{desafío} latam_

Patrones de diseño y callbacks - Parte II

Capítulo: Modelo - Vista - VistaModelo (MVVM)

Competencias

- Enlazar las capas de vista y modelo usando ViewModel
- Conocer las diferencias entre MVVM con MVP

Introducción

MVVM es una evolución del patrón MVC y usa una capa de elementos “no visuales” intermedia entre el modelo y la vista. En Android el patrón MVVM se implementa usando la biblioteca DataBinding que permite la propagación de cambios desde el estado de ViewModel a View. Generalmente la implementación de ViewModel utiliza el patrón observador para informar de los cambios del ViewModel a la vista.

El **ViewModel** es responsable de encapsular el modelo y preparar la data observable necesaria para la vista. Además, provee la forma de pasar los eventos desde la vista al modelo. Sin embargo, el ViewModel no está ligado a la vista

En MVVM el encargado de entregar los cambios desde ViewModel a View es llamado **Binder**, que es usualmente manejado por la plataforma, desligando a los desarrolladores de cómo es efectivamente la vista es notificada de los cambios.

Google actualmente está promoviendo este patrón de arquitectura como la forma recomendada de desarrollar apps nativas. Además, MVVM se lleva muy bien con los componentes de JetPack que interactúan usando este patrón.

Comprender y aplicar este patrón entrega ventajas reales a la hora de diseñar e implementar una aplicación utilizando la última tecnología disponible.

MVVM en Android

En Android el patrón MVVM fue introducido en la Google I/O 2015, presentado junto al conjunto de bibliotecas de JetPack y es implementado usando DataBinding que actúa como el *binder* entre la vista y el ViewModel

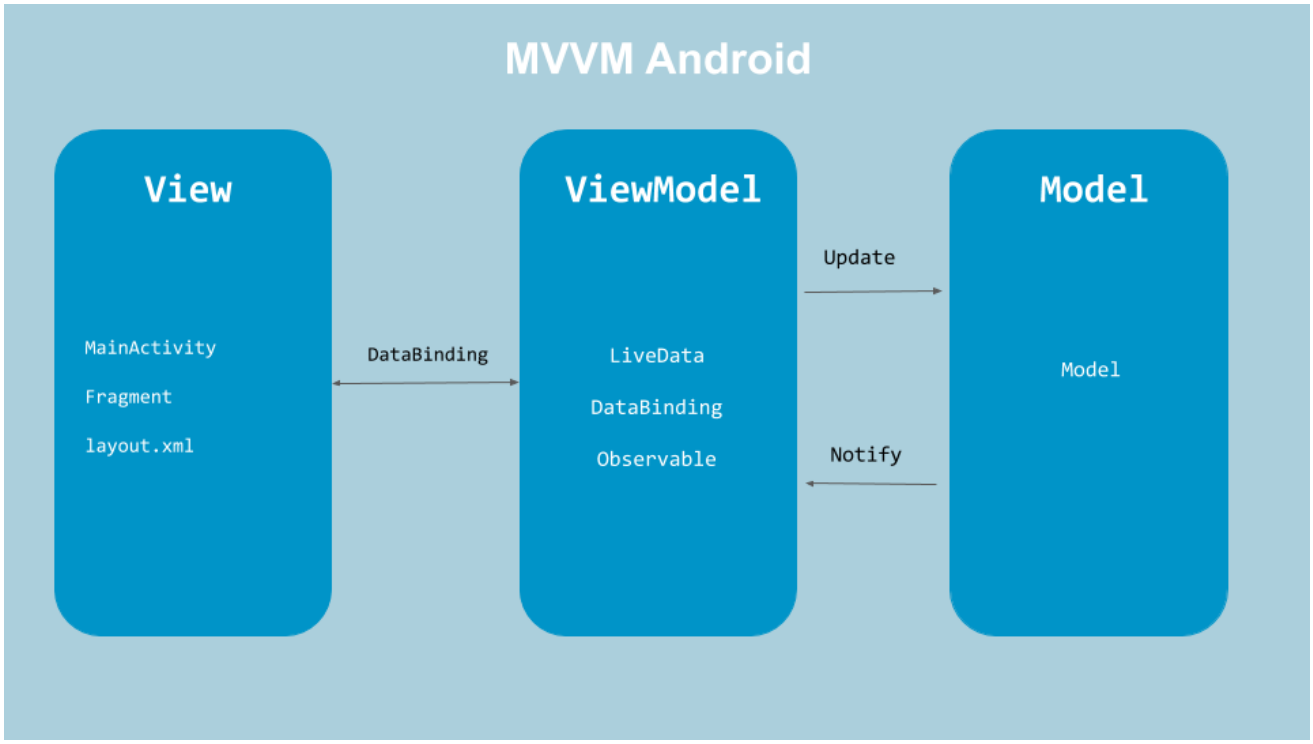

Imagen 1. Model - View - ViewModel.

ViewModel

En Android, la clase ViewModel fue diseñada para almacenar y manejar la información relacionada a la UI de una forma conciente del ciclo de vida. La clase ViewModel permite a la data sobrevivir a los cambios de configuración como la rotación de pantalla

El framework de Android es el que se encarga de manejar los ciclos de vida de los controladores cómo Activities ó Fragments, y es el que decide destruir o recrear estos controladores en respuesta a las acciones de los usuarios o eventos del dispositivo, muy por sobre el nivel de la aplicación, y por lo tanto, son eventos fuera de nuestro alcance

Cada vez que se destruye o recrea uno de estos controladores, la información que contenía se pierde.

¿Cómo afecta esto?

Por ejemplo, una aplicación con un Activity que tiene una lista de elementos que son desplegados al abrirse la aplicación. Si el usuario hace rotar la aplicación, la actividad debe recrearse usando el nuevo layout, pero también debe pedir nuevamente la información a desplegar

Extendiendo la clase ViewModel este comportamiento puede ser manejado de mejor forma. ViewModel retiene la información cuando ocurren cambios en la configuración (como cambiar la orientación) y cuando la actividad es recreada, la información está disponible en forma inmediata

Esta clase MyViewModel tiene una lista de usuarios que se carga al llamar getUsers si es que no han sido inicializados

```
public class MyViewModel extends ViewModel {
  private MutableLiveData<List<User>> users;
  public LiveData<List<User>> getUsers() {
    if (users == null) {
      users = new MutableLiveData<List<User>>();
      loadUsers();
    }
    return users;
  }

  private void loadUsers() {
    // Do an asynchronous operation to fetch users.
  }
}
```

La actividad principal le pide a ViewModelProviders que le entregue una instancia de MyViewModel que fue creada la primera vez que se llama a onCreate en el Activity. Al recrear la actividad se recibe la misma instancia de MyViewModel creada al principio incluyendo su estado.

De esta forma, al recrear el Activity la información es accesible sin tener que pedir nuevamente la lista de usuarios.

```
public class MyActivity extends AppCompatActivity {
    public void onCreate(Bundle savedInstanceState) {
        // Create a ViewModel the first time the system calls an activity's
onCreate() method.
        // Re-created activities receive the same MyViewModel instance created
by the first activity.
        MyViewModel model = ViewModelProviders.of(this).get(MyViewModel.class);
        model.getUsers().observe(this, users -> {
            // update UI
        });
    }
}
```

Ciclo de vida de ViewModel

ViewModel es creado durante la creación de la actividad (*onCreate(...)*) o fragmento (*onCreateView(...)*) y su alcance es transversal al ciclo de vida de la vista.

Cuándo la actividad es finalizada luego de ejecutar *onDestroy()*, el framework llama al método *onCleared()* de la clase ViewModel para limpiar los recursos.

El ViewModel queda en memoria hasta que el ciclo de vida del Activity sea finalizado. Lo mismo es aplicable cuando se trata de Fragments.

El diagrama que sigue muestra el ciclo de vida de Activity y de ViewModel en forma paralela.

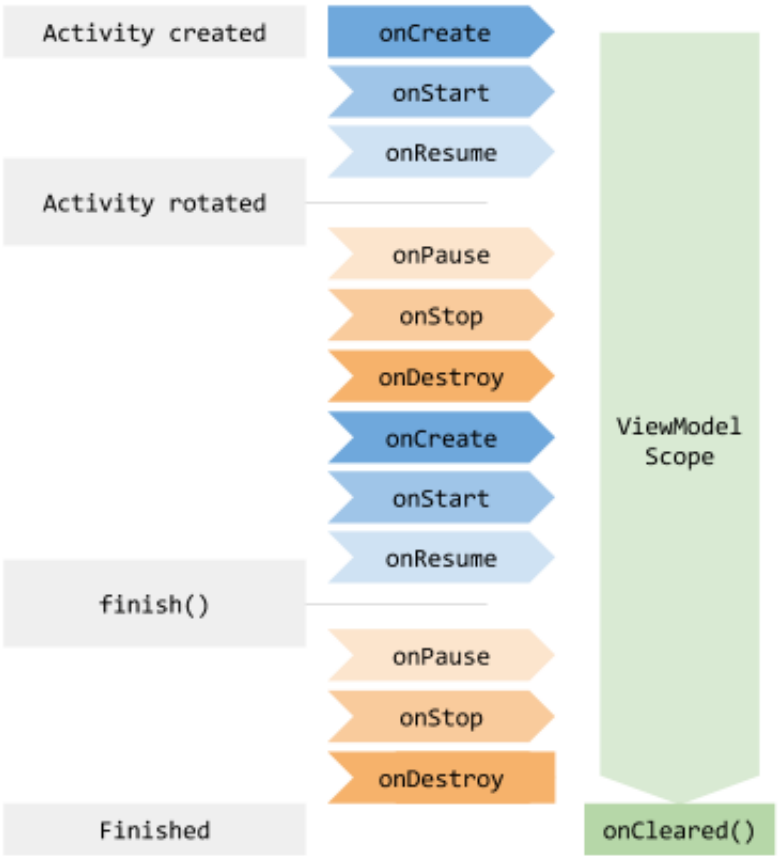

Imagen 2. Ciclo de vida de Activity y ViewModel.

La primera vez que se pida el ViewModel en el onCreate() del Activity se va a crear la instancia, y cada vez que se vuelva a llamar a onCreate() se usará el mismo ViewModel.

Para esto, las clase ViewModelProviders permite acceder a una única instancia de ViewModel que es compartida a través de toda la aplicación.

```
MyViewModel viewModel = ViewModelProviders.of(this).get(MyViewModel.class);
```

La clase ViewModel no es dependiente del ciclo de vida de la aplicación, por lo que no tiene contexto de la aplicación. Esto genera un problema al querer usar algún recurso, como por ejemplo los strings definidos en strings.xml o colores definidos en colors.xml.

Para solucionar esto se puede exponer directamente el ID del recurso para que sea actualizado por la vista. Por ejemplo, si se necesita cambiar el color de alguna de las vistas, el viewModel puede exponer el color usando LiveData y el ID del color a utilizar.

```
public class MyViewModel extends ViewModel {

    private MutableLiveData<Integer> myColor = new MutableLiveData<>();

    public MyViewModel() {
        Timber.d("MyViewModel() called");

        myColor.setValue(Color.RED);
    }
}
```

y en el layout queda enlazado directamente

```
<TextView
    android:id="@+id/textView"
    ...
    android:background="@{myViewModel.strengthColor}"/>
```

Conceptos asociados

Para la implementación del patrón MVVM en Android hay algunas bibliotecas y clases que permiten lograr la comunicación entre el ViewModel y la vista.

LiveData es:

- **Data holder**: Mantiene los datos sin necesidad de tener que pedirlos nuevamente, aún si el Activity cambia de estado (pero sin destruirse). Por ejemplo, al rotar la pantalla, el viewModel mantiene los datos y la actividad los utiliza para la nueva posición
- **Observable**: El viewModel no tiene una referencia directa a la Actividad o Fragment, si no que la Actividad observa los cambios en el viewModel
- **Lifecycle-aware**: Crea y destruye las referencias (Observable) según el ciclo de vida de la actividad en forma automática y solo en los métodos onCreate() y onDestroy(), lo que permite que ante un cambio en la rotación de la pantalla, éstas referencias se mantienen

  Generalmente, LiveData llama a los observadores con actualizaciones sólo cuando la información cambia. Un ejemplo concreto de LiveData como referencia en un ViewModel

```
public class NameViewModel extends ViewModel {

  // Create a LiveData with a String
  private LiveData<String> currentName;

  public LiveData<String> getCurrentName() {
    if (currentName == null) {
    currentName = new LiveData<String>();
    }
    return currentName;
  }

// Rest of the ViewModel...
}
```

Para poder observar los cambios, la actividad se registra como observador de los cambios.

```
public class NameActivity extends AppCompatActivity {

    private NameViewModel model;
    private TextView nameTextView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Other code to setup the activity…

        // Get the ViewModel using ViewModelProviders
        model = ViewModelProviders.of(this).get(NameViewModel.class);

        // Create the observer which updates the UI.
        final Observer<String> nameObserver = new Observer<String>() {
            @Override
            public void onChanged(@Nullable final String newName) {
                // Update the UI, in this case, a TextView.
                nameTextView.setText(newName);
            }
        };

        // Observe the LiveData, passing in this activity as the LifecycleOwner
and the observer.
        model.getCurrentName().observe(this, nameObserver);
    }
}
```

Cada vez que se actualice en el viewModel el nombre, se notifica al observador registrado por la actividad para actualizar el textView con el nuevo nombre.

Implementando MVVM en TicTacToe

ViewModel ocupa el lugar del presentador y suma una capa extra que es manejada completamente por la biblioteca DataBinding asociada al concepto de Binder.El diagrama de clases queda de esta forma

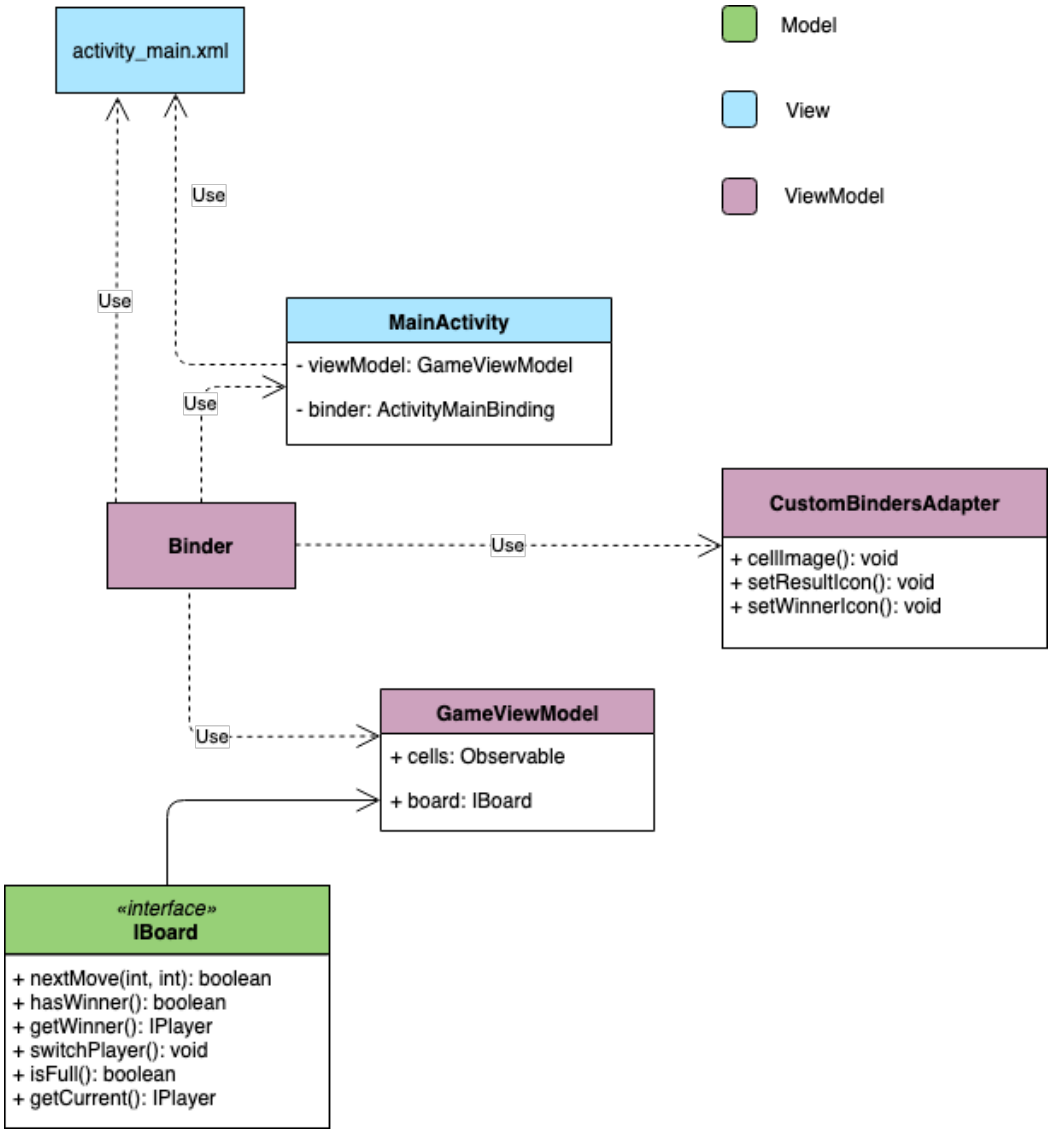

Imagen 3. Diagrama de clases.

El layout para la actividad principal mantiene las mismas vistas, pero suma 3 diferencias importantes

1. Usa DataBinding (agrega un layout como *root*) y define una variable de tipo GameViewModel
2. Le indica a cada celda el manejador del evento onClick()
3. Crea el atributo app:cell_icon que se encarga de desplegar la imagen de cada celda

El evento onClick() queda asociado a un método de la clase GameViewModel. A diferencia del manejo del evento onClick sin DataBinding, este manejador es evaluado en tiempo de compilación y no de ejecución, así que se sabe de antemano si existe un problema con el layout evitando que se caiga la app.

```
<?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">

  <data>
    <variable
      name="myViewModel"
      type="cl.desafiolatam.tictactoe.viewmodel.GameViewModel" />
  </data>

  <androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@drawable/background_screen"
    android:orientation="vertical">

    <GridLayout
      android:id="@+id/main_grid_layout"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_row="3"
      android:layout_column="3"
      android:background="@drawable/background_grid"
      …      >

      <ImageButton
        android:id="@+id/cell_0_0"
        style="@style/cell"
        android:layout_row="0"
        android:layout_column="0"
        android:contentDescription="@string/default_content_description"
        android:onClick="@{() -> myViewModel.onClickedCellAt(0,0)}"
        app:cell_icon='@{myViewModel.cells["00"]}'
        app:srcCompat="@drawable/background_default_cell" />

      ...
    </GridLayout>

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

En nuestro caso, el ViewModel puede enviar información al layout y también observar sus cambios. Para esto, se necesita usar la clase BindingAdapter y definir un atributo personalizado en el XML

El BindingAdapter escucha por los cambios en el atributo y actualiza su valor. Por ejemplo, **cell_icon** recibe el jugador asociado a la celda y actualiza la imagen según el parámetro

```
public class CustomBindingAdapters {
   …

   @androidx.databinding.BindingAdapter("cell_icon")
   public static void cellImage(final ImageView imageView, int player) {
       if (player > 0) {
           imageView.setImageResource(player == 1 ? R.drawable.ic_player_1 :
R.drawable.ic_player_2);
       }
   }
```

El BindingAdapter se genera al utilizar la anotación @androidx.databinding.BindingAdapter("cell_icon") con el nombre del atributo a utilizar en el layout

ViewModel para TicTacToe

El ViewModel implementado debe heredar de la clase ViewModel y se encarga de utilizar el modelo y actualizar la vista.

```
package cl.desafiolatam.tictactoe.viewmodel;

import ...

public class GameViewModel extends ViewModel {

   public ObservableArrayMap<String, Integer> cells;

   private IBoard board;

   private MutableLiveData<IPlayer> winner = new MutableLiveData<>();
```

```
    public void init() {
        IPlayer player1 = new Player("Player 1", IPlayer.TURN_PLAYER_1);
        IPlayer player2 = new Player("Player 2", IPlayer.TURN_PLAYER_2);

        board = new Board(player1, player2);
        cells = new ObservableArrayMap<>();
    }

    public void onClickedCellAt(int row, int column) {
       if (this.board.nextMove(row, column)) {
          cells.put(row + "" + column,
 board.getCurrentPlayer().getPlayerTurn());

          if(board.hasEnded()) {
             winner.setValue(board.getWinner());
           }else {
             board.switchPlayer();
          }
       } else {
          Timber.d("Invalid move. Cell already used");
       }
    }

    public LiveData<IPlayer> getWinner() {
        return winner;
    }

    public void onReset() {
        board.newGame();
        cells.clear();

        winner = new MutableLiveData<>();
    }
 }
```

Pros	**Pruebas: ViewModel no está atado a ninguna vista y es posible probarlo usando tests unitarios. Para probar, basta con verificar que las variables observables son correctamente asignadas cuando cambia el modelo.**
Contras	Mantenimiento: El layout XML no está probado. Hay que tener cuidado en no agregar mucha lógica y tratar de mantenerlo simple para no tener comportamientos extraños. Los valores siempre deben venir del ViewModel más que calculados en la vista.

Comparativa con MVP

- ViewModel reemplaza a la capa del Presentador
- El Presentador mantiene una referencia a la vista. ViewModel no la tiene
- El Presentador actualiza la vista invocando callbacks
- ViewModel envía flujos de datos
- El presentador y la vista tiene una relación de 1 a 1
- La vista y el ViewModel tienen una relación de 1 a muchos
- El ViewModel no conoce a la vista que escucha sus eventos

Referencias

Si estás interesado en ver más ejemplos y explorar más los patrones MVP y MVVM, Google Architecture Blueprints Project es un muy buen comienzo para revisar distintos tipos de acercamiento a la arquitectura para construir apps.

Este artículo de androiddevelopers también tiene un ejemplo simple de ViewModel enfocado en el ciclo de vida del Activity que está bien acabado.

Happy coding!