FIAP ON

TÓPICOS AVANÇADOS DE DESENVOLVIMENTO ANDROID

PADRÕES DE PROJETO ANDROID

RICARDO DA SILVA OGLIARI

3

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

3 PADRÕES DE PROJETO ANDROID

Quando uma plataforma atinge um nível de penetração considerável no mercado, é comum que seus projetos também se tornem maiores e mais complexos. Consequentemente, a demanda por código limpo, escalável e de fácil manutenção torna-se extremamente necessária. Caso contrário, os projetos seriam limitantes por seu tamanho, o que é totalmente contrassenso, visando sucesso e crescimento de projetos, aplicativos e softwares como um todo.

Diferentemente do que víamos em seus primórdios, hoje em dia, há bastante discussão nos times técnicos sobre qual arquitetura de projetos é a mais adequada, sobre qual padrão deve orquestrar o projeto e assim por diante.

Atualmente existem diversas abordagens, dentre as mais comuns podemos listar o MVC (Model View Controller), MVP (Model View Presenter), MVVM (Model View-View Model), MVI (Model View Intent) e Vipher.

Dentre os padrões com mais relevância no mercado, separamos os padrões MVC, MVP e MVVM.

3.1 O padrão MVC

Para desenvolvedores Android mais experientes, é muito comum vermos a história de programação começar pelo MVC, avançar para o MVP e estar, atualmente, no MVVM. Isso porque o MVC é a primeira escolha devido a sua simplicidade e baixa curva de aprendizado. Além disso, o MVC é um padrão de arquitetura adotado em massa para outros ambientes e plataformas, como Web por exemplo.

Na Figura Arquitetura MVC temos uma representação simplificada desse padrão. Ele é dividido em três partes principais:

- **View**: visualização do conteúdo. Na nossa aplicação são as telas, representadas pelas Views (XML´s ou Java/Kotlin).
- **Model**: modelo de dados. Podem ser os famosos Java Beans, POJOS (Plain Old Java Objects) ou Data Classes do Kotlin. Normalmente espelham os dados que estão salvos em bancos de dados local ou externo.

- **Controller**: faz a orquestração entre View e Model. Ou seja, percebe que um modelo foi alterado, a Visão precisa ser alertada e o conteúdo visual daquela tela também deve ser atualizado. É comum vermos uma confusão de imagens representando MVC, sendo que algumas delas a *View* e *Model* só possuem ligação com *Controller*. Em outros casos, como a Figura "Arquitetura MVC", existe também uma ligação entre *Model* e *View*.

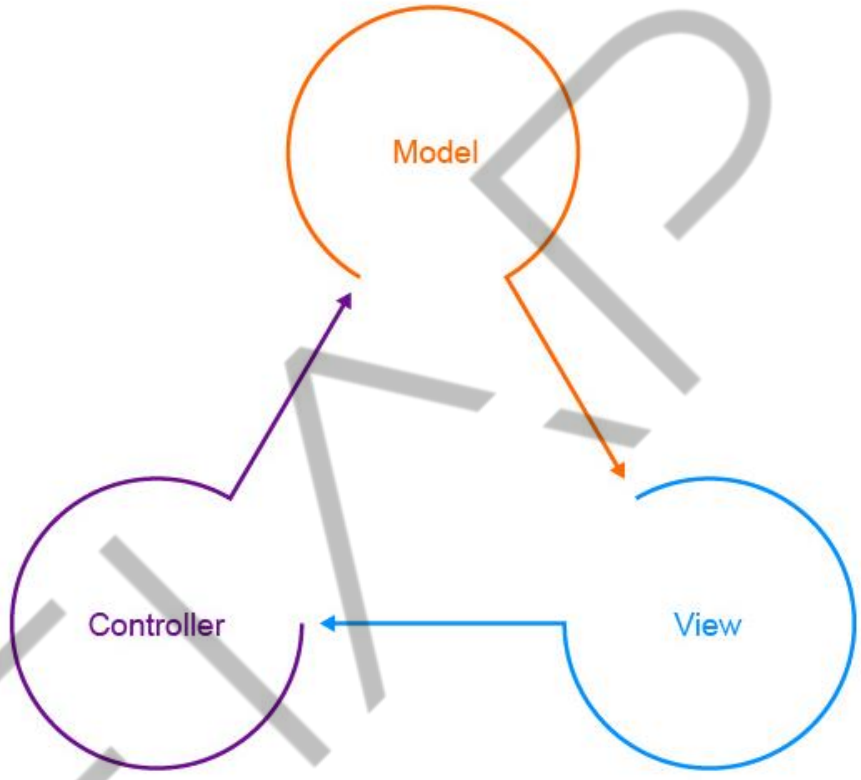

Figura 3.1 - Arquitetura MVC
Fonte: Macoratti (2015)

É importante ressaltar também que às vezes não será possível seguir exaustivamente um padrão, sendo necessárias algumas pequenas alterações para se adaptar totalmente ao seu projeto. Vale ressaltar que não existe uma arquitetura que será a ideal em todos os tipos possíveis de projetos (silver bullet). Isso vale para MVC, MVVM e MVP.

3.2 MVP

O MVP é muito parecido com o MVC, nos conceitos. Sua maior diferença é na implementação propriamente dita, em que vamos usar uma interface como um contrato das alterações que uma View pode ter.

Mas, antes disso, vamos ao conceito.

Da mesma forma, também temos três partes principais:

- **View**: conceito muito parecido com o MVC. Porém, no MVP, ele entende que a Activity, os Fragments e as Views fazem parte da camada de visão. A Activity não é mais entendida como o Controller.
- **Model**: o mesmo intuito do Model do MVC.
- **Presenter**: uma camada que define um contrato sobre as alterações e funções que uma visão pode ter. Por exemplo, em uma tela de login poderia ser a verificação dos campos (se não estão em branco) e a resposta da chamada ao serviço de login (sucesso ou falha).

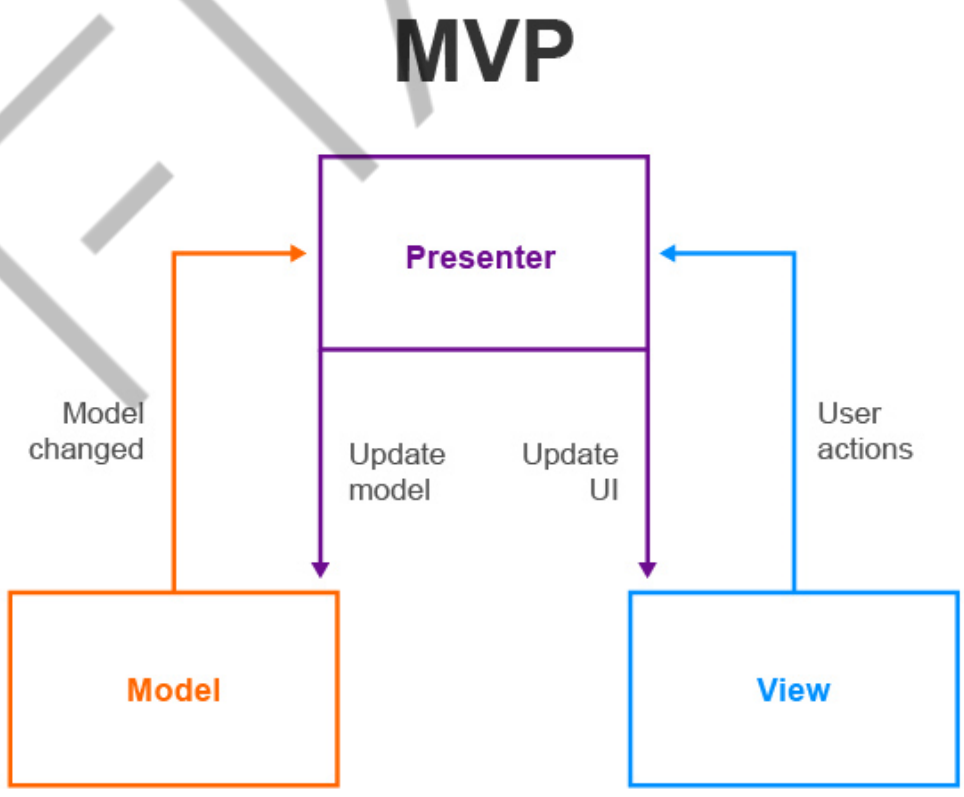

Figura 3.2 - Arquitetura Model View Presenter
Fonte: PIRES (2019)

3.3 Padrão MVVM

O MVVM também é parecido, conceitualmente, com o MVC e MVP. Perceba que a camada de ViewModel ocupa o lugar do Controller ou do Presenter nas outras arquiteturas. O ViewModel se comunica com os modelos de dados para fornecê-los às visualizações, além de ouvir possíveis eventos gerados por elas.

Um ponto crucial neste modelo, que não chega a ser uma camada, mas tem importância, é o Binder, que é justamente essa ligação entre o View e o ViewModel.

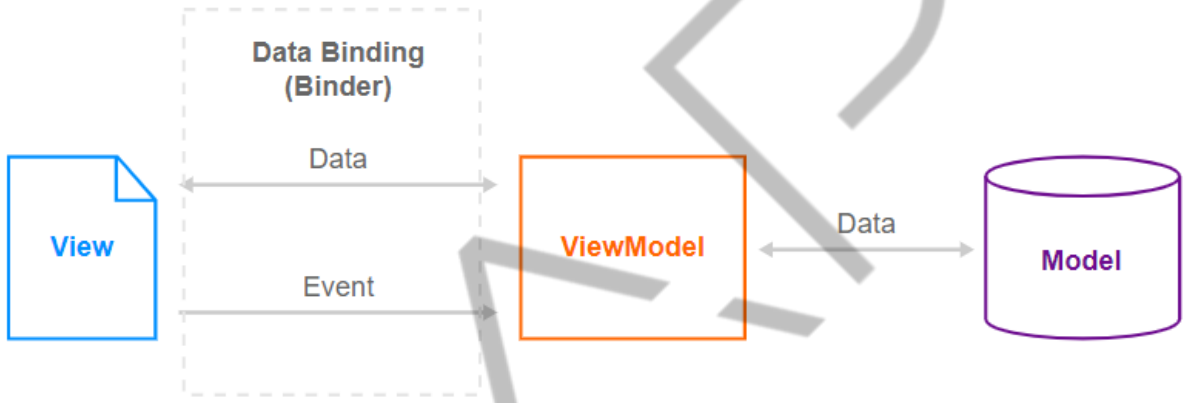

Figura 3.3 - Arquitetura Model View-View Model
Fonte: ZK (2020)

3.4 JetPack

Nos últimos anos, o Android entregou um conjunto de bibliotecas agrupadas em um conjunto chamado JetPack. Dentre esse conjunto, repleto de opções, temos os componentes arquiteturais que oferecem classes e interfaces que facilitam o uso do MVVM no Android.

3.5 A Aplicação

Para aplicarmos os conceitos das arquiteturas de forma prática, visando melhor compreensão, construiremos uma aplicação de lista de compras e limite de gastos.

Veja abaixo as duas únicas telas que a aplicação terá:

Figura 3.4 - Tela de definição de limite
Fonte: Elaborado pelo autor (2020)

Figura 3.5 - Tela de listagem de itens
Fonte: Elaborado pelo autor (2020)

A aplicação, Figura Tela de definição de limite, apresenta uma tela inicial de definição de limite no valor da compra. Na sequência, Figura Tela de listagem de itens, temos a listagem dos itens, podendo ser definidos a quantidade e o valor, e chegar ao preço final da compra.

3.6 Codificando em MVC

Inicialmente vamos codificar essa aplicação seguindo o padrão MVC.

O primeiro passo é criar uma aplicação Android padrão com o nome de ControleCompras*.* Já defina uma *EmptyActivity* como a primeira *activity* criada. Por fim, defina essa mesma tela inicial com o nome de *DefineLimitActivity.*

3.7 Definindo os drawables

Edite o arquivo *styles.xml* que está no caminho *res -> values* com o conteúdo do código-fonte Conteúdo do arquivo styles.xml:

```
<resources>
   <style name="AppTheme" parent="Theme.MaterialComponents.Light">
       <item name="colorPrimary">@color/colorPrimary</item>
       <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
       <item name="colorAccent">@color/colorAccent</item>
   </style>

</resources>
```

Código-fonte 3.1 – Conteúdo do arquivo styles.xml
Fonte: Elaborado pelo autor (2020)

Estamos criando um tema a partir de seu pai, o *Theme.MaterialComponents.Light.* Definimos os itens de *colorPrimary, colorPrimaryDark* e *colorAccent.* Todas as cores estão no arquivo *res -> values -> colors.xml.* Seu conteúdo é mostrado no código-fonte Conteúdo do arquivo colors.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <color name="colorPrimary">#32CD32</color>
   <color name="colorPrimaryDark">#32CD32</color>
   <color name="colorAccent">#32CD32</color>
</resources>
```

Código-fonte 3.2 – Conteúdo do arquivo colors.xml
Fonte: Elaborado pelo autor (2020)

Também temos alguns recursos de texto que são ilustrados no código-fonte Conteúdo do arquivo strings.xml. O arquivo alvo é o *res -> values -> strings.xml.*

```
<resources>
   <string name="app_name">Minhas Compras</string>
   <string name="show_status">Máximo %1$s. Total %2$s</string>
</resources>
```

Código-fonte 3.3 – Conteúdo do arquivo strings.xml
Fonte: Elaborado pelo autor (2020)

A imagem que fica na tela de definição de limite foi baixada da internet e editada para remover espaços em branco que existiam nas laterais da figura. O link da imagem original é <>https://rsaude.com.br/img/noticias/g/o-poder-dos-vegetais-voce-conhece-os-compostos-bioativos-28092016144520.png.

3.8 Definindo os layouts

Independentemente da arquitetura, teremos três arquivos de layouts. No MVC eles farão parte da camada de View. No código-fonte Conteúdo do arquivo activity_main.xml temos o XML, que define a primeira tela do limite de compra.

```
      <resources>
      <?xml version="1.0" encoding="utf-8"?>
      <androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
         xmlns:app="http://schemas.android.com/apk/res-auto"
         xmlns:tools="http://schemas.android.com/tools"
         android:layout_width="match_parent"
         android:padding="20dp"
         android:layout_height="match_parent"
         tools:context=".DefineLimitActivity">

         <LinearLayout
             android:orientation="vertical"
             app:layout_constraintTop_toTopOf="parent"
             app:layout_constraintBottom_toBottomOf="parent"
             android:layout_width="match_parent"
             android:layout_height="wrap_content">
             <androidx.appcompat.widget.AppCompatImageView
                 android:layout_gravity="center_horizontal"
                 android:src="@drawable/frutas"
                 android:layout_width="240dp"
                 android:layout_marginBottom="20dp"
                 android:layout_height="240dp"/>
             <com.google.android.material.textfield.TextInputLayout
                 android:id="@+id/txtValue"
                 android:layout_width="match_parent"
                 android:layout_height="wrap_content"
                 app:layout_constraintBottom_toBottomOf="parent"
                 app:layout_constraintLeft_toLeftOf="parent"
                 app:layout_constraintRight_toRightOf="parent"
                 app:layout_constraintTop_toTopOf="parent"
                 android:hint="Valor Máximo">

                 <com.google.android.material.textfield.TextInputEditText
                     android:id="@+id/edtValue"
                     android:inputType="numberDecimal"
                     android:layout_width="match_parent"
                     android:layout_height="wrap_content"/>

             </com.google.android.material.textfield.TextInputLayout>

             <androidx.appcompat.widget.AppCompatButton
```

```
                android:layout_width="wrap_content"
                android:text="Iniciar"
                android:id="@+id/btnGo"
                android:layout_gravity="right"
                android:layout_height="wrap_content"/>
        </LinearLayout>

    </androidx.constraintlayout.widget.ConstraintLayout>
```

Código-fonte 3.4 – Conteúdo do arquivo activity_main.xml
Fonte: Elaborado pelo autor (2020)

Fazemos uso do *ConstraintsLayouts* para alinhar um *LinearLayout* no centro vertical da tela. No container desses últimos, temos: um *AppCompatImageView,* um *TextInputLayout* e um *AppCompatButton.*

O segundo arquivo de layout que iremos estudar é o *tem_list.xml.* Como teremos um *RecyclerView* na segunda tela, precisamos definir o layout de cada item que irá compor a lista.

```
    <?xml version="1.0" encoding="utf-8"?>
    <LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:gravity="center_vertical"
        android:layout_height="wrap_content">

        <androidx.appcompat.widget.AppCompatTextView
            android:id="@+id/txtTotalItem"
            android:layout_width="wrap_content"
            android:text="R$ 0,00"
            android:layout_marginRight="10dp"
            android:textSize="14sp"
            android:textStyle="bold"
            android:layout_height="wrap_content"/>

        <androidx.appcompat.widget.AppCompatTextView
            android:id="@+id/txtItemLabel"
            android:layout_width="0dp"
            android:text="Definição do item..."
            android:layout_weight="1"
            android:textSize="14sp"
            android:layout_height="wrap_content"/>

        <com.google.android.material.textfield.TextInputEditText
            android:hint="Qtd"
            android:textSize="14sp"
            android:id="@+id/edtQtd"
            android:inputType="numberDecimal"
            android:layout_width="50dp"
            android:layout_marginRight="10dp"
            android:layout_height="wrap_content"/>


        <com.google.android.material.textfield.TextInputEditText
            android:textSize="14sp"
            android:inputType="numberDecimal"
```

```
            android:hint="Valor"
            android:id="@+id/edtValue"
            android:layout_width="50dp"
            android:layout_height="wrap_content"/>

    </LinearLayout>
```

Código-fonte 3.5 – Conteúdo do arquivo item_list.xml
Fonte: Elaborado pelo autor (2020)

O item foi criado com um *LinearLayout* horizontal, como *ViewGroup.* Em seguida, temos dois *AppCompatTextView* que mostrarão o valor total daquele item, bem como seu identificador textual, respectivamente. Na sequência, temos então dois *TextInputEditText* para o usuário definir o valor e quantidade daquele item.

Para finalizar os layouts, visualize o código-fonte Conteúdo do arquivo activity_list_itens.xml com o conteúdo do arquivo *activity_list_itens.xml.*

```
    <?xml version="1.0" encoding="utf-8"?>
    <androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".ListItensActivity">

        <androidx.recyclerview.widget.RecyclerView
            android:padding="12dp"
            android:id="@+id/recycler"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            app:layout_constraintTop_toTopOf="parent"
            app:layout_constraintBottom_toTopOf="@id/lineSeparator"/>

        <View
            android:id="@+id/lineSeparator"
            app:layout_constraintTop_toBottomOf="@id/recycler"
            app:layout_constraintBottom_toTopOf="@id/txtStatus"
            android:background="@color/colorAccent"
            android:layout_width="match_parent"
            android:layout_height="1dp"/>

        <TextView
            app:layout_constraintTop_toBottomOf="@id/lineSeparator"
            app:layout_constraintBottom_toTopOf="@id/txtItem"
            android:id="@+id/txtStatus"
            android:gravity="center"
            android:text="@string/show_status"
            android:layout_width="match_parent"
            android:layout_height="40dp"/>

        <com.google.android.material.textfield.TextInputLayout
            android:id="@+id/txtItem"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            app:layout_constraintBottom_toBottomOf="parent"
```

```
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent"
            app:layout_constraintTop_toBottomOf="@id/txtStatus"
            android:hint="Insira o item">

            <com.google.android.material.textfield.TextInputEditText
                android:id="@+id/edtNewItem"
                android:lines="1"
                android:imeOptions="actionSend"
                android:maxLines="1"
                android:singleLine="true"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"/>

        </com.google.android.material.textfield.TextInputLayout>

    </androidx.constraintlayout.widget.ConstraintLayout>
```

Código-fonte 3.6 – Conteúdo do arquivo activity_list_itens.xml
Fonte: Elaborado pelo autor (2020)

Novamente usamos um *ConstraintLayout* para organizar os elementos filhos, um *RecyclerView* e depois um *View* como linha separadora. Na sequência, temos um *TextView* que vai mostrar o status da compra, o limite e o valor corrente. E finalmente um *TextInputLayout,* para entrada de texto e inserção de um novo item.

3.9 Trabalhando com a camada Model

Pensando no MVC, o passo anterior foi realizado para definir a camada de visualização (View). Agora vamos focar na camada de modelo (Model). Antes da codificação, peço a você que crie a seguinte estrutura de pastas e arquivos em seu projeto: veja a Figura Estrutura do projeto com MVC:

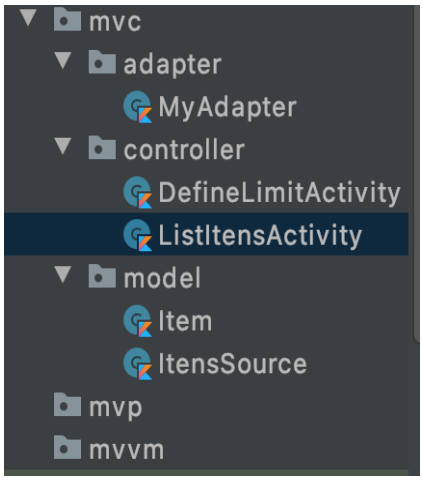

Figura 3.6 - Estrutura do projeto com MVC
Fonte: Elaborado pelo autor (2020)

O arquivo *DefineLimitActivity* já deve estar presente no seu projeto, pois foi criado automaticamente pela própria IDE.

Agora vamos especificamente para a pasta *model* e suas duas classes. O *Item* é apenas um *data class* do Kotlin que espelha os dados do modelo, que poderia estar salvo em um banco de dados local ou externo. Veja seu conteúdo no código-fonte Conteúdo do arquivo Item.kt:

```
data class Item (
   val label: String,
   var total: Double
)
```

Código-fonte 3.7 – Conteúdo do arquivo Item.kt
Fonte: Elaborado pelo autor (2020)

A classe *ItensSource* serve para indicar uma possível integração com uma fonte dos dados referentes ao modelo, podendo ser um banco ou até mesmo uma integração com serviços externos, como: Firebase, Hasura ou algo semelhante. Veja o conteúdo dessa classe no código-fonte Conteúdo do arquivo ItensSource.kt:

```
class ItensSource {

   val myDataset = mutableListOf<Item>()

}
```

Código-fonte 3.8 – Conteúdo do arquivo ItensSource.kt
Fonte: Elaborado pelo autor (2020)

3.10 Camada de Controller

No Android, a camada de Controller do MVC são, basicamente, as classes *Activities* e *Fragments.*

Vamos começar os estudos com o conteúdo da classe *DefineLimitActivity.* Lembrando que essa classe se refere à primeira tela da aplicação, em que definimos o limite da nossa compra. Veja o conteúdo dessa classe no código-fonte Conteúdo do arquivo DefineLimitActivity.kt:

Aqui encontramos alguns códigos padrões para uma tela, como:

- Herança de uma classe Activity, ou filha dela. Nesse caso, herdamos de *AppCompatActivity.*

- Sobrescrita do método *onCreate* que faz parte do ciclo de vida de uma tela. O ciclo de vida é de suma importância, mas, como vamos trabalhar diretamente com ele no MVVM, falaremos mais sobre esse item nas seções posteriores a esta.

Como foi usado o *kotlin synthetic*, não precisamos do famigerado *findViewById.* Basta acessar o elemento visual pelo seu id como se estivéssemos acessando uma simples variável ou constante. É o que estamos fazendo no *btnGo*, definindo o *listener* para seu evento de clique.

Para responder ao evento de clique, fornecemos um bloco de código para o setOnClickListener. Nele, criamos uma instância de Intent, definindo os parâmetros de contexto, como a tela corrente, e de direcionamento, como o class.java da tela de listagem de itens. Antes de chamar efetivamente a tela, com a chamada startActivity, verificamos se existe um valor inserido pelo usuário.

```
class DefineLimitActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContentView(R.layout.activity_main)

       btnGo.setOnClickListener {

           val intent = Intent(
               this@DefineLimitActivity,
               ListItensActivity::class.java
           )

           if (edtValue.text.toString().trim().length > 0) {
               intent.putExtra(
                     "limit",
                     edtValue.text.toString().toDouble())
               startActivity(intent)
           } else {
               edtValue.setError("Insira um valor válido para seguir!")
           }
       }
   }

}
```

Código-fonte 3.9 – Conteúdo do arquivo DefineLimitActivity.kt
Fonte: Elaborado pelo autor (2020)

Talvez você já tenha percebido um problema nessa camada de *Controller,* mas vale frisar: a exibição de mensagens de erro deveria ser parte da visualização, afinal, elas são apresentadas na tela para o usuário. Sendo assim, já estamos misturando as camadas de controle e visualização.

Outro problema pode ser imaginado, pois estamos trabalhando com uma classe muito simples, com uma lógica simplória. Vamos agora imaginar aplicações mais complexas; nesse caso, nossa Activity teria muito mais código e estaria interdependente a vários elementos visuais da interface gráfica, indo contra a recomendação da orientação a objetos na qual precisamos criar classes altamente coesas e com baixo acoplamento.

Na comunidade Android, foi criado até um conceito de arquitetura God Activity, ou ainda, GAA (God Activity Architecture). Apesar da pitada de ironia, o conceito implícito é importante: evitar que a classe de Activity tenha todo o controle, sendo um Deus em seu projeto e, portanto, dificultando em demasia o controle da aplicação, a escalabilidade e os testes.

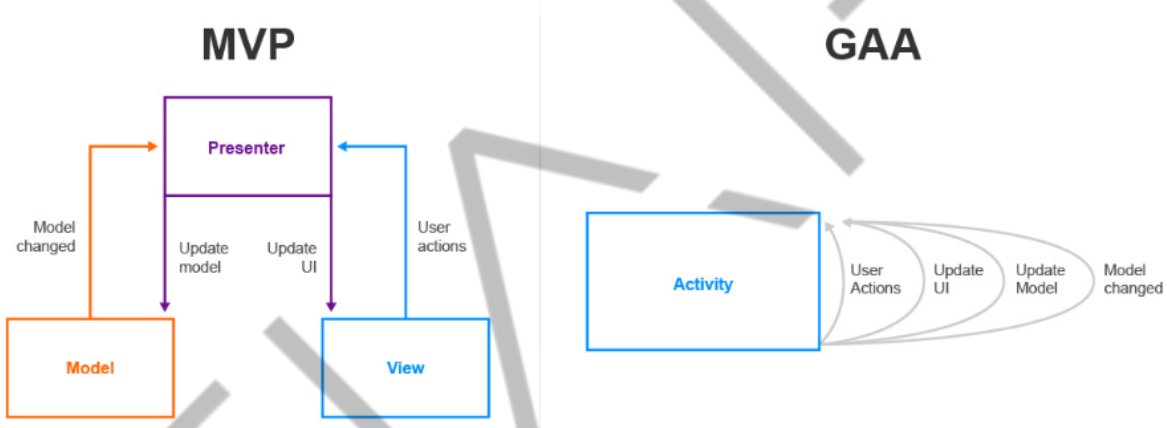

Figura 3.7 - God Activity Architecture
Fonte: CASE (2019)

Vamos olhar o conteúdo da classe *ListItensAcitivity,* ilustrado no código-fonte Conteúdo do arquivo ListItensActivity.kt.

A classe possui quatro variáveis. As duas primeiras (*LayoutManager* e *MyAdapter*) serão usadas pelo *RecyclerView.* A terceira variável é o *limit,* que será recebido via *intent extras*. E, por fim, uma instância de *NumberFormat* para apresentar o valor monetário da forma correta (R$25,00, por exemplo).

Dentro do *onCreate,* é definido o conteúdo visual da tela e recuperamos o valor de limite, oriundo da tela anterior. Usando o *?.let* do Kotlin, para evitar o *nullPointerException,* verificamos que realmente recebemos um limite. Se isso ocorrer, formatamos o valor de limite e configuramos o texto do elemento visual com id *txtStatus.* Esse *TextView* aparece logo acima do campo de entrada de item, na parte inferior da tela. Perceba também que o valor da compra ainda é R$0,00.

Na sequência, ainda usando o *synthetics,* definimos o listener do evento de teclado (*KeyListener*) passando, como parâmetro, o *this*; ou seja, a própria classe em que essa linha está inserida. Isso é possível porque a classe implementa a interface *View.OnKeyListener* e sobrescreve o método *onKey.*

No *onKey,* por sua vez, verificamos se a tecla digitada é de envio ou de *Enter,* o que nos indica que o usuário quer inserir um novo item na lista. Sendo assim, criamos uma nova instância da classe *Item,* enviando-a como parâmetro para o método *add* do *viewAdapter.* O ViewAdapter é o adaptador do *RecyclerView,* que iremos estudar na sequência.

Voltando ao *onCreate,* temos a instância de *viewManager* e *viewAdapter.* Nesse último, criamos uma instância de *MyAdapter*, já passando um método por parâmetro, que é exigido na instância dessa classe. Esse retorno indica que um item teve sua quantidade ou preço alterados, logo precisamos mudar o valor do total da compra no *TextView* de status.

Perceba que os problemas do MVC se repetem:

- Controles de lógica relacionados à visão, presentes no controlador.
- Acoplamento alto entre visão, controller e lógica de negócio.

```
class ListItensActivity : AppCompatActivity(), View.OnKeyListener {

   //quatro variáveis
   private lateinit var viewManager: RecyclerView.LayoutManager
   private lateinit var viewAdapter: MyAdapter

   private var limit: Double? = null
   private val format: NumberFormat =
NumberFormat.getCurrencyInstance(Locale.getDefault())

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContentView(R.layout.activity_list_itens)

       //recupera o limite da tela anterior e já mostra esse
       //valor no status
       limit = intent.getDoubleExtra("limit", 0.0)
       limit?.let {
           val totalStr: String = format.format(it)
           txtStatus.setText(getString(R.string.show_status, totalStr,
"R$0,00"))
       }

       edtNewItem.setOnKeyListener(this@ListItensActivity)

       viewManager = LinearLayoutManager(this)
       viewAdapter =
```

```
            MyAdapter() {
                limit?.let { safeLimit ->
                    val totalStr: String = format.format(safeLimit)
                    val parcialStr: String = format.format(it)
                    txtStatus.setText(
                        getString(
                            R.string.show_status,
                            totalStr,
                            parcialStr
                        )
                    )
                }
            }

        recycler.apply {
            setHasFixedSize(true)
            layoutManager = viewManager
            adapter = viewAdapter
        }
    }

    //listener chama esse método quando o campo de entrada de item
    //é editado
    override fun onKey(view: View?, keyCode: Int, event: KeyEvent?):
Boolean {
        event?.let {
            if ((it.getAction() == KeyEvent.ACTION_DOWN) &&
                (keyCode == KeyEvent.KEYCODE_ENTER)) {
                viewAdapter.add(
                    Item(
                        label = edtNewItem.text.toString(),
                        total = 0.0
                    )
                )
                edtNewItem.setText("")
                return true;
            }
        }

        return false
    }

}
```

Código-fonte 3.10 – Conteúdo do arquivo ListItensActivity.kt
Fonte: Elaborado pelo autor (2020)

Usamos *RecyclerView* para representar a lista dos itens de compra. Sendo assim, também precisamos do adaptador que está na codificação da classe *MyAdapter.* O conteúdo dela é mostrado no código-fonte Conteúdo do arquivo MyAdapter.kt:

No construtor da classe, estamos recebendo uma função que será acionada quando a soma total da compra for atualizada.

Logo no início, criamos uma instância de *ItensSource*, discutida anteriormente. Importante ressaltar que, em projetos profissionais, essa dependência poderia ser feita de uma maneira melhor; usando uma biblioteca específica, como: o *Dagger,* o *Koin* ou o *Hilt*.

A classe de *ViewHolder* tem bastante regra de negócio (ainda veremos esse assunto). Os métodos seguintes são apenas para cumprir o contrato definido entre a herança dessa classe e a da *RecyclerView.Adapter.*

No *onCreateViewHolder,* apenas inflamos o arquivo de layout que será usado em cada item, além de passar essa instância de *View* para instanciar nosso *ViewHolder* interno. Como essa classe também possui a passagem de uma função nos parâmetros de seu construtor, já definimos a função interna anônima. Quando acontece esse retorno, acionamos o *returnSum* indicando o novo valor total da compra naquele momento.

No *onBindViewHolder* é feita a ligação entre cada posição do adaptador com seu *ViewHolder* correspondente. Para tanto, chamamos o método *bind,* que ainda estudaremos. Também criamos um ouvidor para o evento de clique longo em um item. Esse evento fará com que aquele item seja removido da lista e, consequentemente, a soma do valor total da compra seja refeita.

Outro método que merece destaque é o *add.* Ele recebe uma instância de *Item* e o adiciona no dataset da nossa fonte de dados; em seguida, avisamos ao adaptador que um novo item foi adicionado e ele precisa tomar as medidas de atualizações necessárias para que a interface corresponda ao novo estado de uso.

```
class MyAdapter(
   val returnSum: (Double) -> Unit
) : RecyclerView.Adapter<MyAdapter.MyViewHolder>() {

   val dataSource = ItensSource()

   class MyViewHolder(
       view: View,
       private val makeCalNow: () -> Unit) :
RecyclerView.ViewHolder(view) {

       ...

   }

   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
MyViewHolder {
       val view = LayoutInflater
```

```
                        .from(parent.context)
                        .inflate(R.layout.item_list, parent, false)
        return MyViewHolder(
            view,
            {
                returnSum(
                    dataSource.myDataset.sumByDouble {
                       item -> item.total
                    }
                )
            })
    }

    override fun onBindViewHolder(holder: MyViewHolder, position: Int) {
        holder.bind(position, dataSource.myDataset[position])
        holder.labelTextView.setOnLongClickListener {
            dataSource.myDataset.removeAt(position)
            notifyItemRemoved(position)
            returnSum(dataSource.myDataset.sumByDouble { item ->
item.total })
            true
        }
    }

    override fun getItemViewType(position: Int): Int {
        return position
    }

    override fun getItemId(position: Int): Long {
        return position.toLong()
    }

    override fun getItemCount() = dataSource.myDataset.size

    fun add(item: Item) {
        dataSource.myDataset.add(item)
        notifyItemInserted(dataSource.myDataset.size - 1)
    }
}
```

Código-fonte 3.11 – Conteúdo do arquivo MyAdapter.kt
Fonte: Elaborado pelo autor (2020)

Por fim, olhe o código-fonte Conteúdo da classe interna ViewHolder com o conteúdo da classe interna *ViewHolder* presente em *MyAdapter.* No construtor recebemos uma *View* e uma função como retornos para avisarmos que o item mudou o valor, por meio da edição de quantidade e/ou de preço.

A quantidade alta de variáveis é devido ao número de elementos visuais, assim como um inteiro que guarda sua posição na lista do *recycler,* uma instância de *Item* com o modelo de dado com que esse *ViewHolder* trabalha.

Temos ainda o *init block,* que será chamado logo após o construtor principal. Estamos adicionando *listeners* para ouvirmos mudanças de texto nos *inputs* de

quantidade e valor. Logo, quando isso acontece, nos dois casos, estamos fazendo uma chamada ao método *calcValue.*

No *calcValue* é feito o cálculo do valor do item, multiplicando-se a quantidade pelo valor. Com o novo valor editamos o *TextView* do elemento do índice da lista para mostrar o valor do total de um dos itens que, nesse caso, está sendo controlado pela nova instância de *ViewHolder.* No final desse método, chamamos o *makeCallNow* passado por parâmetro pelo construtor.

```
class MyViewHolder(
   view: View,
   private val makeCalNow: () -> Unit) : RecyclerView.ViewHolder(view) {

   val labelTextView: AppCompatTextView = view.txtItemLabel
   val edtQtd = view.edtQtd;
   val edtValue = view.edtValue;
   val txtTotalItem = view.txtTotalItem

   var position: Int? = null
   var item: Item? = null

   init {
       edtQtd.addTextChangedListener(object: TextWatcher {
           override fun beforeTextChanged(s: CharSequence?, start: Int,
count: Int, after: Int) {}
           override fun onTextChanged(s: CharSequence?, start: Int,
before: Int, count: Int) {
               calcValue()
           }
           override fun afterTextChanged(s: Editable?) {}
       })

       edtValue.addTextChangedListener(object: TextWatcher {
           override fun beforeTextChanged(s: CharSequence?, start: Int,
count: Int, after: Int) {}
           override fun onTextChanged(s: CharSequence?, start: Int,
before: Int, count: Int) {
               calcValue()
           }
           override fun afterTextChanged(s: Editable?) {}
       })
   }

   fun calcValue(){
       val qtd = edtQtd.text.toString()
       val value = edtValue.text.toString()

       var total: Double = 0.0;
       try {
           total = qtd.toDouble() * value.toDouble()
       } catch (exc: NumberFormatException){}

       txtTotalItem.text = "R$ ${total}"

       position?.let {
           item?.total = total
```

```
            makeCalNow()
        }

    }

    fun bind(pos: Int, i: Item){
        position = pos
        item = i
        labelTextView.text = i.label
    }

}
```

Código-fonte 3.12 – Conteúdo da classe interna ViewHolder
Fonte: Elaborado pelo autor (2020)

Perceba como nesses últimos códigos temos uma mistura grande entre funções ligadas às camadas de visualização e de controlador. Esse problema tende a se tornar ainda mais impactante conforme o projeto se torna maior e mais complexo.

Outro fator também importante é que, devido ao alto acoplamento, implementar uma alta cobertura de testes nesse projeto seria uma tarefa árdua.

3.11 Codificando em MVP

Começamos esta seção com uma boa notícia.

Grande parte do conhecimento adquirido e dos códigos feitos anteriormente serão reaproveitados aqui.

Veja como fica a nossa estrutura de pacotes e classes na Figura Estrutura de pastas para o MVP:

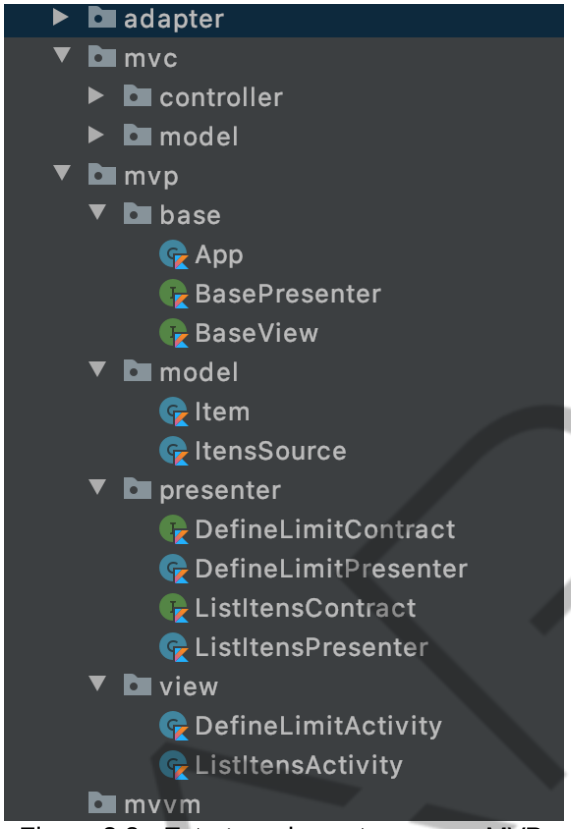

Figura 3.8 - Estrutura de pastas para o MVP.
Fonte: Elaborado pelo autor (2020)

A primeira mudança foi mover a pasta *adapter* para a raiz do projeto porque o adaptador será usado da mesma forma no MVC e MVP. Dentro da pasta *mvp,* que anteriormente estava vazia, criamos quatro pastas: *base, model, presenter* e *view.* Em cada uma dessas pastas, criamos novas classes.

Temos algumas considerações: as classes *Item* e *ItensSource*, que estão na pasta *model*, estão exatamente iguais àquelas da pasta *mvc*; e na pasta *view,* temos duas classes homônimas, as que estavam na pasta *mvc,* porém, vamos estudá-las ao longo desta seção, e veremos mudanças significativas.

Vamos diretamente à pasta *base* neste momento, dentro dela temos duas interfaces chaves: *BasePresenter* e *BaseView.* No código-fonte Conteúdo da interface BasePresenter apresentamos conteúdo do *BasePresenter.kt.*

A interface tem uma função *start.* Em algumas bibliografias é comum encontrarmos esse padrão, pois é comum os presenters precisarem inicializar alguma variável, ou qualquer outro estado, na sua criação; então o *start* é um bom lugar para isso.

```
interface BasePresenter {

   fun start()

}
```

Código-fonte 3.13 – Conteúdo da interface BasePresenter
Fonte: Elaborado pelo autor (2020)

No código-fonte Conteúdo da interface BaseView, temos o conteúdo da interface *BaseView.* Usamos o *generics* para definir o tipo da variável *presenter* que a interface terá. Ambas as classes de base serão usadas nos contratos entre as camadas de *Presenter* e *View.*

```
interface BaseView<T> {
   var presenter : T
}
```

Código-fonte 3.14 – Conteúdo da interface BaseView
Fonte: Elaborado pelo autor (2020)

A principal pasta que precisamos estudar é a *presenter,* essencial para o entendimento da arquitetura. Perceba que cada tela terá seu contrato e seu *presenter.* A razão disso é que no MVP o arquivo XML de layout e as Activities/ Fragments são parte da camada de *View.*

Quanto ao código-fonte Conteúdo da interface DefineLimitContract, temos nele o conteúdo da interface *DefineLimitContract.* Esse contrato define as duas interfaces para *View* e *Presenter,* baseado nas interfaces de base que criamos anteriormente.

A interface de *View* compreende o que deve ser implementado na *Activity* e, consequentemente, perceber que são ações que impactam visualmente a nossa tela. Essa é a ação de mostrar a mensagem de erro no campo de entrada de texto, avisando-nos que o valor de limite é inexistente ou inválido.

A interface de *Presenter* define o contrato de métodos que a classe de *presenter* precisará fornecer para sua implementação concreta. Perceba que aqui temos as lógicas de negócio que iremos codificar. Devemos checar se o campo digitado tem um valor válido ou não: se for afirmativo, vamos para a próxima tela e, se for negativo, precisamos enviar a mensagem de erro para a *View*.

```
interface DefineLimitContract {

    /**
     * Nossa Activity precisa implementar os métodos definidos abaixo
     */
    interface View : BaseView<DefineLimitPresenter> {
        fun showErrorInLimitValue()
    }

    /**
     * Nosso Presenter precisa implementar os seguintes métodos
     */
    interface Presenter : BasePresenter {
        fun limitIsValid(limit: String)
    }

}
```

Código-fonte 3.15 – Conteúdo da interface DefineLimitContract
Fonte: Elaborado pelo autor (2020)

A classe *DefineLimitPresenter* terá o conteúdo mostrado no código-fonte 3.16. Veja que o *start* não foi usado, mas como implementamos a interface *BasePresenter,* indiretamente é necessária a implementação concreta.

O método *limitIsValid* apenas tem a mesma lógica que havíamos codificado na Activity. Observe que estamos usando uma classe *App.* Veremos seu código, mas, por ora, basta saber que ela é uma forma de fornecer uma implementação de *Context,* sem precisar criar uma dependência forte entre *Context, Activity e Presenter.*

Mais um detalhe importante: no construtor dessa classe é passada a instância do *DefineLimitContract.View* e, como estamos falando de uma interface, receberemos sua implementação concreta, o que é necessário para enviar a mensagem de erro, caso o valor de limite não seja válido.

```
class DefineLimitPresenter(private val view : DefineLimitContract.View) :
DefineLimitContract.Presenter {

    override fun limitIsValid(limit: String) {
        if (limit.trim().length > 0) {
            App.context?.let { safeContext ->
                val intent = Intent(safeContext,
ListItensActivity::class.java)
                intent.putExtra("limit", limit.toDouble())
                safeContext.startActivity(intent)
            }

        } else {
            view.showErrorInLimitValue()
        }
    }

    override fun start() {
```

```
    }

}
```

Código-fonte 3.16 – Conteúdo da classe DefineLimitPresenter
Fonte: Elaborado pelo autor (2020)

O código-fonte Conteúdo da classe App nos apresenta o conteúdo da classe *App,* localizada na pasta *base.* No *onCreate,* que será chamado no início do ciclo de vida da aplicação, criamos a instância de *context,* deixada dentro do *company object,* tratada como estático e com acesso facilitado em qualquer classe do projeto.

```
class App : Application() {

    companion object {
        var context: Context? = null
    }

    override fun onCreate() {
        super.onCreate()
        context = getApplicationContext();
    }

}
```

Código-fonte 3.17 – Conteúdo da classe App.
Fonte: Elaborado pelo autor (2020)

E para fechar o ciclo da tela de definição de limite e suas camadas relacionadas, vamos estudar o código-fonte Conteúdo da classe DefineLimitActivity, em que apresenta o conteúdo da classe *DefineLimitActivity.* Lembrando que estamos falando da *activity* que está no pacote *mvp -> view.*

Inicialmente precisamos destacar a diferença de tamanho dessa classe e também como separamos completamente a lógica de visão e a lógica de negócio (que está no *presenter*). Quando definimos que essa classe implementa a interface *DefineLimitContract.View,* obrigatoriamente, precisamos fornecer a implementação concreta do método *showErrorInLimitValue.*

Também é necessária a criação de uma variável do tipo *DefineLimitPresenter.* Ela foi marcada com *lateinit* porque sua instância é criada somente no *onCreate.* Passamos como parâmetro um *this,* porque é necessário que o *presenter* conheça que é a sua visão relacionada.

Por fim, ao tratar o *listener* do evento de clique, enviamos o valor do campo de entrada de texto como parâmetro para o método *limitIsValid.*

```
class DefineLimitActivity : AppCompatActivity(), DefineLimitContract.View
{

   override fun showErrorInLimitValue() {
       edtValue.setError("Insira um valor válido para seguir!")
   }

   override lateinit var presenter : DefineLimitPresenter

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContentView(R.layout.activity_main)

       presenter = DefineLimitPresenter(this)

       btnGo.setOnClickListener {
           presenter.limitIsValid(edtValue.text.toString())
       }
   }

}
```

Código-fonte 3.18 – Conteúdo da classe DefineLimitActivity
Fonte: Elaborado pelo autor (2020)

Antes de partirmos para a implementação do MVP, na tela de listagens de itens, precisamos dar uma olhada no *AndroidManifest.xml,* ilustrado no código-fonte Conteúdo do arquivo AndroidManifest.xml. A primeira diferença é a propriedade *android:name* na tag *application.* Também é importante que as propriedades *android:name* nas tags *activities* apontem para a pasta *.mvp.view.*

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.estudo.android.controlecompras">

   <application
       android:name=".mvp.base.App"
       android:allowBackup="true"
       android:icon="@mipmap/ic_launcher"
       android:label="@string/app_name"
       android:roundIcon="@mipmap/ic_launcher_round"
       android:supportsRtl="true"
       android:theme="@style/AppTheme">
       <activity
           android:name=".mvp.view.ListItensActivity"
           android:screenOrientation="portrait"/>
       <activity
           android:name=".mvp.view.DefineLimitActivity"
           android:screenOrientation="portrait">
           <intent-filter>
               <action android:name="android.intent.action.MAIN" />

               <category android:name="android.intent.category.LAUNCHER"
/>
           </intent-filter>
       </activity>
   </application>
```

```
</manifest>
```

Código-fonte 3.19 – Conteúdo do arquivo AndroidManifest.xml.
Fonte: Elaborado pelo autor (2020)

Agora sim, podemos nos concentrar na tela de lista de itens.

Começaremos pela interface *ListItensContract,* mostrada no código-fonte 3.20. Lembrando que a *View* compreende ações que afetarão a parte visual da tela, ou seja, a alteração do status das compras em relação ao limite. E o *presenter* tem relação com a lógica de negócios que, neste caso, é a leitura de limite da tela anterior e os cálculos necessários, quando uma nova parcial de valor total da compra é lançada.

```
private var connectivityManager: ConnectivityManager? = null
interface ListItensContract {

   interface View : BaseView<ListItensPresenter> {
       fun showNewStatus(status: String)
   }

   interface Presenter : BasePresenter {
       fun readLimit(intent: Intent)
       fun newParcial(value: Double)
   }

}
```

Código-fonte 3.20 – Código da interface ListItensContract.
Fonte: Elaborado pelo autor (2020)

Já no *ListItensPresenter,* vamos encontrar justamente as implementações dos métodos definidos na interface. Veja que no código-fonte Novas variáveis para a MainActivity, grande parte desse código estava nas *Activities*, quando a arquitetura usada era a MVC.

```
class ListItensPresenter(private val view : ListItensContract.View) :
ListItensContract.Presenter {

   private var limit: Double? = null
   private val format: NumberFormat = NumberFormat.getCurrencyInstance(
     Locale.getDefault()
   )

   override fun readLimit(intent: Intent) {
       limit = intent.getDoubleExtra("limit", 0.0)

       limit?.let {
           App.context?.let {safeContext ->
               val totalStr: String = format.format(it)
               view.showNewStatus(
                  safeContext.getString(
                     R.string.show_status,
                     totalStr,
```

```
                        "R$0,00")
                    )
                }
            }
        }

        override fun newParcial(value: Double) {
            limit?.let { safeLimit ->
                val totalStr: String = format.format(safeLimit)
                val parcialStr: String = format.format(value)
                App.context?.let { safeContext ->
                    view.showNewStatus(
                        safeContext.getString(
                            R.string.show_status,
                            totalStr,
                            parcialStr
                        )
                    )
                }
            }
        }

        override fun start() {}

}
```

Código-fonte 3.21 – Novas variáveis para a MainActivity.
Fonte: Elaborado pelo autor (2020)

E, por fim, temos a nova *ListItensActivity,* do pacote *mvp -> view.* Vemos no código-fonte Conteúdo da classe ListItensActivity que nesta *Activity* também tivemos uma redução considerável, visto que, toda lógica de negócio foi para o *presenter*, restando apenas o código relacionado à visão, afinal, as *Activities* pertencem à camada de *View* no MVP.

```
class ListItensActivity : AppCompatActivity(), View.OnKeyListener,
ListItensContract.View {

    private lateinit var viewManager: RecyclerView.LayoutManager
    private lateinit var viewAdapter: MyAdapter

    override lateinit var presenter : ListItensPresenter

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_list_itens)

        presenter = ListItensPresenter(this)
        presenter.readLimit(intent)

        edtNewItem.setOnKeyListener(this@ListItensActivity)

        viewManager = LinearLayoutManager(this)
        viewAdapter =
            MyAdapter() {
                presenter.newParcial(it)
            }
```

```
        recycler.apply {
            setHasFixedSize(true)
            layoutManager = viewManager
            adapter = viewAdapter
        }
    }

    override fun onKey(view: View?, keyCode: Int, event: KeyEvent?):
Boolean {
        event?.let {
            if ((it.getAction() == KeyEvent.ACTION_DOWN) &&
                (keyCode == KeyEvent.KEYCODE_ENTER)) {
                viewAdapter.add(
                    Item(
                        label = edtNewItem.text.toString(),
                        total = 0.0
                    )
                )
                edtNewItem.setText("")
                return true;
            }
        }

        return false
    }

    override fun showNewStatus(status: String) {
        txtStatus.setText(status)
    }

}
```

Código-fonte 3.22 – Conteúdo da classe ListItensActivity.
Fonte: Elaborado pelo autor (2020)

O código do *adapter* se manteve igual, porém, seria possível colocar a lógica de negócio do adaptador também em um *presenter*.

3.12 MVVM

Chegamos então à última arquitetura a ser estudada. O MVVM é mais parecido com o MVP porque o *View-Model* toma o lugar do *Presenter* naquela imagem das três camadas inter-relacionadas.

Apesar das semelhanças, há algumas diferenças cruciais. Temos a presença do JetPack e suas bibliotecas de componentes arquiteturais, como também esses mesmos componentes já relacionados com o ciclo de vida das telas e *Fragments*. Além disso, o JetPack ainda fornece classes com o nome muito claro, em relação à arquitetura MVVM, como *ViewModel* e *DataBinding.*

3.13 JetPack

O JetPack é um conjunto de bibliotecas que fazem parte de uma tentativa, do Android, de padronizar o desenvolvimento mobile em sua plataforma, e também unificou bibliotecas de suporte, evitando a confusão que existia nos pacotes android-support-<versão>.

De acordo com a documentação oficial do Android JetPack (https://developer.android.com/jetpack), podemos dividi-los em quatro grandes grupos:

- **Base**: fornecem funcionalidade transversal, como compatibilidade com versões anteriores, testes e compatibilidade com a linguagem Kotlin.
- **Arquitetura**: ajudam a criar apps robustos, testáveis e de fácil manutenção.
- **Comportamento**: ajudam seu app a se integrar aos serviços padrão do Android, como notificações, permissões, compartilhamento e o Assistente.
- **IU**: fornecem *widgets* e assistentes para tornar seu app não apenas fácil, como também agradável de usar. Aprenda sobre o JetPack *Compose* para ajudar a simplificar o desenvolvimento da IU.

É, portanto, na parte dos componentes arquiteturais que o Google indica uma proximidade ao MVVM, nessa tendência de desenvolvimento.

Veja a Figura Guia de Arquitetura Google, ele é encontrado em outro documento oficial do Android chamado: Guia para a arquitetura do app: <https://developer.android.com/jetpack/docs/guide#overview>. Veja como fica sua comparação com o MVVM:

- **Camada de Model**: é o *Repository* e suas camadas inferiores. A ideia é que tenhamos um repositório de dados como fonte verdadeira do conhecimento. Essa camada saberá de onde precisa buscar esses dados, os quais podem ser de uma fonte local, como um banco SQLite ou, de uma fonte externa, como um Firebase, por exemplo.
- **Camada View**: são as *Activities* e *Fragments*, mas também temos os arquivos XML de layout.

- **View/Model**: também é o nome de classe homônima. Além disso, para o binder, entre modelo e visão, a biblioteca já oferece as classes de *DataBinding* e *LiveData.*

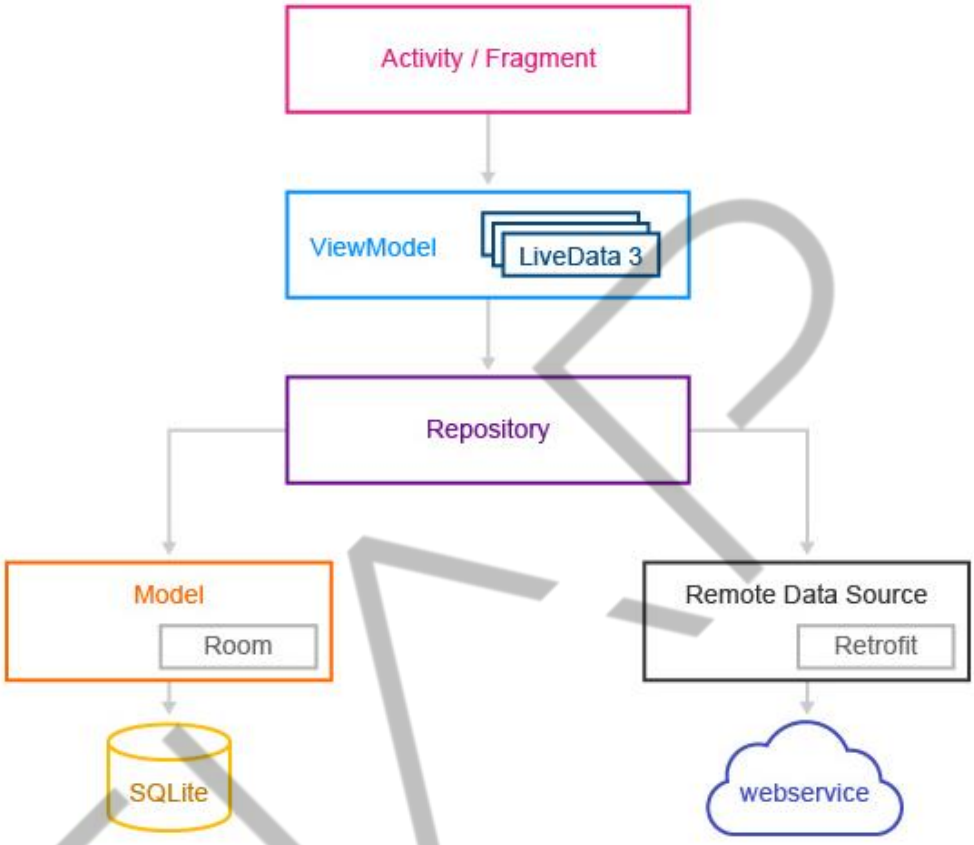

Figura 3.9 - Guia de Arquitetura Google
Fonte: Developer (2020)

3.14 Codificando em MVVM e JetPack

Para começar, faça algumas modificações na estrutura de pastas e arquivos do nosso projeto. Veja como deve ficar seu projeto na Figura Arquitetura de pastas e arquivos para a implementação MVVM.

A grande diferença está na pasta *model.* Ela já havia sido usada exatamente da mesma forma, nas pastas mvc e mvp. Como ela seria usada novamente, foi movida para a raiz para não a deixar duplicada em mvc e mvp. Além disso, a pasta mvvm não está mais vazia e contém subpastas e arquivos. Importante também perceber que temos uma nova pasta *adapter* dentro de mvvm.

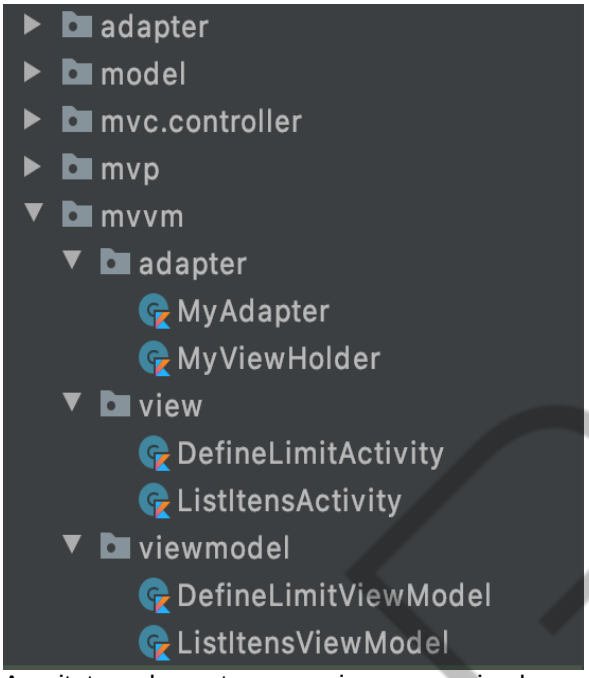

Figura 3.10 - Arquitetura de pastas e arquivos para a implementação MVVM
Fonte: Elaborado pelo autor (2020)

Uma vez feitas essas alterações, precisamos definir o suporte a todas bibliotecas do JetPack que vamos usar no projeto. Portanto, abra o arquivo *build.gradle* pertencente ao módulo *app,* e deixe seu conteúdo conforme o código-fonte Alterações no serviço declarado no manifesto. As alterações necessárias foram grafadas em negrito, nas linhas.

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
apply plugin: 'kotlin-kapt'

android {
   compileSdkVersion 29
   buildToolsVersion "29.0.3"

   defaultConfig {
       applicationId "com.estudo.android.controlecompras"
       minSdkVersion 21
       targetSdkVersion 29
       versionCode 1
       versionName "1.0"

       testInstrumentationRunner
"androidx.test.runner.AndroidJUnitRunner"
   }

   buildFeatures {
       dataBinding = true
   }

   buildTypes {
```

```
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-
optimize.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    def lifecycle_version = "2.2.0"

    // ViewModel
    implementation "androidx.lifecycle:lifecycle-viewmodel-
ktx:$lifecycle_version"
    // LiveData
    implementation "androidx.lifecycle:lifecycle-livedata-
ktx:$lifecycle_version"
    // Lifecycles only (without ViewModel or LiveData)
    implementation "androidx.lifecycle:lifecycle-runtime-
ktx:$lifecycle_version"

    implementation "androidx.core:core-ktx:1.3.1"

    implementation fileTree(dir: "libs", include: ["*.jar"])
    implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
    implementation 'androidx.core:core-ktx:1.3.1'
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'

    implementation 'com.google.android.material:material:1.3.0-alpha02'
    implementation "androidx.recyclerview:recyclerview:1.1.0"
}
```

Código-fonte 3.23 – Alterações no serviço declarado no manifesto.
Fonte: Elaborado pelo autor (2020)

Depois de sincronizar o projeto novamente, vamos estudar a classe *DefineLimitViewModel,* localizada na pasta *mvvm.*

Logo no início temos duas variáveis: a primeira totalmente comum, apenas uma *String* que começa com valor padrão *“”*. A segunda é um conceito novo. A variável *errorMessage* é do tipo *MutableLiveData*, como o nome indica, é um dado “vivo”, em que podemos usar o design *pattern Observable* facilmente por meio da própria API da classe. Como esse erro faz parte do campo de entrada de texto da primeira tela, precisamos justamente observar sua mudança para, quando existir um erro, passá-lo para o método *setError* do componente visual.

Perceba que esse erro é configurado na cláusula *else* do método *goList* e, como não estamos trabalhando diretamente com um booleano, é necessário configurar o *value* do *MutableLiveData.* O restante deste código, dentro do método, é a mesma

lógica que usávamos no *presenter.* Porém temos uma grande diferença: não há ligação direta com a *View* e a visão de quem se conectará com nosso *viewmodel*.

```
class DefineLimitViewModel : ViewModel() {

   var limitValue: String  = ""
   var errorMessage = MutableLiveData<Boolean>()

   fun goList(){
       if (limitValue.trim().length > 0) {
           App.context?.let { safeContext ->
               val intent = Intent(safeContext,
ListItensActivity::class.java)
               intent.putExtra("limit", limitValue.toDouble())
               safeContext.startActivity(intent)
           }
       } else {
           errorMessage.value = true
       }
   }

}
```

Código-fonte 3.24 – Conteúdo da classe DefineLimitViewModel.
Fonte: Elaborado pelo autor (2020)

Já no código-fonte Conteúdo da classe DefineLimitActivity, temos o conteúdo da classe *DefineLimitActivity,* que está no pacote *mvvm -> view*. É perceptível como o código efetivo da tela diminui ainda mais em relação ao MVP.

Dentro do método *onCreate,* estamos criando uma instância do nosso *DefineLimitViewModel.* Para isso, usamos a classe auxiliar *ViewModelProvider* e seu método *get* no qual passamos por parâmetro, o *class* do *viewmodel*, que queremos instanciar.

Outra mudança relevante está no uso do *setContentView.* Estamos usando um método homônimo, porém, ele é estático e pertence à classe *DataBindingUtil.* A variável nomeada como *binding* é do tipo *ActivityMainMvvmBinding.* Essa classe ainda não foi vista, ela é gerada de forma automática quando definimos o arquivo de layout XML usando a tag *layout.* Mas isso é assunto para alguns parágrafos adiante.

Por enquanto, basta saber que essa classe é gerada automaticamente e que ela funciona como uma intermediária entre nossa *Activity* e a nossa *View* do XML. No *binding* estamos definindo a *viewMovel* e o seu *lifecycleOwner.* O proprietário do ciclo de vida serve para que a própria API já faça esse controle, e não precisamos nos preocupar em liberar recursos quando o ciclo da *Activity* se encerrar.

Nas últimas linhas do *onCreate* estamos criando um *observer* para o *errorMessage* da *viewModel*. Lembre-se de que definimos essa variável como *MutableLiveData.* Sendo assim, quando ela for alterada, seremos direcionados automaticamente para a classe interna anônima, filha de *Observer,* que já definimos neste código. Se o novo valor do *liveData* for *true*, configuramos a mensagem de erro do campo de entrada de texto.

```
class DefineLimitActivity : AppCompatActivity() {

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)

       val viewModel: DefineLimitViewModel  =
ViewModelProvider(this).get(DefineLimitViewModel::class.java)

       val binding: ActivityMainMvvmBinding =
DataBindingUtil.setContentView(this, R.layout.activity_main_mvvm)
       binding.viewModel = viewModel
       binding.lifecycleOwner = this

       viewModel.errorMessage.observe(this, Observer {
           if (it){
               txtValue.error = "Insira um valor válido para seguir!"
           } else {
               txtValue.error = null
           }
       })
   }

}
```

Código-fonte 3.25 – Conteúdo da classe DefineLimitActivity.
Fonte: Elaborado pelo autor (2020)

Já estudamos o código do *ViewModel* da *Activity*, bem como o código da nossa tela que é um *AppCompatActivity.* Para total entendimento, precisamos estudar o nosso arquivo de layout. Crie um arquivo XML chamado *activity_main_mvvm.xml* e codifique-o exatamente como é mostrado no código-fonte Conteúdo do arquivo activity_main_mvvm.xml:

A tag raiz passa a ser um *layout* e dentro dessa tag temos as definições que tínhamos anteriormente; no nosso caso, um *ConstraintLayout.* Podemos também definir uma variável dentro da tag *data,* com o uso da tag *variable.* Definimos o nome dessa variável como *viewModel* e seu tipo como a *DefineLimitViewModel,* que estudamos agora há pouco.

Ao salvar este arquivo, automaticamente a classe *ActivityMainMvvmBinding* será criada. O nome da classe gerada sempre será o nome do arquivo xml, sem

underlines, acrescida do sufixo *Binding.* Agora deve ter ficado mais nítida a razão de termos a linha "*binding.viewModel = viewModel"* na nossa *activity*. Ela serve para definir a variável que foi criada neste layout, dentro da tag *data->variable.*

Agora, no componente *TextInputEditText*, estamos passando uma propriedade *text* diferenciada. Quando usarmos *@{},* estaremos apenas mostrando o valor de uma variável ou uma constante. Quando usamos *@={},* como é nosso caso, estamos criando um *binding* em duas direções; ou seja, inicialmente ele mostra o valor da variável, mas, à medida que editamos esse campo de entrada de texto, ele também altera diretamente essa variável.

Em nosso código estamos passando esse valor para o *text: @={viewModel.limitValue}*, significa que sempre que alteramos esse campo, automaticamente ele altera a variável *limitValue* da classe apontada na variável *viewModel,* na definição do *data,* neste arquivo de layout.

Por fim, veja que no *AppCompatButton* estamos passando para a propriedade *onClick* o valor *"@{() -> viewModel.goList()}".* Com isso, o evento de clique causará a chamada ao método da classe direcionada na variável *viewModel,* com o identificador *goList.* Perceba que é o nome do método que criamos na nossa classe de *ViewModel.*

Uma diferença importante que temos neste código: a visão que se conectou ao nosso *ViewModel* e não o nosso *presenter* se conectando à Visão, como acontecia anteriormente no MVP. Isso diminui o acoplamento e consequentemente fica mais fácil evitar *memory leaks.*

```
<service
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">

    <data>
        <variable
            name="viewModel"
type="com.estudo.android.controlecompras.mvvm.viewmodel.DefineLimitViewMo
del" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:padding="20dp"
        android:layout_height="match_parent"
        tools:context=".mvc.controller.DefineLimitActivity">
        <LinearLayout
            android:orientation="vertical"
```

```
            app:layout_constraintTop_toTopOf="parent"
            app:layout_constraintBottom_toBottomOf="parent"
            android:layout_width="match_parent"
            android:layout_height="wrap_content">
            <androidx.appcompat.widget.AppCompatImageView
                android:layout_gravity="center_horizontal"
                android:src="@drawable/frutas"
                android:layout_width="240dp"
                android:layout_marginBottom="20dp"
                android:layout_height="240dp"/>
            <com.google.android.material.textfield.TextInputLayout
                android:id="@+id/txtValue"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                app:layout_constraintBottom_toBottomOf="parent"
                app:layout_constraintLeft_toLeftOf="parent"
                app:layout_constraintRight_toRightOf="parent"
                app:layout_constraintTop_toTopOf="parent"
                android:hint="Valor Máximo">

                <com.google.android.material.textfield.TextInputEditText
                    android:id="@+id/edtValue"
                    android:text="@={viewModel.limitValue}"
                    android:inputType="numberDecimal"
                    android:layout_width="match_parent"
                    android:layout_height="wrap_content"/>

            </com.google.android.material.textfield.TextInputLayout>

            <androidx.appcompat.widget.AppCompatButton
                android:layout_width="wrap_content"
                android:text="Iniciar"
                android:onClick="@{() -> viewModel.goList()}"
                android:id="@+id/btnGo"
                android:layout_gravity="right"
                android:layout_height="wrap_content"/>
        </LinearLayout>

    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Código-fonte 3.26 – Conteúdo do arquivo activity_main_mvvm.xml.
Fonte: Elaborado pelo autor (2020)

Depois da tela de definição de limite, partimos para a segunda tela que é a listagem dos itens. Começamos pelo seu *ViewModel* codificado na classe *ListItensViewModel.* Veja seu conteúdo no código-fonte Conteúdo da classe ListItensViewModel:

Grande parte da lógica de negócios é apenas uma cópia do que havíamos feito no *presenter* do MVP, porém, temos uma pequena diferença que é de grande impacto. Antes, passávamos para o contrato a responsabilidade de mudança na *View*. Aqui apenas criamos uma variável *status,* do tipo *MutableLiveData<String>.* Posteriormente, nossa *View* vai se conectar a este dado, trabalhando como um

Observer e alterando seu conteúdo automaticamente, quando a *viewModel* a modificar.

```
class ListItensViewModel : ViewModel() {

   private var limit: Double? = null
   private val format: NumberFormat =
NumberFormat.getCurrencyInstance(Locale.getDefault())

   var status = MutableLiveData<String>()

   fun readLimit(intent: Intent) {
       limit = intent.getDoubleExtra("limit", 0.0)

       limit?.let {
           App.context?.let { safeContext ->
               val totalStr: String = format.format(it)
               status.value = safeContext.getString(R.string.show_status,
totalStr, "R$0,00")
           }
       }
   }

   fun newParcial(value: Double) {
       limit?.let { safeLimit ->
           val totalStr: String = format.format(safeLimit)
           val parcialStr: String = format.format(value)
           App.context?.let { safeContext ->
               status.value = safeContext.getString(
                   R.string.show_status,
                   totalStr,
                   parcialStr
               )
           }
       }
   }

}
```

Código-fonte 3.27 – Conteúdo da classe ListItensViewModel.
Fonte: Elaborado pelo autor (2020)

Passamos do *ViewModel* para a *ListItensActivity,* que está na pasta *mvvm->view.* Veja seu conteúdo no código-fonte Conteúdo da classe ListItensActivity. Novamente temos um esboço semelhante àquele visto no MVP, porém, no MVVM precisamos criar o *binding* entre a *Activity* e o *ViewModel.* Grafamos com negrito os pontos de atenção neste código.

```
class ListItensActivity : AppCompatActivity(), View.OnKeyListener {

   private lateinit var viewManager: RecyclerView.LayoutManager
   private lateinit var viewAdapter: MyAdapter

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
```

```
        val viewModel: ListItensViewModel =
ViewModelProvider(this).get(ListItensViewModel::class.java)

        val binding: ActivityListItensMvvmBinding =
DataBindingUtil.setContentView(this, R.layout.activity_list_itens_mvvm)
        binding.viewModel = viewModel
        binding.lifecycleOwner = this

        viewModel.readLimit(intent)

        edtNewItem.setOnKeyListener(this@ListItensActivity)

        viewManager = LinearLayoutManager(this)
        viewAdapter =
            MyAdapter() {
                viewModel.newParcial(it)
            }

        recycler.apply {
            setHasFixedSize(true)
            layoutManager = viewManager
            adapter = viewAdapter
        }
    }

    override fun onKey(view: View?, keyCode: Int, event: KeyEvent?):
Boolean {
        event?.let {
            if ((it.getAction() == KeyEvent.ACTION_DOWN) &&
                (keyCode == KeyEvent.KEYCODE_ENTER)) {
                viewAdapter.add(
                    Item(
                        label = edtNewItem.text.toString(),
                        total = 0.0
                    )
                )
                edtNewItem.setText("")
                return true;
            }
        }

        return false
    }
}
```

Código-fonte 3.28 – Conteúdo da classe ListItensActivity.
Fonte: Elaborado pelo autor (2020)

O arquivo de layout dessa tela também precisou de algumas alterações pontuais, porém, cruciais. Veja o código-fonte Conteúdo do arquivo activity_list_itens_mvvm.xml, que representa o arquivo *activity_list_itens_mvvm*. Nele, o *dataBinding* foi usado com mais economia, apenas no *TextView* de status passamos para a propriedade *text* o seguinte valor: *android:text="@{viewModel.status}"*. Dessa forma, o valor da variável *status,* dentro da nossa *viewModel,* será inferida no elemento visual de forma automática.

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools">

   <data>
       <variable
           name="viewModel"

type="com.estudo.android.controlecompras.mvvm.viewmodel.ListItensViewMode
l" />
   </data>

   <androidx.constraintlayout.widget.ConstraintLayout
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       tools:context=".mvc.controller.ListItensActivity">

       <androidx.recyclerview.widget.RecyclerView
           android:padding="12dp"
           android:id="@+id/recycler"
           android:layout_width="match_parent"
           android:layout_height="0dp"
           app:layout_constraintTop_toTopOf="parent"
           app:layout_constraintBottom_toTopOf="@id/lineSeparator"/>

       <View
           android:id="@+id/lineSeparator"
           app:layout_constraintTop_toBottomOf="@id/recycler"
           app:layout_constraintBottom_toTopOf="@id/txtStatus"
           android:background="@color/colorAccent"
           android:layout_width="match_parent"
           android:layout_height="1dp"/>

       <TextView
           app:layout_constraintTop_toBottomOf="@id/lineSeparator"
           app:layout_constraintBottom_toTopOf="@id/txtItem"
           android:id="@+id/txtStatus"
           android:gravity="center"
           android:text="@{viewModel.status}"
           android:layout_width="match_parent"
           android:layout_height="40dp"/>

       <com.google.android.material.textfield.TextInputLayout
           android:id="@+id/txtItem"
           android:layout_width="match_parent"
           android:layout_height="wrap_content"
           app:layout_constraintBottom_toBottomOf="parent"
           app:layout_constraintLeft_toLeftOf="parent"
           app:layout_constraintRight_toRightOf="parent"
           app:layout_constraintTop_toBottomOf="@id/txtStatus"
           android:hint="Insira o item">

           <com.google.android.material.textfield.TextInputEditText
               android:id="@+id/edtNewItem"
               android:lines="1"
               android:imeOptions="actionSend"
               android:maxLines="1"
               android:singleLine="true"
               android:layout_width="match_parent"
               android:layout_height="wrap_content"/>
```

```
        </com.google.android.material.textfield.TextInputLayout>

    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Código-fonte 3.29 – Conteúdo do arquivo activity_list_itens_mvvm.xml.
Fonte: Elaborado pelo autor (2020)

Na segunda tela, a principal mudança será no nosso *adapter*, no qual faremos o uso do *dataBinding* também. Começaremos pelo novo arquivo de layout, que foram os itens do *RecyclerView.* Crie um novo arquivo de layout chamado *item_list_mvvm* e codifique-o conforme o código-fonte Conteúdo do arquivo item_list_mvvm.xml.

O layout foi transformado para ter suporte ao *dataBinding.* Dessa forma, possuímos a tag raiz *layout,* seguida da tag *data* e sua *variable.* Neste, apontamos para a classe *Item,* que no nosso projeto é apenas um *dataClass.* Todos os componentes visuais agora usam o *@{}* para mostrar os dados que estão configurados na instância da classe Item, referenciado pela variável *item* que criamos no *variable.*

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <variable
            name="item"
            type="com.estudo.android.controlecompras.model.Item" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:gravity="center_vertical"
        android:layout_height="wrap_content">

        <androidx.appcompat.widget.AppCompatTextView
            android:id="@+id/txtTotalItem"
            android:layout_width="wrap_content"
            android:text='@{item.valueInCurrency}'
            android:layout_marginRight="10dp"
            android:textSize="14sp"
            android:textStyle="bold"
            android:layout_height="wrap_content"/>

        <androidx.appcompat.widget.AppCompatTextView
            android:id="@+id/txtItemLabel"
            android:layout_width="0dp"
            android:text="@{item.label}"
            android:layout_weight="1"
            android:textSize="14sp"
            android:layout_height="wrap_content"/>

        <com.google.android.material.textfield.TextInputEditText
            android:hint="Qtd"
```

```
            android:textSize="14sp"
            android:id="@+id/edtQtd"
            android:inputType="numberDecimal"
            android:layout_width="50dp"
            android:text="@{item.qtdStr}"
            android:layout_marginRight="10dp"
            android:layout_height="wrap_content"/>

        <com.google.android.material.textfield.TextInputEditText
            android:textSize="14sp"
            android:inputType="numberDecimal"
            android:hint="Valor"
            android:text="@{item.valueStr}"
            android:id="@+id/edtValue"
            android:layout_width="50dp"
            android:layout_height="wrap_content"/>

    </LinearLayout>
</layout>
```

Código-fonte 3.30 – Conteúdo do arquivo item_list_mvvm.xml.
Fonte: Elaborado pelo autor (2020)

Você deve ter percebido que estamos referenciando as variáveis *qtdStr* e *valueStr*, *as* quais não existiam na classe *Item.* De fato, também fizemos algumas mudanças nessa classe, que são ilustradas no código-fonte Conteúdo do Item.kt.

A classe agora possui mais duas propriedades: *qtd* e *value.* Além disso, possui métodos que tratam os dados que são números reais, transformando em *Strings* para melhor compreensão do usuário. Normalmente, em projetos maiores, temos o conceito de *Mapper,* que seria uma classe intermediária, que trabalha em cima de um modelo de dados e já entrega os dados tratados e formatados, de acordo com o caso de uso necessário.

```
package com.estudo.android.controlecompras.model

import java.text.NumberFormat
import java.util.*

data class Item (
    val label: String,
    var total: Double,
    var qtd: Double = 0.0,
    var value: Double = 0.0
) {

    private val format: NumberFormat =
NumberFormat.getCurrencyInstance(Locale.getDefault())

    fun getValueInCurrency() : String{
        return format.format(total)
    }
```

```
    fun getQtdStr() : String{
        if (qtd.compareTo(0) == 0){
            return ""
        } else if (Math.ceil(qtd) == Math.floor(qtd)){
            return qtd.toInt().toString()
        } else {
            return qtd.toString()
        }
    }

    fun getValueStr() : String{
        if (value.compareTo(0) == 0){
            return ""
        } else if (Math.ceil(value) == Math.floor(value)){
            return value.toInt().toString()
        } else {
            return value.toString()
        }
    }

}
```

Código-fonte 3.31 – Conteúdo do Item.kt.
Fonte: Elaborado pelo autor (2020)

O nosso adapter, como um todo, foi dividido em duas classes no MVVM. Temos a *MyViewHolder* e a *MyAdapter.* Começaremos pelo *MyViewHolder* que pode ser visualizado no código-fonte Conteúdo do arquivo MyViewHolder.kt.

Logo no início, já notamos que esse *holder* tem um método a mais, recebido por parâmetro. Trata-se do *deleteItem.* Também temos uma nova variável chamada de *binding.* Como nosso arquivo de layout dos itens do *recycler* agora possui o padrão para *binder,* uma classe foi gerada com o nome do arquivo (sem underlines) acrescido do sufixo *binding*: a classe *ItemListMvvmBinding.*

Para criar essa variável, estamos usando a classe auxiliar *DataBindingUtil* e seu método estático *bind.* O parâmetro é a *View* que será recebida no construtor da presente classe.

No bloco *init*, estamos criando os *listeners* para os componentes visuais, mas, não estamos acessando-os diretamente da *View* e, sim, diretamente do *binding.*

Com o método *calcValue,* após a lógica de negócio de somar o novo valor da compra, estamos mudando os valores da instância de *Item* associados ao *binding,* bem como editando o valor textual do componente visual que possui o id *txtTotalItem.*

Por fim, temos o método *bind.* Ele recebe a posição desse *holder* no *RecyclerView,* assim como a instância de *Item* que temos nessa mesma posição. Com

isso, definimos o item associado ao *binding* e configuramos os valores iniciais dos elementos de entrada de texto. Isso foi necessário porque índices da listagem podem ser excluídos, dessa forma, precisamos atualizar os outros itens de maneira adequada.

```
<?xml version="1.0" encoding="utf-8"?>
class MyViewHolder(
   view: View,
   private val makeCalNow: () -> Unit,
   private val deleteItem: (position: Int) -> Unit
) : RecyclerView.ViewHolder(view) {

   val binding: ItemListMvvmBinding? = DataBindingUtil.bind(view)

   var position: Int? = null

   init {
       binding?.edtQtd?.addTextChangedListener(object: TextWatcher {
           override fun beforeTextChanged(s: CharSequence?, start: Int,
count: Int, after: Int) {}
           override fun onTextChanged(s: CharSequence?, start: Int,
before: Int, count: Int) {
               calcValue()
           }
           override fun afterTextChanged(s: Editable?) {}
       })

       binding?.edtValue?.addTextChangedListener(object: TextWatcher {
           override fun beforeTextChanged(s: CharSequence?, start: Int,
count: Int, after: Int) {}
           override fun onTextChanged(s: CharSequence?, start: Int,
before: Int, count: Int) {
               calcValue()
           }
           override fun afterTextChanged(s: Editable?) {}
       })


       binding?.txtItemLabel?.setOnLongClickListener {
           position?.let {
               deleteItem(it)
           }

           true
       }

   }

   fun calcValue(){
       val qtd = binding?.edtQtd?.text.toString()
       val value = binding?.edtValue?.text.toString()

       var total: Double = 0.0;
       try {
           total = qtd.toDouble() * value.toDouble()
       } catch (exc: NumberFormatException){}

       position?.let {
```

```
            binding?.item?.total = total
            binding?.item?.qtd = try { qtd.toDouble() } catch(exception:
java.lang.NumberFormatException) { 0.0 }
            binding?.item?.value = try { value.toDouble() }
catch(exception: java.lang.NumberFormatException) { 0.0 }
            binding?.txtTotalItem?.text =
binding?.item?.getValueInCurrency()
            makeCalNow()
        }

    }

    fun bind(pos: Int, item: Item){
        binding?.item = item
        binding?.executePendingBindings()

        if (item.qtd.compareTo(0) != 0){
            binding?.edtQtd?.setText(item.qtd.toString())
        }

        if (item.value.compareTo(0) != 0){
            binding?.edtValue?.setText(item.value.toString())
        }

        position = pos
    }

}
```

Código-fonte 3.32 – Conteúdo do arquivo MyViewHolder.kt.
Fonte: Elaborado pelo autor (2020)

O último passo da jornada é estudar a classe *MyAdapter,* mostrada no código-fonte Conteúdo do arquivo MyAdapter.kt. Com a separação do *holder* em uma classe externa, foi possível enxugar o número de linhas da *MyAdapter.*

O código que sobrou está muito parecido ao que já havíamos estudado no capítulo sobre MVP. Há uma diferença: a da hora de instanciar a *MyViewHolder.* Perceba que precisamos passar dois métodos anônimos, justamente porque mudamos o construtor dessa classe.

O segundo método receberá um valor inteiro, indicando qual índice da posição deve ser removido. O código restante é a mesma lógica anterior, porém, só a mudamos de lugar.

```
class MyAdapter(
    val returnSum: (Double) -> Unit
) : RecyclerView.Adapter<MyViewHolder>() {

    val dataSource = ItensSource()

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
MyViewHolder {
        return MyViewHolder(
```

```
            LayoutInflater
                .from(parent.context)
                .inflate(R.layout.item_list_mvvm, parent, false),
            {
                returnSum(dataSource.myDataset.sumByDouble { item ->
item.total })
            },
            deleteItem = { position ->
                dataSource.myDataset.removeAt(position)
                notifyItemRemoved(position)
                returnSum(dataSource.myDataset.sumByDouble { item ->
item.total })
            }
        )
    }

    override fun onBindViewHolder(holder: MyViewHolder, position: Int) {
        holder.bind(position, dataSource.myDataset[position])
    }

    override fun getItemViewType(position: Int): Int {
        return position
    }

    override fun getItemId(position: Int): Long {
        return position.toLong()
    }

    override fun getItemCount() = dataSource.myDataset.size

    fun add(item: Item) {
        dataSource.myDataset.add(item)
        notifyItemInserted(dataSource.myDataset.size - 1)
    }
}
```

Código-fonte 3.33 – Conteúdo do arquivo MyAdapter.kt.
Fonte: Elaborado pelo autor (2020)

E, dessa forma, concluímos nosso projeto em MVVM e JetPack.

CONCLUSÃO

As arquiteturas para desenvolvimento em Android ganharam muita força nos últimos anos, devido ao crescimento e amadurecimento da plataforma. Com efeito, os projetos ficaram maiores, bem como as equipes de desenvolvimento, demandando maior padronização na forma de desenvolver os aplicativos.

Existem diversas arquiteturas. Algumas delas ganharam mais destaques no mercado ou fizeram parte do crescimento do Android ao longo do tempo. Por isso, é muito comum os desenvolvedores terem passado pela tríade: MVC-MVP-MVVM. Há ainda outras boas opções, como MVI e Vipe, que não foram citadas. No entanto, com o advento do JetPack, o Android direcionou seu futuro para o MVVM, razão pela qual foi priorizado neste capítulo.

Apresentamos as diferenças e a evolução entre as três principais arquiteturas. Não existe uma forma exata para utilizar qualquer uma delas, mas devemos seguir guias e boas práticas de suas implementações para tirar melhor proveito possível de cada uma delas.

Fazendo isso, teremos códigos escaláveis, testáveis, modernos, performáticos e de alta manutenibilidade. E o mais importante: é o que o mercado exige dos bons profissionais nos dias atuais.

Link para o GitHub do projeto completo:
<https://github.com/FIAPON/ControleCompras>.

REFERÊNCIAS

ANDROID. **Android Layout Training**. [s.d.]. Disponível em: <https://developer.android.com/training/wearables/ui/layouts>. Acesso em: 17 set. 2020.

CASE, T. **God Activity Architecture** – One Architecture To Rule Them All. 2019. <https://medium.com/@taylorcase19/god-activity-architecture-one-architecture-to-rule-them-all-62fcd4c0c1d5>. Acesso em: 17 set. 2020.

MACORATTI, J. C. ASP .NET MVC - Criando uma aplicação básica - CRUD com dropdownlist (Iniciante) - I. 2015. Disponível em: <http://www.macoratti.net/15/04/mvc_crudb1.htm>. Acesso em: 17 set. 2020.

MATERIAL DESIGN ICONS. [s.d.]. Disponível em: <https://materialdesignicons.com/>. Acesso em: 17 set. 2020.

PIRES, A. F. Padrão de Arquitetura MVP no Android. 2019. Disponível em: <https://medium.com/@alifyzfpires/padr%C3%A3o-de-arquitetura-mvp-no-android-23a6fa96a27b>. Acesso em: 17 set. 2020.

STAT COUNTER. **Mobile Operating System Market Share Worldwide**. Disponível em: <https://gs.statcounter.com/os-market-share/mobile/worldwide>. Acesso em: 17 set. 2020.

ZK Developer. **MVVM**. 2015. <https://www.zkoss.org/wiki/ZK_Developer%27s_Reference/MVVM>. Acesso em: 17 set. 2020.