View. E podemos ter um problema que pode não ser evidente até que chegue ao Teste de Unidade . Uma vez que seu controlador de visualização está fortemente acoplado com a visualização, torna-se difícil testar porque você tem que ser muito criativo ao simular visualizações e seu ciclo de vida, ao escrever o código do controlador de visualização de tal forma que sua lógica de negócios seja separada tanto quanto possível a partir do código de layout de visualização.

Mas vamos avaliar o Cocoa MVC em termos de características definidas no início do artigo:

Distribution - a View e o Model de fato separados, mas a View e o Controller estão fortemente acoplados.

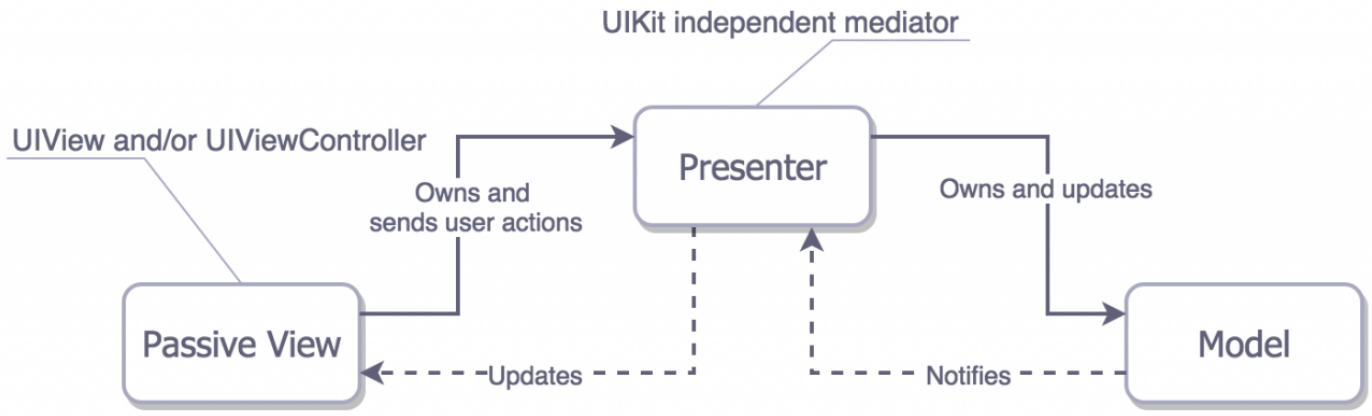
Testabilidade - devido à má distribuição, você provavelmente só testará seu modelo. Facilidade de uso - a menor quantidade de código entre outros padrões. Além disso, todos estão familiarizados com ele, portanto, é facilmente mantido, mesmo por desenvolvedores inexperientes.

Se Cocoa MVC é o padrão de sua escolha, e você não está pronto para investir mais tempo em sua arquitetura e sente que algo com custo de manutenção mais alto é um exagero para seu projeto de estimação minúsculo.

Mas podemos dizer que o Cocoa MVC é o melhor padrão arquitetônico em termos de velocidade de desenvolvimento.

MVP

No MVP a View está fortemente acoplado ao Controller, enquanto o mediador do MVP, Presenter, não tem nada a ver com o ciclo de vida do view controller, não há código de layout no Presenter , mas ele é responsável por atualizar a View com dados e estado.



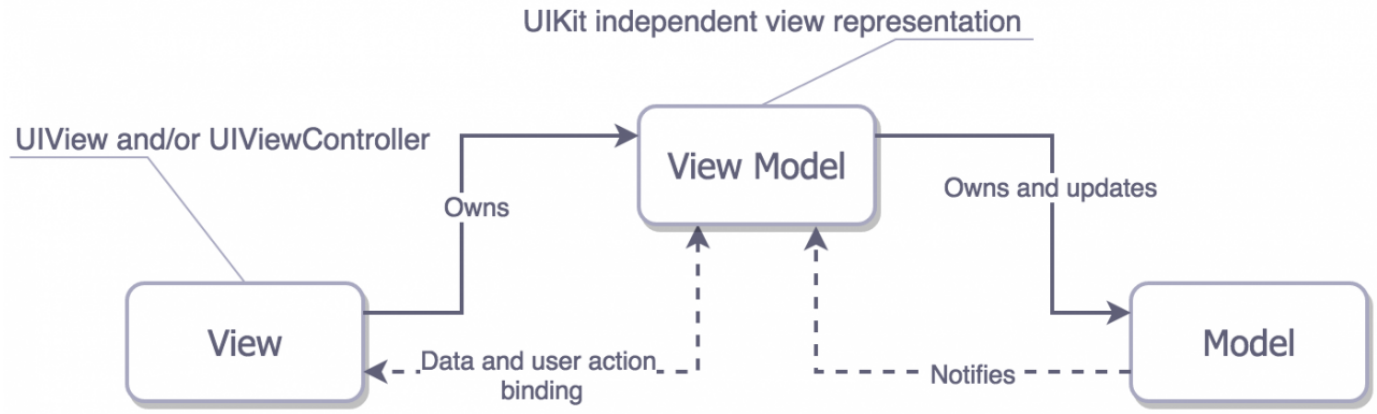
Se dissermos que o UIViewController é o View, em termos de MVP, as subclasses de UIViewController são, na verdade, as visualizações e não os apresentadores. Essa distinção fornece testabilidade excelente, o que tem o custo da velocidade de desenvolvimento, porque você tem que ter dados manuais e associação de eventos.

Vejamos os recursos do MVP: **Distribuição** - tem a maior parte das responsabilidades divididas entre o MVP no iOS, excelente testabilidade e muito código.

MVP Com Bindings e Hooters Existe outro ponto do MVP - o MVP do Controlador de Supervisão. Esta variante inclui vinculação direta da Visualização e do Modelo enquanto o Presenter (O Controlador de Supervisão) ainda lida com ações da Visualização e é capaz de alterar a Visualização .

MVVM O mais recente e o melhor dos MV(X)'s, em teoria, o Model-View-ViewModel parece muito bom. A View e o Model já são familiares para nós, mas também o Mediador, representado como o View Model.

MVVM É muito semelhante ao MVP: o MVVM trata o controlador de visualização como o View Não há um acoplamento forte entre a vista e o modelo Além disso, ele faz ligações como a versão de supervisão do MVP; entretanto, desta vez não entre a vista e o modelo , mas entre a vista e o modelo de vista . Então, qual é o modelo de visualização na realidade do iOS? É basicamente uma representação independente do UIKit de sua Visualização e seu estado. O View Model invoca mudanças no Model e se atualiza com o Model atualizado , e uma vez que temos uma ligação entre View e View



Model , o primeiro é atualizado de acordo.

Bindings As vinculações vêm prontas para o desenvolvimento do OS X, mas não as temos na caixa de ferramentas do iOS. Claro que temos o KVO e as notificações, mas eles não são tão convenientes quanto as ligações. Portanto, desde que não queiramos escrevê-los nós mesmos, temos duas opções: Uma das bibliotecas de ligação baseadas em KVO, como RZDataBinding ou SwiftBond As bestas de programação reactive funcional em escala real , como ReactiveCocoa , RxSwift ou PromiseKit . Se você ouve “MVVM” - você pensa em ReactiveCocoa, e vice-versa. Embora seja possível construir o MVVM com as ligações simples, ReactiveCocoa permitirá que você obtenha a maior parte do MVVM.

Testabilidade - o View Model não sabe nada sobre a View , isso nos permite testá-la facilmente. O modo de exibição também pode ser testado, mas como é dependente do UIKit, você pode querer ignorá-lo.

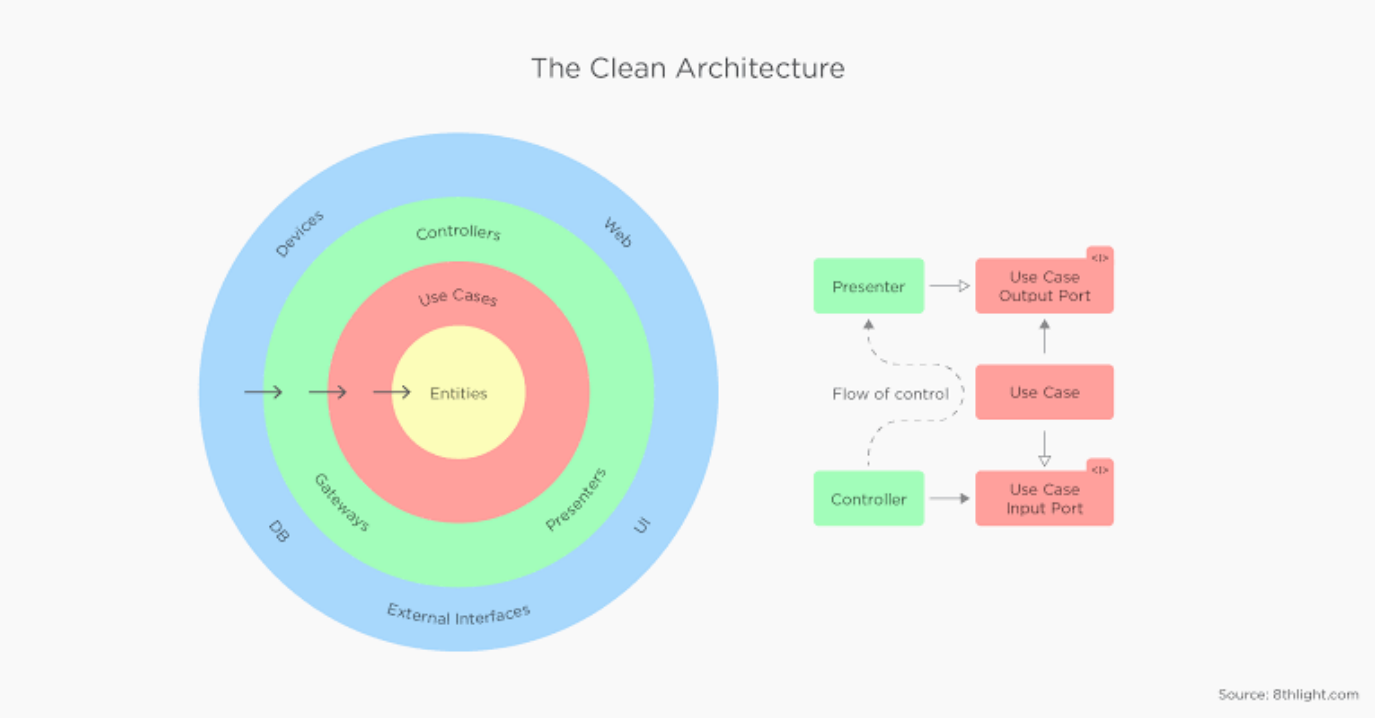
Fácil de usar - O MVVM é muito atraente, pois combina os benefícios das abordagens citadas e, além disso, não requer código extra para as atualizações do View devido aos bindings do lado do View. No entanto, a testabilidade ainda está em um bom nível.

Clean Architecture com MVVM em aplicações Android

09-11-2020 por ederson

A Clean Architecture foi idealizada por Robert C. Martin, autor de um livro abordando este tema, ajudou a criar uma arquitetura em que os componentes fossem desacoplados, testáveis e de fácil manutenção.

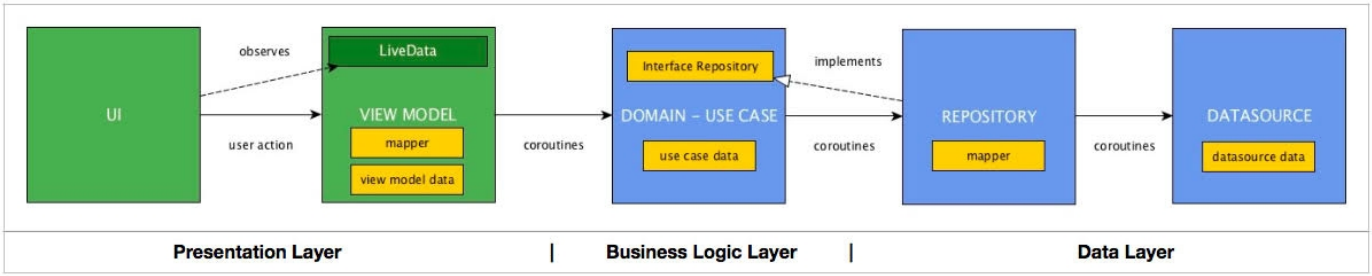
O que é a Clean Architecture? A Clean Architecture consiste em um diagrama de camadas, em que cada um dos seus componentes possuem suas próprias responsabilidades e cada uma delas tem conhecimento apenas de camadas de dentro, ou seja, a camada de “Frameworks e Drivers” enxerga somente a de “Interface Adapters”.



Vantagens: O código é facilmente testável, se comparado a arquitetura MVVM simples; Componentes ainda mais desacoplados, a estrutura do pacote é facilmente de se navegar entre eles; Novas funcionalidades podem ser adicionadas rapidamente pelo time de desenvolvedores.

Desvantagens: Curva de aprendizado relativamente íngreme, considerando que todas as camadas funcionam juntas, exigindo um certo tempo para entender seus conceitos, principalmente desenvolvedores provenientes de padrões como MVVM e MVP simples; Pela arquitetura exigir o

acréscimo de muitas classes adicionais, este modelo não é ideal para projetos de baixa complexidade. A intenção é de demonstrar as regras de dependência dentro da arquitetura.



Cada camada de MVVM usando Clean Architecture no Android e os códigos se dividem em três camadas:

Camada de Apresentação (Presentation Layer):

Nesta camada, são incluídas as “Activity”s, “Fragment”s como sendo as “Views”, e as “ViewModel”s, devem ser mais simples e intuitivo possível e ainda, não devem ser incluídas regras de negócio nas “Activity”s e “Fragment”s.

Uma “View” irá se comunicar com sua respectiva “ViewModel”, e assim, a “ViewModel” se comunicará com a camada de domínio para que sejam executadas determinadas ações. Uma “ViewModel” nunca se comunicará diretamente com a camada de dados;

Camada de Domínio (Domain Layer):

Na camada de domínio, devem conter todos os casos de uso da aplicação. Os casos de uso tem como objetivo serem mediadores entre suas “ViewModel”s e os “Repository”s.

Caso for necessário adicionar uma nova funcionalidade, tudo o que deve ser feito é adicionar um novo caso de uso e todo seu respectivo código estará completamente separado e desacoplado dos outros casos de uso. A criação de um novo caso de uso é justamente para evitar que ao adicionar novas funcionalidades, quebrar as preexistentes no código;

Camada de Dados (Data Layer):

Esta camada possui todos os repositórios que a camada de domínio tem disponíveis para utilização e “DataSource”s, que são responsáveis por decidir qual a fonte em que devem ser recuperados os dados, sendo de uma base de dados local ou servidor remoto.

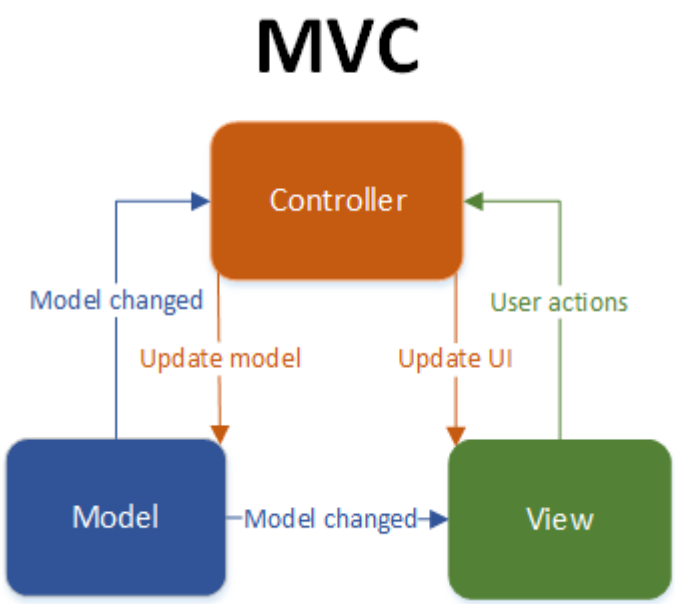
Arquitetura Android mvc x mvp x mvvm

26-10-2020 por ederson

Fixando o conhecimento basico em arquiteura Android, vou descrever os Patterns de arquitetura já utilizados:

- MVC - Model View Controller
- MVP - Model View Presenter
- MVVM - Model View ViewModel

MVC



Esta abordagem separa sua aplicação em um nível macro com 3 conjuntos de responsabilidades. Sendo eles:

Model No Model irá conter Data + State + Business Logic, de forma não técnica. Podemos usar como exemplo lógica comercial, acesso a dados e regra de negócios, que não está ligado a View ou Controller e com isto se torna muito reutilizável.

View Representa o Model, ela é a UI (user interface) e faz a comunicação com a Controller sempre que ocorre uma interação do usuário. Quanto menos eles souberem do que deve ser feito com a interação, mais flexíveis serão para mudar.

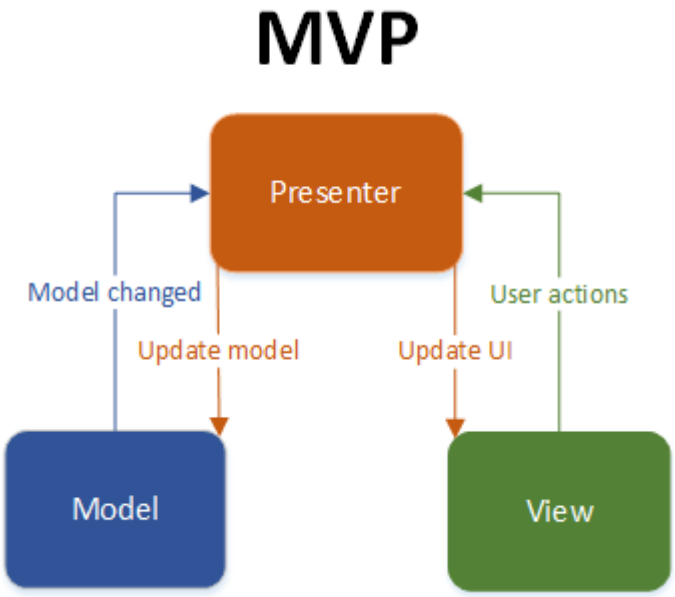
Controller Ele é o responsável pelo que acontece no aplicativo. Quando a View diz para o Controller que um usuário clicou em um botão, ele decide como interagir. Se ocorrer modificação de dados no Model, o Controller pode decidir atualizar o estado de exibição, quase sempre representado por uma activity ou fragment.

Desvantagens:

Testabilidade - O controlador está tão ligado às APIs do Android que é difícil testar a unidade. Modularidade e flexibilidade - Os Controllers estão bem acoplados às Views. Pode também ser uma

extensão da View. Caso você mude a View, devemos voltar e mudar o Controller. Manutenção - Pode ocorrer de o código começar a ser transferido para os Controllers, tornando-os cheios, complexos e com grande facilidade de crashs.

MVP



O MVP quebra o Controller de modo que o acoplamento de View/Activity pode ocorrer sem amarrá-lo ao restante das responsabilidades do “Controller”. Veremos mais sobre isso abaixo, mas comecemos novamente com uma definição comum de responsabilidades, em comparação com o MVC.

Model Igual ao MVC / Nenhuma mudança

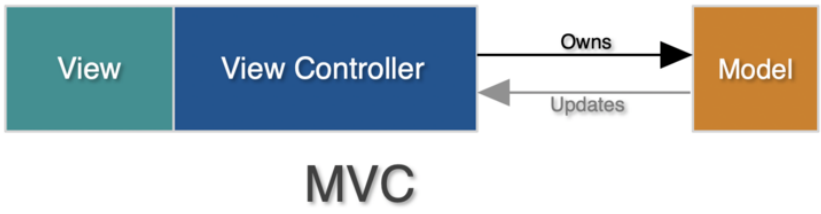
View A única alteração aqui é que a Activity/Fragment agora é considerada parte da View. A boa prática é ter uma Activity implementando uma interface de exibição para que o Presenter tenha uma interface para codificar. Com isto é eliminado o acoplamento e permite testes unitários.

Presenter Este é o Controller do MVC, exceto por ele não estar vinculado ao View, apenas a uma interface, com isto ele não mais gerencia o tráfego de solicitações recebidas, como é feito no Controller.

Desvantagens:

Manutenção - Os Presenters, assim como os Controllers, são propensos a colecionar lógica comercial adicional, espalhados com o tempo. Podemos nos deparar com grandes Presenters difíceis de separar.

MVVM



MVC

MVVM

MVVM com o Data Binding tem como benefícios testes mais fáceis e modularidade, ao mesmo tempo que reduz a quantidade de código que temos que escrever para conectar o Model com a View. Este Pattern suporta ligação bidirecional entre View e ViewModel, com isto nos permite ter propagação automática de mudanças. Bem, vou explicar.

Model Igual ao MVC e MVP / Nenhuma mudança

View A View liga-se a variáveis Observable e ações expostas pelo ViewModel de forma flexível.

ViewModel O ViewModel é responsável por expor métodos, comandos e propriedades que mantém o estado da View, assim como manipular a Model com resultados de ações da View e preparar dados Observable necessários para a exibição.

Desvantagens:

Manutenção - As Views podem se ligar (bind) a ambas as variáveis e expressões, adicionando código efetivamente ao XML. O ideal é obter valores diretamente do ViewModel em vez de tentar calcular utilizando no XML.