{desafío} latam_

# Patrones de diseño y callbacks - Parte I

## Patrones de diseño

### Competencias

- Conocer qué son los patrones de diseño
- Identificar patrones comunes
- Conocer qué son los anti-patrones de diseño

### Motivación

En el libro *A Pattern Language* :

*"Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo ni siquiera dos veces de la misma forma."*

- Christopher Alexander

Christopher es un arquitecto reconocido por sus diseños centrados en las personas, con la premisa de que los usuarios de los espacios arquitectónicos saben más que los arquitectos sobre el tipo de construcción que necesitan. Con sus teorías ha influido en áreas más allá de la arquitectura, incluyendo el desarrollo urbano, sociología y software.

Su libro fue tomado como base en el año 1994 por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, conocidos como *Gang of Four* **(**GoF**),** para su libro *Design Patterns: Elements of Reusable Object-Oriented Software* donde identifican 23 patrones clásicos detallados por su nombre, clasificación, motivación, estructura, solución, consecuencias, entre otros.

Ahí indican que la meta era capturar experiencias de diseño en un formato que pudiera ser utilizado efectivamente, permitiendo hacer mucho más fácil reutilizar diseños y arquitecturas exitosas, además de hacerlo más accesible para los nuevos desarrolladores.

*Put simply, design patterns help a designer get a design "right" faster.“*

- GoF

Los patrones de diseño de software son técnicas para resolver problemas comunes en el desarrollo y en el diseño de las interacciones de los elementos. Para que sea aceptado como patrón tiene que haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores.

Por otro lado, cuando un código es escrito sin utilizar patrones de diseño para solucionar problemas comunes, se convierte en un enigma por descifrar para cada persona que mire el código, incluso para la persona que lo escribió luego de un tiempo

## Ventajas de utilizar patrones de diseño

- Facilitar el aprendizaje de las nuevas generaciones condensando conocimiento ya existente.
- Código estructurado, donde la forma la entrega el patrón de diseño
- Tener un catálogo de elementos reusables o patrones para aplicar en problemáticas conocidas.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Tener un punto común de comunicación, formalizando un vocabulario común, es más simple indicar que se usa el patrón de diseño Observer y no tener que definir en detalle cómo implementar una clase que observe y responda a eventos.
- Los patrones de diseño pueden incluso mejorar la documentación y el mantenimiento de los sistemas existentes al proporcionar una especificación explícita de las interacciones de clase y objeto y su intención subyacente.

## Los patrones de diseño no buscan

- Imponer alternativas de diseño frente a otras
- Eliminar la creatividad inherente al proceso de diseño
- Solucionar todos los problemas

No existe obligación en utilizar los patrones de diseño, pero es altamente recomendable utilizarlos al enfrentarse a un problema que ha sido descrito por algún patrón por las ventajas mencionadas anteriormente, pero sin perder de vista que los patrones no describen todas las posibles soluciones y que es necesario analizar las distintas alternativas manteniendo un espíritu crítico sobre cada una de ellas, hasta encontrar la más adecuada

# Casos de estudio

A continuación, se definen 3 casos de estudio, donde a partir de una problemática, se elabora una posible solución para luego, identificar un patrón que describa la solución

## Caso de estudio 1

### Problemática

Se dispone de una serie de listas de distinto tipo, como `LinkedList<Integer>` o `ArrayList<Integer>`, que deben ser ordenadas en tiempo de ejecución dentro de la misma lista.

### Solución propuesta

1. Definir el comportamiento en una interfaz para separar la definición de la implementación.

```
interface SortingStrategy {
  List<Number> sort(List<Number> array);
}
```

2. Definir una familia de clases que implementen la interfaz para hacerlas intercambiables. Todas las clases tienen la misma firma del método sort, pero lo implementan de forma distinta

```
class HeapSort implements SortingStrategy {
  public List<Number> sort(List<Number> array) {
    // implementation here
    ...
  }
}

class BubbleSort implements SortingStrategy {
  public List<Number> sort(List<Number> array) {
    // implementation here
    ...
  }
}

class QuickSort implements SortingStrategy {
  public List<Number> sort(List<Number> array) {
    // implementation here
    ...
  }
}
```

3. Ocultar el comportamiento a quien lo utiliza

```
class Sorting {
  private SortingStrategy sortingStrategy;
  private List<Number> array;

  public SortingProgram(List<Number> array) {
    this.array = array;
  }

  public List<Number> runSort(SortingStrategy sortingStrategy) {
    return this.sortingStrategy.sort(this.array);
  }
}
```

Sorting no conoce la implementación real del ordenamiento, sólo hace uso del método **sort()** proporcionando al objeto concreto para el ordenamiento

```
Sorting sortingProgram = new Sorting(Arrays.asList(5, 6, 10, 12, 7, 9, 6, 3));
sortingProgram.sort(new BubbleSort());
```

**Ejemplo**

La clase Collections tiene exclusivamente métodos estáticos para operar sobre las listas. Contiene algoritmos polimórficos que operan sobre las colecciones o retornan una nueva instancia dada una específica.

Por ejemplo, los métodos estáticos sort(...) y max(...) operan sobre cualquier clase que implemente la interfaz.

```
List<Integer> list = Arrays.asList(5, 6, 10, 12, 7, 9, 6, 3);
List<Integer> linkedList = new LinkedList<>(list);
```

Estas dos implementaciones de la interfaz *List* pueden ser utilizadas como parámetro permitiendo abstraerse de las particularidades de la implementación

| **Código** | **Salida** |
| --- | --- |
| Collections.sort(list);<br>list.forEach(n -> print(n + ", "));<br> | 3, 4, 5, 6, 7, 9, 10, 11, 12 |
| Collections.sort(linkedList);<br>linkedList.forEach(n -> print(n + ", "));<br> | 3, 4, 5, 6, 7, 9, 10, 11, 12 |
| int max = Collections.max(list);<br>print(max)<br> | 12 |
| int max = Collections.max(list);<br>print(max)<br> | 12 |

**Patrón de diseño Strategy**

Este comportamiento descrito para la clase **Collections** está definido por el patrón de diseño **Strategy** (Estrategia) que es parte de los patrones de diseño de **comportamiento**.

Los patrones de comportamiento ofrecen soluciones respecto a la interacción y responsabilidades entre clases y objetos, así como los algoritmos que encapsulan.

## Caso de estudio 2

### Problemática

En una aplicación que interactúa con distintas fuentes de datos, se debe verificar que cada parámetro entregado a un método debe ser instanciado antes de utilizarlo

### Solución propuesta

1. Encapsular el comportamiento común en una clase, encapsulando la verificación de nulidad
2. Usar la definición en cada elemento que lo necesite

### Ejemplo

Android provee una librería de anotaciones en la que se incluye @**NonNull**, que es aplicable al parámetro de un método para ser analizado en tiempo de compilación y entregar un *warning* en caso de que se pase un valor posiblemente *null.*

### Patrón de diseño Decorato

Este comportamiento está definido por el patrón de diseño **Decorator** (Decorador) que es parte de los patrones **Estructurales** que por definición solucionan problemas de composición (agregación) de clases y objetos.

## Caso de estudio 3

### Problemática

Dentro de una aplicación que consume datos en forma masiva, el acceso a información en forma recurrente ha sobrecargado los tiempos de respuesta de la app, por lo que se cree que agregando un sistema de caching podría aliviar el problema.

Un sistema de caching o caché se encarga de almacenar datos para que las solicitudes futuras sean respondidas más rápido

Crear un caché permite mantener una instancia para evitar crear / eliminar objetos bajo mucha demanda, y así solventar el consumo masivo de éstas instancias

### Solución propuesta

1. Una clase que es responsable de crear una única instancia
2. Asegurar la instancia con un constructor privado
3. Permitir el acceso global a dicha instancia mediante un método de clase.

### Ejemplo

La clase Unique tiene un constructor privado, por lo que no puede ser inicializado directamente y define una instancia única como atributo estático.

La única forma de acceder a una instancia de Unique es llamar a getInstance() para utilizar los métodos de la clase

```
public class Unique {
  private static final Unique INSTANCE = new Unique();

  // El constructor privado no permite que se instancia la clase
  private Unique() {}

  // Solo es posible ocupar la instancia previamente creada
  public static Unique getInstance() {
    return INSTANCE;
  }    ....
}
```

## Patrón de diseño Singleton

Este comportamiento está definido por el patrón **Singleton** que es parte de los patrones **Creacionales**.

Esta familia de patrones están enfocados en solucionar problemas de creación de instancias, ayudando a encapsular y abstraer dicha creación

## Clasificación de los patrones

Los 3 patrones ejemplificados en las situaciones anteriores, describen la clasificación de los patrones según su propósito

| Clasificación | Patrón |
| --- | --- |
| **Creacionales**<br>Buscan ocultar la forma en que la objeto se crea e inicializa y, por otra parte, definen la forma de instanciar un objeto | Factory Method<br>AdapterAdapter<br>Builder<br>Prototype<br>**Singleton** |
| **De comportamiento**<br>La mayoría de estos patrones son sobre la comunicación entre objetos | Interpreter<br>Template Method<br>Chain of responsability<br>Command<br>Iterator<br>Mediator<br>Memento<br>Observer<br>State<br>**Strategy**<br>Visitor |
| **Estructurales**<br>Explican cómo ensamblar clases para construir una estructura más grande y mantenerlas flexibles y eficientes | Adapter<br>Bridge<br>Composite<br>**Decorator**<br>Facade<br>Flyweight<br>Proxy |

# Antipatrones

Los antipatrones son malas prácticas documentadas y son una recomendación de las cosas que no se debería hacer. Describe una solución comúnmente aplicada a un problema que genera consecuencias negativas como constante

Mientras que los patrones ayudan a identificar, diseñar e implementar código funcional, los antipatrones de diseño hacen exactamente lo contrario. En el libro Antipatterns Refactoring Software, Architectures, and Projects in Crisis, se define una lista de 40 antipatrones en las áreas de desarrollo de software, arquitectura y manejo de proyectos

# Antipatrones comunes

De esos 40 antipatrones, hay 4 que lamentablemente son demasiado comunes

**Swiss Army Knife**: Una navaja suiza es una clase o interfaz excesivamente compleja que intenta anticiparse a cada posible escenario.

Para solucionarlo se debe definir un propósito claro para el componente y crear métodos más generales que faciliten su utilización

**Reinvent the Wheel**:

Que cada aplicación que se desarrolle esté aislada del resto de las aplicaciones, convirtiéndose en *silos* e implementando la misma funcionalidad en cada una de las aplicaciones. Esto tiene consecuencias negativas como atraso en el *time-to-market* del producto, mayores costos de desarrollo y baja mantenibilidad

Se puede solucionar definiendo una interfaz común que generalice su utilización para utilizar en todas las aplicaciones

**Golden Hammer**: Es una tecnología familiar o un concepto aplicado obsesivamente a varios problemas, aunque sea a la fuerza.

Hay que tener cuidado de no tener solo un martillo y ver todo como clavo, debe primar por sobre todo un análisis de las posibles soluciones utilizando la mayor cantidad de información disponible.

La solución a este antipatrón incluye expandir el conocimiento de los desarrolladores mediante la educación, entrenamiento y grupos de estudio para exponer a los desarrolladores alternativas de tecnología y acercamiento a una solución.

**Cut–and–Paste Programming**: La reutilización de código copiando trozos lleva a problemas importantes de mantención.

Alternativamente, convertirlo en código que pueda ser utilizado por distintos participantes, reduce los problemas de mantención dado que existe un código común, que puede ser *testeado* y documentado

## Referencias para profundizar

Los patrones de diseño son un tema muy amplio e interesante de analizar. Ayudan a ampliar la visión sobre cómo afrontar situaciones revisando las experiencias pasadas que están documentadas y categorizadas.

Han pasado varios años desde la definición del concepto y varios patrones ya están desactualizados, en su mayoría incorporados dentro de los distintos lenguajes y herramientas, y también hay nuevos patrones que son parte de las nuevas tendencias y tecnologías.

Las bases quedan sentadas para descubrir más sobre patrones revisando alguno de estos libros

- Design Patterns: Elements of Reusable Object-Oriented SoftwareErich Gamma, Ralph Johnson, John Vlissides, Richard Helm
- Head First Design Patterns Eric Freeman, Kathy Sierra, Bert Bates, Elisabeth Robson
- Antipatterns Refactoring Software, Architectures, and Projects in CrisisWilliam J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick

# Modelo y Vista

## Competencias

- Conocer la capa de Modelo
- Entender las responsabilidades de la capa de modelo
- Agregar y probar test unitario para el modelo
- Conocer la capa de Vista
- Entender las responsabilidades de la capa de vista

## Introducción

**¿Qué sucede cuando abrimos una aplicación como Spotify o Netflix?**

La pantalla de la app es la responsable de interactuar con el usuario, recibiendo instrucciones y presentando información.

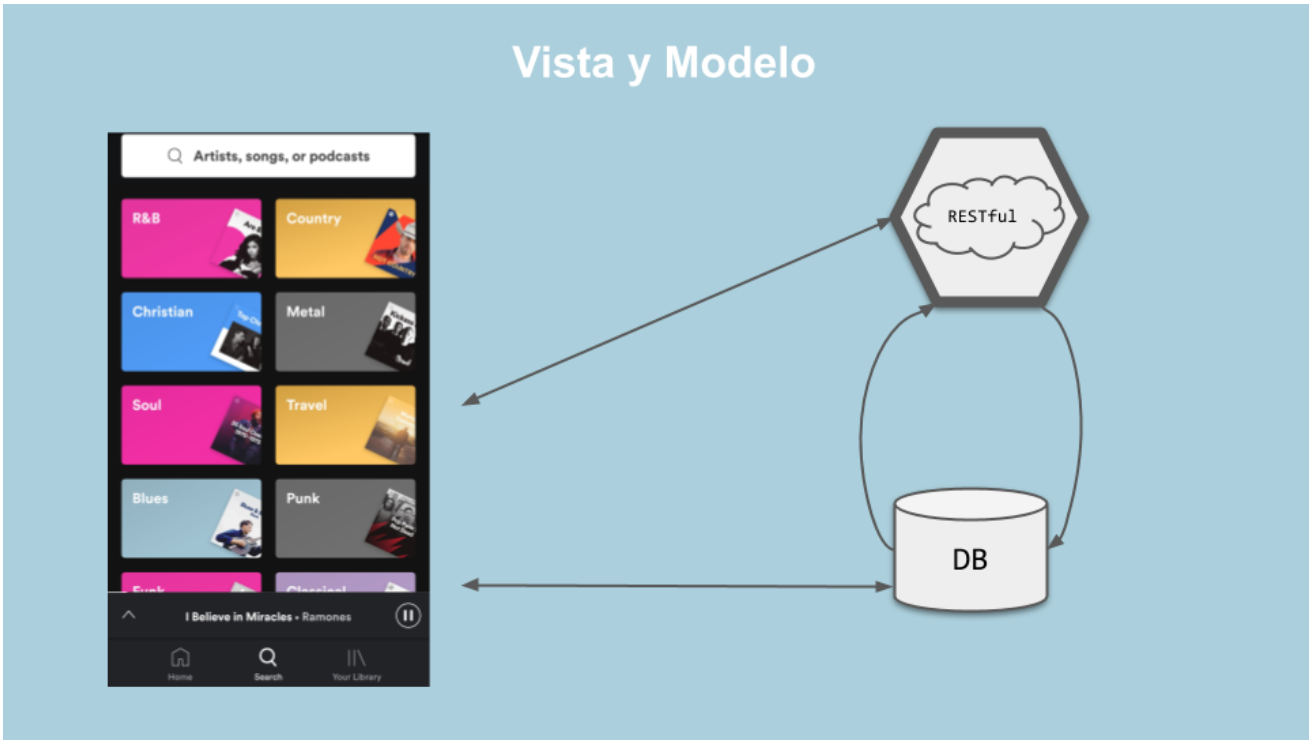


Imagen 1. Vista - Modelo

**¿Pero de dónde viene esa información?**

Proviene de alguna fuente de datos, en particular una API alojada en algún servidor remoto, en otras palabras, el **backend** de la aplicación

Esta separación de responsabilidades se aplica al diseño interno de una aplicación, definiendo 2 capas de abstracción:

1. La UI es la responsable de mostrar información e interactuar mediante eventos con el usuario
2. El modelo es responsable del negocio, el estado y los datos

A través de los años son varios los patrones de diseño que usan esta división y además agregan una capa extra que se encarga de interactuar entre ambas, utilizándose como piezas independientes que se juntan para armar la app.

Esta capa intermedia permite mantener separadas las capas de Vista (*User Interface*) del modelo (Model) para **evitar dependencias** entre ambas, y permite principalmente:

- Separar responsabilidades (desacoplar el código)
- Escribir código reutilizable
- Escribir código fácil de probar (testing)
- Encapsular responsabilidades

La capa que une la vista y el modelo es la que varía en los 3 patrones de diseño más utilizados en el desarrollo de aplicaciones para Android

1. Model - View - **Controller** (MV**C**)
2. Model - View - **Presenter** (MV**P**)
3. Model - View - **ViewModel** (MV**VM**)

Los patrones MVC y MVP fueron propuestos por Martin Fowler en la guía de arquitectura de interfaces visuales (GUI architectures hace algún tiempo y han sido incorporadas como buenas prácticas en el desarrollo de aplicaciones Android.

Por su parte MVVM está siendo impulsado por Google para el desarrollo de apps nativas como parte de JetPack.

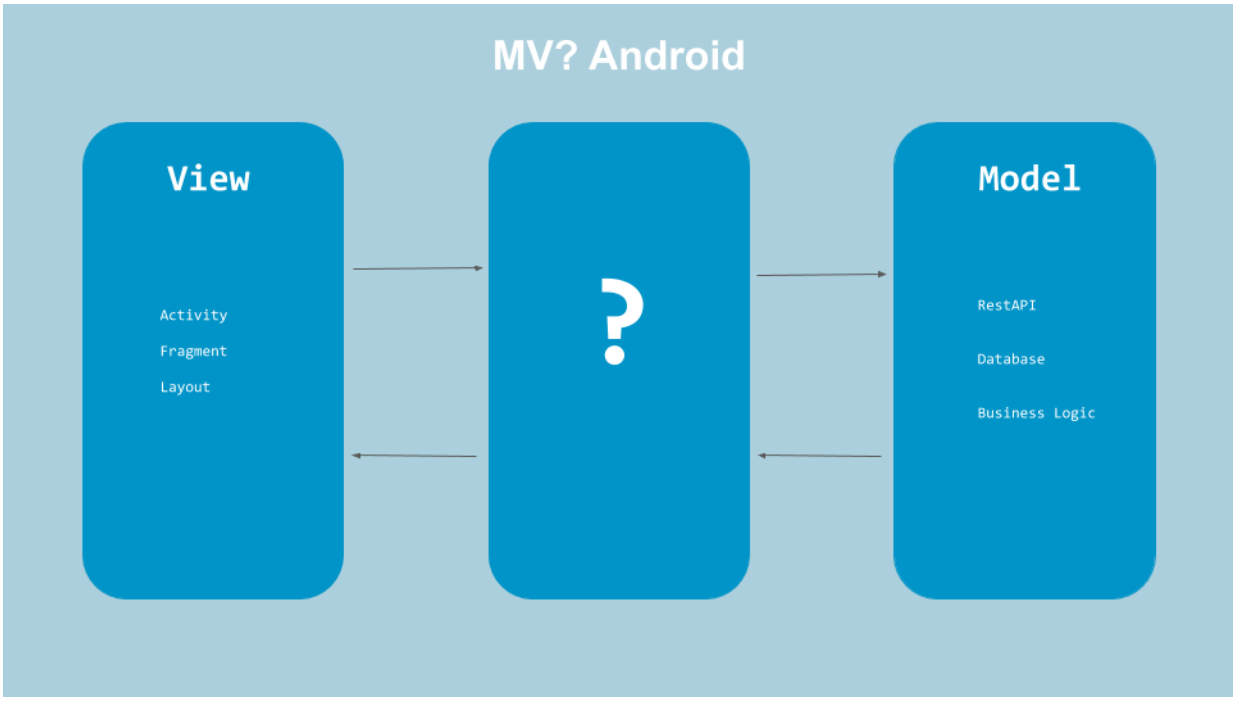


Imagen 2. MV_.

Profundizaremos sobre la parte común de los 3 patrones: **Model - View**, que son la base para poder entender y aplicar los patrones de diseño nombrados.

## Modelo (Model)

El modelo corresponde a un conjunto de clases que se encargan de la **lógica del negocio**, tanto del modelo de negocios como de los **datos** que lo compone, además de las reglas de cómo los datos son manipulados y maneja su propio **estado**.

A este nivel no hay referencias de cómo la información va a ser desplegada o de cómo manejar los eventos originados por el usuario. Es solo una caja negra que entrega información y reacciona a eventos definidos

En forma aún más específica, el modelo no debería tener ninguna referencia a Android, siendo compuesto por POJOs (Plain Old Java Objects) sin referencia alguna al SDK de Android.

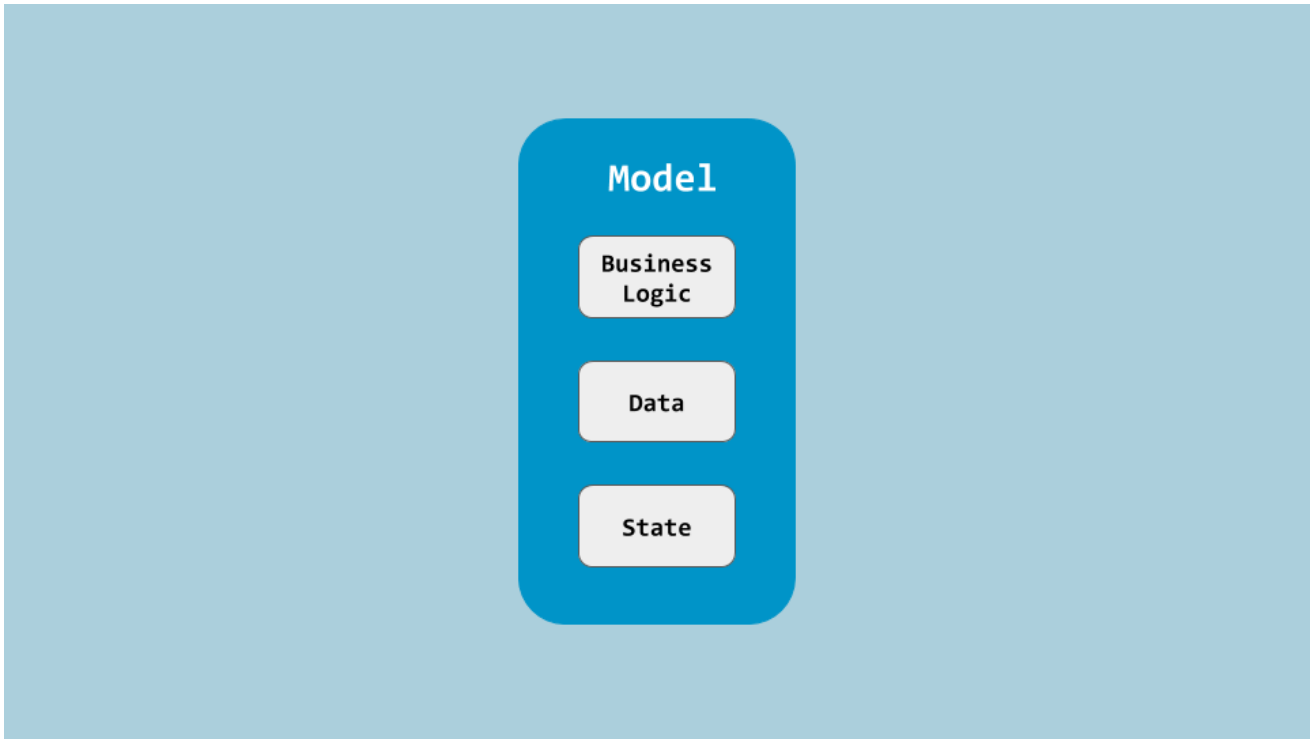


Imagen 3. Componentes.

El modelo se puede considerar casi como un módulo externo de tal modo que la implementación queda encapsulada, y es desarrollado buscando separarlo de las otras partes de la aplicación.

El modelo debe ser probado en forma separada sin importar el patrón de diseño utilizado, entonces , el modelo:

- Debe entregar una interfaz que sea fácilmente *mockeable* para facilitar las pruebas como un módulo externo
- NO debe tener referencias a la vista
- Debe ser probado en forma independiente

## Modelo para TicTacToe

Entonces, ¿cómo se refleja esto en el diseño de la app?

Para analizar un ejemplo práctico tenemos una app para jugar TicTacToe(*Gato* o *3 en línea*)

La pantalla principal está compuesta por un tablero de 3x3. Cuando un jugador elige una de las 9 posiciones disponibles al empezar el juego, la celda es creada para ese jugador y luego se agrega a la matriz en la posición indicada quedando ocupada durante el juego

Por cada casilla ocupada se revisa todo el tablero buscando 3 elementos en línea para el mismo jugador, analizando las diagonales, las filas y las columnas. Si existe un ganador se debe indicar que el jugador es el ganadorEl tablero además debe saber si está completo y tener un registro del jugador actual.

El modelo está compuesto por un tablero (**Board**) compuesto por una matriz de celdas (**Cell**) de 3x3 donde cada celda tiene 1 jugador asociado (**Player**)

El diagrama de clases aborda las interfaces, clases y relaciones es el siguiente

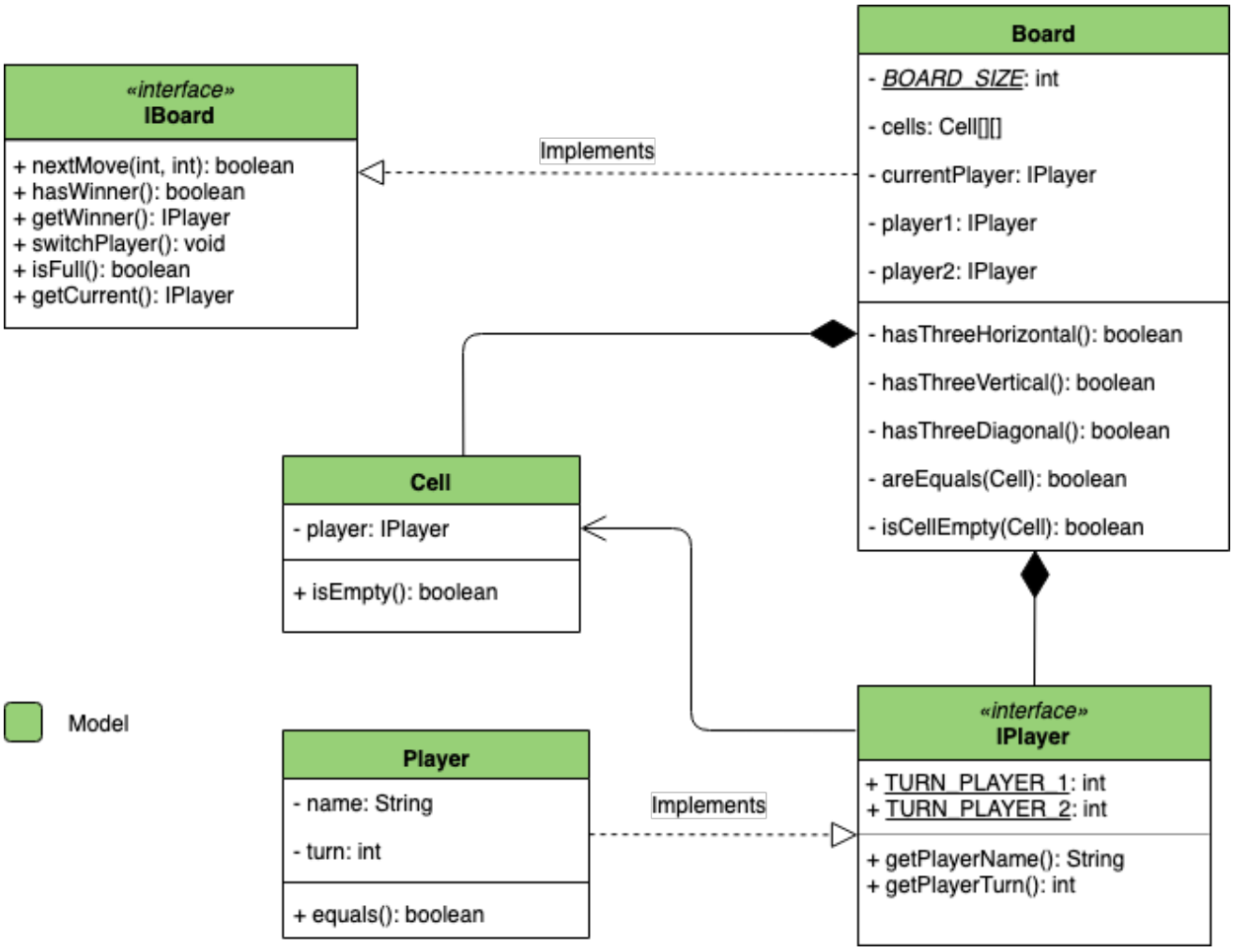


Imagen 4. Diagrama de clases

En el diagrama de clases se incluyen 2 interfaces: IBoard, IPlayer.

**IBoard** es el **contrato** con el que se expone el tablero, o sea, el punto de comunicación hacia la capa del modelo y es el que contiene la declaración para interactuar con las clases Cell y Player

Entonces, Board es la encargada de implementar las funcionalidades definidas en la interfaz IBoard y además tiene algunos métodos privados que encapsulan algunos comportamientos como revisar si existe un ganador en forma horizontal, vertical o diagonal.

**¿Por qué exponer sólo la interfaz IBoard?**

Al exponer solo la interfaz, se desliga completamente el “cliente” del modelo de la implementación concreta del modelo. Con esto, se puede intercambiar el modelo por cualquier que implemente la interfaz IBoard, por ejemplo, un objeto *mock* para hacer pruebas o desarrollar un nuevo modelo que reemplace al actual sin afectar a los clientes del modelo

Suena cool? Bueno, lo es!

La interfaz **IPlayer** describe a un jugador con su nombre y si juega primero o segundo. Nuevamente la intención de la interfaz es separar al jugador concreto, dado que se utiliza tanto en la celda como en el tablero es preferible mantener una referencia al contrato más que a la implementación. No así celda que es solo utilizada por el tablero.

La clase Board es el modelo concreto que implementa la interfaz IBoard

```
public class Board implements IBoard {

    private static final int BOARD_SIZE = 3;

    @VisibleForTesting
    protected Cell[][] cells;
    @VisibleForTesting
    protected IPlayer currentPlayer;
    private IPlayer player1;
    private IPlayer player2;

    public Board(@NonNull IPlayer player1, @NonNull IPlayer player2) {
        cells = new Cell[BOARD_SIZE][BOARD_SIZE];
        this.player1 = player1;
        this.player2 = player2;
        this.currentPlayer = player1;
    }

    /*********************************************
     ******* IBoard interface implementation ******
     *********************************************/
    /**
     * Set a cell if the location is available
     *
     * @return {@code true} if cell was assigned. {@code false} otherwise
     */
    public boolean nextMove(final int row, final int col) {
        if (!isCellNullOrEmpty(this.cells[row][col])) return false;

        Cell cell = new Cell(this.currentPlayer);
        this.cells[row][col] = cell;

        return true;
    }
```

```
    /**
     * @return {@code true} if one player has three cells in a row from
horizontal,
     * vertical or diagonal way
     */
    public boolean hasWinner() {
        return hasThreeHorizontal() || hasThreeVertical() || hasThreeDiagonal();
    }

    public IPlayer getWinner() {
        return currentPlayer;
    }

    /**
     * Change the turn between players
     */
    public void switchPlayer() {
        Timber.d("switchPlayer() called");
        this.currentPlayer = this.currentPlayer == player1 ? player2 : player1;
    }

    /**
     * @return {@code true} if all cells were assigned. {@code false} otherwise
     */
    public boolean isFull() {
        for (Cell[] row : cells) {
            for (Cell cell : row) {
                if (isCellNullOrEmpty(cell)) {
                    return false;
                }
            }
        }
        return true;
    }

    public int getCurrentTurn() {
        return currentPlayer.getPlayerTurn();
    }
}
```

La clase Board además tiene algunos métodos propios de la clase para aislar algunos comportamientos

```
public class Board implements IBoard {
    ...

     /*******************************************
     ************ Internal use methods ************
     *******************************************/

    /**
     * Look up for three cells for the same player in rows
     *
     * @return {@code true} for coincidence
     */
    @VisibleForTesting()
    protected boolean hasThreeHorizontal() {
        for (int i = 0; i < BOARD_SIZE; i++) {
            if (areEqual(cells[i][0], cells[i][1], cells[i][2])) {
                return true;
            }
        }
        return false;
    }

    /**
     * Look up for three cells for the same player in columns
     *
     * @return {@code true} for coincidence
     */
    @VisibleForTesting()
    protected boolean hasThreeVertical() {
        for (int i = 0; i < BOARD_SIZE; i++) {
            if (areEqual(cells[0][i], cells[1][i], cells[2][i])) {
                return true;
            }
        }
        return false;
    }
```

```
    /**
     * Look up for three cells for the same player in at least one of the
diagonals
     *
     * @return {@code true} for coincidence
     */
    @VisibleForTesting()
    protected boolean hasThreeDiagonal() {
        return areEqual(cells[0][0], cells[1][1], cells[2][2]) ||
                areEqual(cells[0][2], cells[1][1], cells[2][0]);
    }

    /**
     * 2 cells are equal if:
     * 1- Both have non null values
     * 2- Both have equal values
     *
     * @param cells: Cells to check if are equal
     * @return {@code false} if cells are empty or not equals to the first cell
     * @throws {@code NullPointerException} if cells is null. Please, do not
send null cells
     */
    @VisibleForTesting()
    protected boolean areEqual(@NonNull Cell... cells) {
        for (Cell cell : cells) {
            if (isCellNullOrEmpty(cell)) {
                return false;
            }
        }

        Cell comparisonBase = cells[0];
        for (int i = 1; i < BOARD_SIZE; i++) {
            if (!comparisonBase.equals(cells[i])) {
                return false;
            }
        }

        return true;
    }
```

```
    /**
     * A cell is empty if cell is null or actually empty
     *
     * @param cell to evaluate
     * @return {@code true} if null or empty
     */
    @VisibleForTesting()
    protected boolean isCellNullOrEmpty(Cell cell) {
        return cell == null || cell.isEmpty();
    }
}
```

Hasta este punto, el modelo está compuesto por un tablero representado por Board que expone a través de la interfaz IBoard para ser utilizados por algún cliente del modelo, pero ¿cómo se puede asegurar que el modelo funciona correctamente? o más allá, ¿cómo se verifica en forma determinística que el modelo se comporta correctamente?

## Probando el modelo

Es posible tratar el modelo como un módulo externo y empaquetado que provee cierta funcionalidad, pero que además puede “demostrar” su validez.

La estructura de la clase Board y la separación de responsabilidades para realizar tareas específicas permite la creación de tests unitarios y de esta forma, una validación del comportamiento en forma concreta. Además, es una excelente forma de desarrollar las funcionalidades agregando test unitarios para validar el código mientras se realiza.

Se utiliza la anotación `@VisibleForTesting()` para indicar que la visibilidad del método es más relajada de lo necesario para poder hacer las pruebas.

Al inicializar los test usando setUp() se crean 2 jugadores cada uno con una celda asociada.

```
public class BoardTest {

   private Board board;
   private Cell cell_1;
   private Cell cell_2;

   @Before
   public void setUp() {
       IPlayer player1 = new Player("May", IPlayer.TURN_PLAYER_1);
       IPlayer player2 = new Player("Aron", Player.TURN_PLAYER_2);

       cell_1 = new Cell(player1);
       cell_2 = new Cell(player2);

       board = new Board(player1, player2);
   }
   ...
}
```

El método nextMove asigna una celda a una posición dada por las coordenadas row, col si la posición está disponible, por lo que se crean 2 tests unitarios.

a. Asignar una celda a una posición libre

b. Asignar una celda a una posición ocupada

```
@Test
public void nextMove_ok() {
   assertTrue(board.nextMove(0, 0));
}

@Test
public void nextMove_nok() {
   int row = 0;
   int col = 0;
   board.nextMove(row, col);

   // cell cannot be assigned
   assertFalse(board.nextMove(row, col));
}
```

El método hasThreeHorizontal puede ser probado verificando:

a. Que existan 3 celdas en línea horizontal del mismo jugador

b. Que existan 3 celdas en línea horizontal de jugadores distintos

c. Que no existan 3 celdas en línea horizontal

```
@Test
public void hasThreeHorizontal_match() {
   board.cells[0][0] = cell_1;
   board.cells[0][1] = cell_1;
   board.cells[0][2] = cell_1;
   assertTrue(board.hasThreeHorizontal());
}
@Test
public void hasThreeHorizontal_different_player() {
   board.cells[0][0] = cell_1;
   board.cells[0][1] = cell_1;
   board.cells[0][2] = cell_2;
   assertFalse(board.hasThreeHorizontal());
}
```

```
@Test
public void hasThreeHorizontal_not_match() {
   board.cells[0][0] = cell_1;
   board.cells[0][1] = cell_1;
   assertFalse(board.hasThreeHorizontal());
}
```

De esta misma forma es posible probar hasThreeVertical() y hasThreeDiagonal() definiendo

El método areEqual debe probar que las celdas proporcionadas son iguales, independiente de la cantidad

```
@Test
public void areEqual_ok() {
   assertTrue(board.areEqual(cell_1, cell_1, cell_1));
}

@Test
public void areEqual_not_equal() {
   assertFalse(board.areEqual(cell_1, cell_2));
}
```

Al correr los tests, probamos en muy poco tiempo que los cambios introducidos al modelo no han cambiado el comportamiento ya comprobado

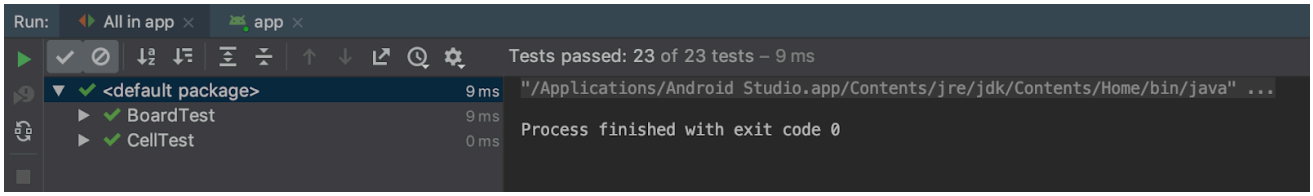


Imagen 5. Resultado al correr los test.

**Ejercicio**

Agregar tests unitarios al método *hasThreeDiagonal*() para cubrir los casos de ambas diagonales con celdas del mismo jugador.

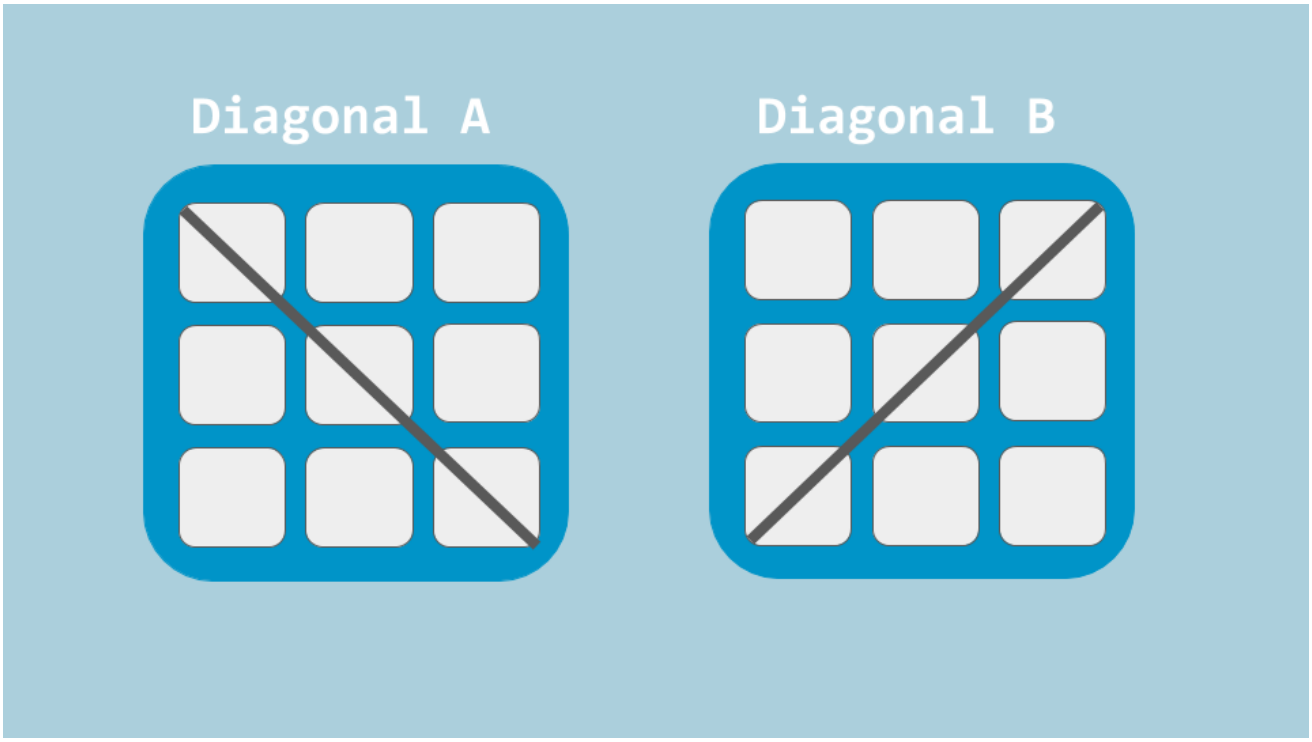


Imagen 6. Agregando test unitarios.

¿Cuáles otros casos pueden ser agregados para hasThreeDiagonal()?

**Solución**

1. Agregar un test para probar la diagonal A

```
@Test
public void hasThreeDiagonal_match_left_right() {
  board.cells[0][0] = cell_1;
  board.cells[1][1] = cell_1;
  board.cells[2][2] = cell_1;
  assertTrue(board.hasThreeDiagonal());
}
```

2. Agregar un test para probar la diagonal B

```
@Test

public void hasThreeDiagonal_match_right_left() {
  board.cells[2][0] = cell_1;
  board.cells[1][1] = cell_1;
  board.cells[0][2] = cell_1;
  assertTrue(board.hasThreeDiagonal());
}
```

3. Agregar un test para probar una diagonal con celdas de distintos jugadores

```
@Test
public void hasThreeDiagonal_different_player() {
  board.cells[0][0] = cell_1;
  board.cells[1][1] = cell_1;
  board.cells[2][2] = cell_2;
  assertFalse(board.hasThreeDiagonal());
}
```

4. Agregar un test para probar la respuesta si no hay diagonales

```
@Test
  public void hasThreeDiagonal_not_match() {
  board.cells[0][0] = cell_1;
  board.cells[1][1] = cell_1;
  assertFalse(board.hasThreeDiagonal());
}
```

## Vista (View)

La vista es una representación del modelo y tiene la responsabilidad de proveer la Interfaz de Usuario (*UI*). Tiene muy poca (o no existe) lógica y desconoce completamente la existencia del modelo y su estado.

Mientras menos conozca la vista sobre las otras 2 capas, tendrá menos acoplamiento por lo que se verá menos afectada por los cambios externos.

La vista en Android está compuesta por las layouts XML que define el diseño y las clases que presentan este diseño como Activity y Fragment. Este conjunto se define como la UI de la app y es la encargada de

- Recibir eventos gatillados por el usuario y propagarlos a la capa intermedia
- Presentar información en la pantalla de forma que el usuario la pueda consumir

## Vista para TicTacToe

Un tablero de TicTacTow o 3 en línea está compuesto por una matriz de 3x3 que corresponden a 3 filas y 3 columnas, resultando 9 casillas disponibles.

Participan 2 jugadores que juegan en sus respectivos turnos intercalados, primero el primer jugador, luego el segundo, que escogen una casilla tratando de ubicar 3 elementos en línea, ya sea horizontal, vertical o diagonal

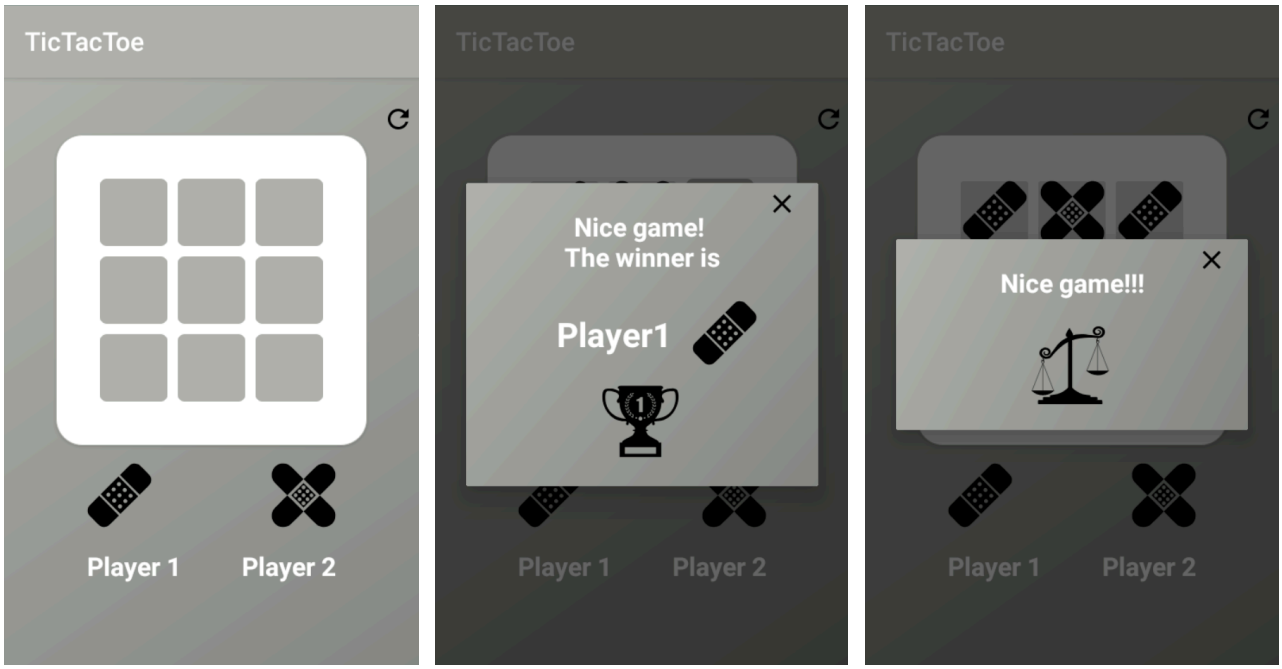


Imagen 7. Vistas de TicTacToe.

La vista está compuesta por la actividad principal (MainActivity) y una pantalla para mostrar el resultado (GameDialogFragment). MainActivity contiene el tablero y la información de los jugadores. Por su parte, el diálogo es sirve para mostrar el resultado del juego

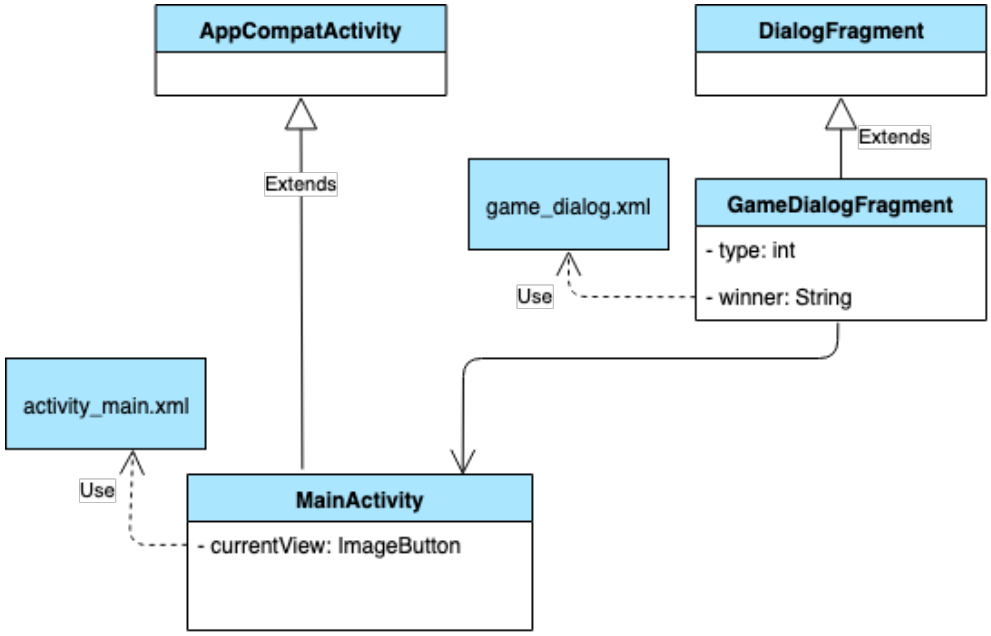


Imagen 8. Diagrama de la vista de TicTacToe

Para diseñar el tablero del juego que tiene las 9 casillas para jugar se usa GridLayout que se comporta como cuadrícula con filas y columnas definidas por `android:layout_row="3"` y `android:layout_column="3"`.

El resto de los elementos se ubican en forma relativa al elemento central que corresponde al GridLayout.

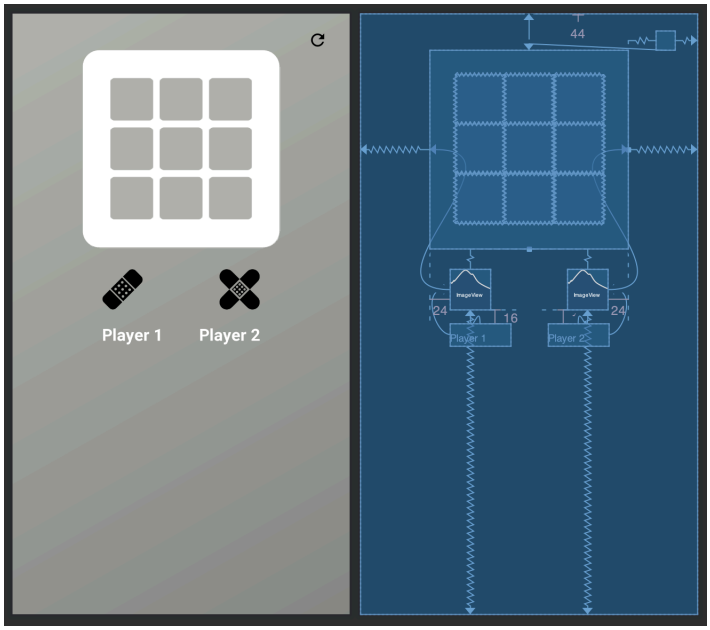


Imagen 9. Diseñando la vista con GridLayout.

Para mostrar el resultado de la partida se va a utilizar la clase GameDialogFragment que extiende de DialogFragment, que muestra una ventana flotante por sobre la actividad mostrando al ganador o empate

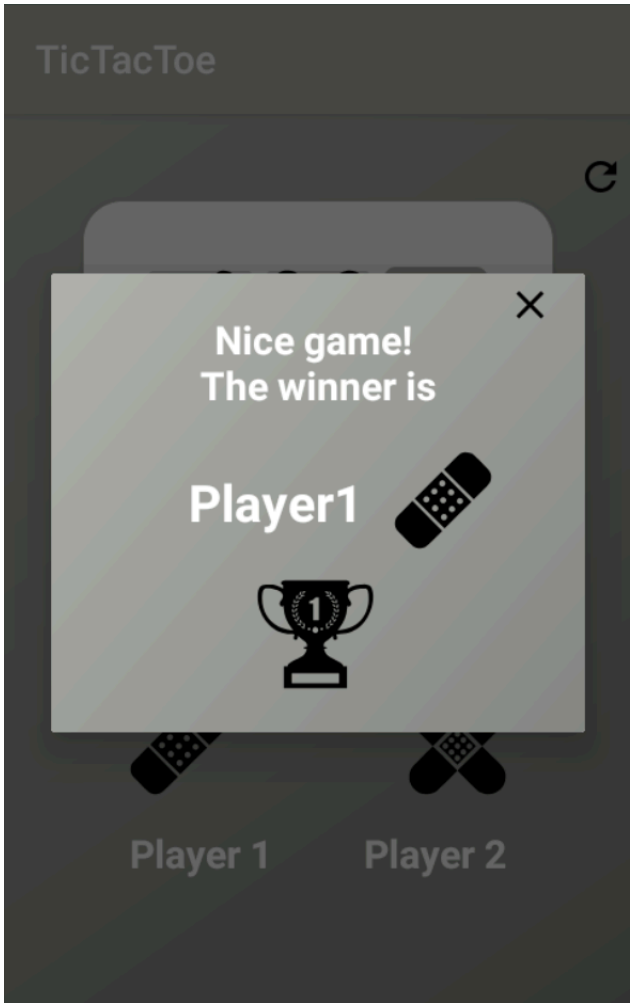


Imagen 10. Resultado utilizando DialogFragmet.

# Modelo - Vista - Controlador (MVC)

## Competencia

- Enlazar las capas de vista y modelo usando un controlador
- Conocer pros y contras de utilizar MVC

## Introducción

Martin Fowler describió el modelo MVC como parte de la guía de arquitectura de interfaces visuales y fue ampliamente usada en el desarrollo Android para separar la vista del modelo usando un **controlador** que sirve como el mediador entre ambas

Es un patrón de diseño enfocado en separar las responsabilidades dentro de la aplicación y es ampliamente utilizado en el desarrollo web por las ventajas que tiene respecto a no utilizar algún patrón. En tanto, en las apps Android fue el primer patrón utilizado y aún existen muchos proyectos que lo utilizan, con las ventajas y desventajas propias del patrón

## Características

Está enfocado en separar las responsabilidades: Separa las responsabilidades de la vista y del modelo, usando como “pegamento” el controlador, pudiendo probar el modelo independiente de la vista

Reutilizar código: Los cambios que se hagan al modelo no alteran el funcionamiento de la vista, pudiendo reutilizar elementos visuales o parte de estos

La actividad o fragmento como controladorEn particular en Android, el controlador es implementado en la actividad o en el fragmento y se encarga de manejar todo lo que le ocurre a la aplicación.

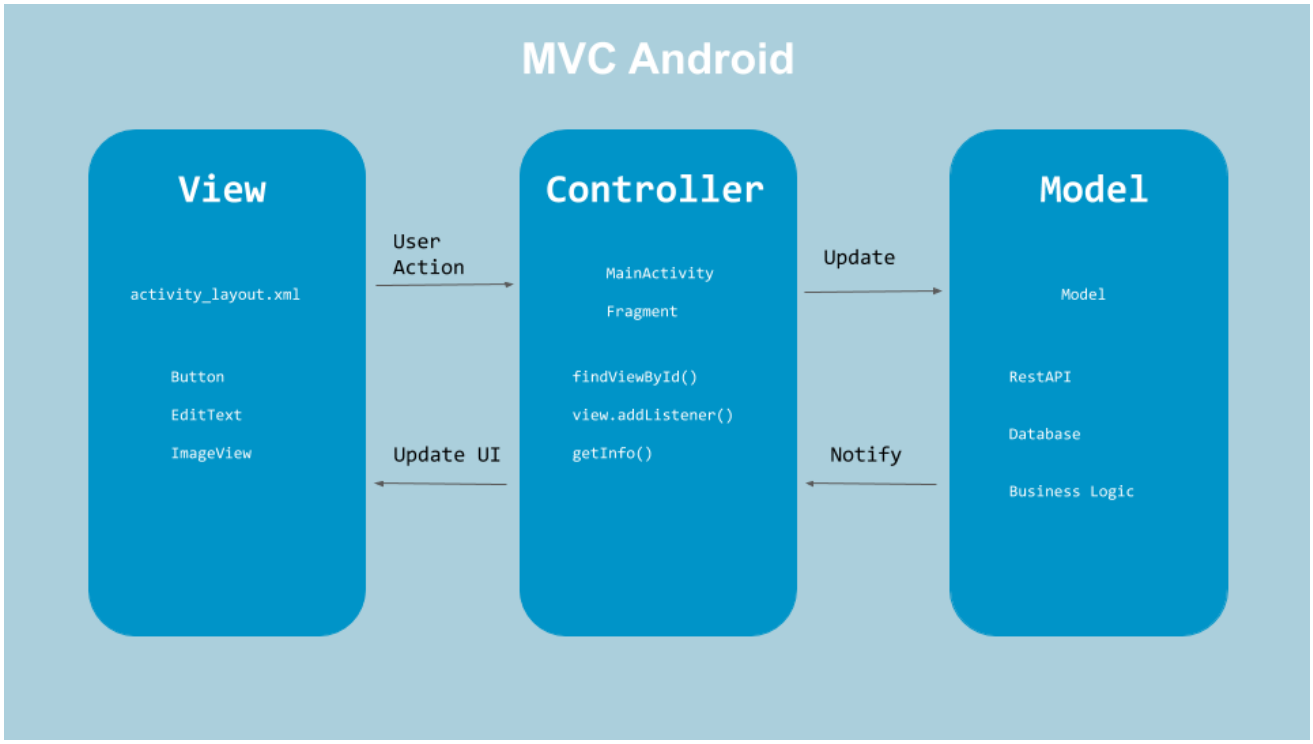


Imagen 11. Modelo - Vista - Controlador.

De la figura, no existen interacciones entre vista y modelo, todo pasa por el controlador:

1. Es el responsable de procesar los requerimientos (User Action)
2. Procesar los datos usando el modelo (Update)
3. El modelo entrega los resultados al controlador del procesamiento
4. Para luego entregar los resultados a la vista (Update UI)

Es posible que el controlador llame al modelo más de una vez para poder completar los requerimientos antes de entregar una respuesta a la vista.

Una actividad que implemente este patrón debe encargarse de instanciar el modelo e inicializar las vistas.

Para el juego TicTacToe la actividad principal se encarga de manejar los eventos onClick asociados a cada celda, definiendo en el layout la propiedad onClick.

```
<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="onCellClicked"
  app:srcCompat="@drawable/background_default_cell" />
```

En este caso, en el método onCellClicked de la MainActivity es responsable de:

- Interactuar con el modelo
- Actualizar la vista del tablero
- Verificar los resultados del juego (ganador o empate)
- Mostrar los resultados usando GameDialogFragment

```
public class MainActivity extends AppCompatActivity {

   private IBoard model;

   private boolean gameInProgress;

   ...

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);

       initGame();
   }
   private void initGame(){
       IPlayer player1 = new Player("Player1", IPlayer.TURN_PLAYER_1);
       IPlayer player2 = new Player("Player2", IPlayer.TURN_PLAYER_2);

       model = new Board(player1, player2);

       gameInProgress = true;
   }
```

```
    public void onCellClicked(View v) {
        Pair location =
getLocation(getResources().getResourceEntryName(v.getId()));

        if (this.model.nextMove(location.first, location.second)) {
            //Update view with the current player image
            ImageButton view = (ImageButton) v;

view.setImageResource(getPlayerResource(this.model.getCurrentPlayer().getPlayer
Turn()));

            if (this.model.hasWinner()) {
                showWinner(this.model.getWinner());
                gameInProgress = false;
                return;
            }

            if (this.model.isFull()) {
                showDraw();
                gameInProgress = false;
            }

            //Ready for the next move
            this.model.switchPlayer();

        } else {
            Timber.d("Invalid cellSelectedAt. Cell already used");
            // nothing to do here for now
            // maybe display a message
        }
    }
    ...
}
```

| **Pros** | **Fácil de implementar: No tiene dependencias ni capas extras, por lo que es fácil y rápido de implementar.** |
|---|---|
| Contra | El controlador está muy atado a la API de Android y no es posible hacer pruebas unitarias al controlador mismo. |
| Modularidad y Flexibilidad | El controlador está acoplado a las vistas, por lo que cualquier cambio en las vistas repercute en cambios en el controlador. |
| Mantenimiento | Con el tiempo y a medida que se incluyen más vistas a la interfaz, tanto el controlador como la vista crecen y se vuelve difícil de mantener. |

# Modelo - Vista - Presentador (MVP)

## Competencias

- Enlazar las capas de vista y modelo usando un presentador
- Realizar testing a un presentador

## Introducción

Los problemas del patrón MVC pueden ser mejorados desacoplando el controlador de la vista y creando el presentador, completamente separado de la vista, con la que se comunica utilizando callbacks.

MVP tomó fuerza hace unos años y fue adoptado por parte de la comunidad como respuesta principalmente a las dificultades para hacer testing de MVC. Al conocer y entender MVP, se tiene una herramienta muy útil para trabajar con aplicaciones heredadas o que por su problemática simple sean desarrolladas con este patrón.

## Presentador

El presentador se encarga de darle a la vista la información requerida y de recibir las peticiones desde la vista y utilizando el modelo, actualizar la vista con la nueva información.

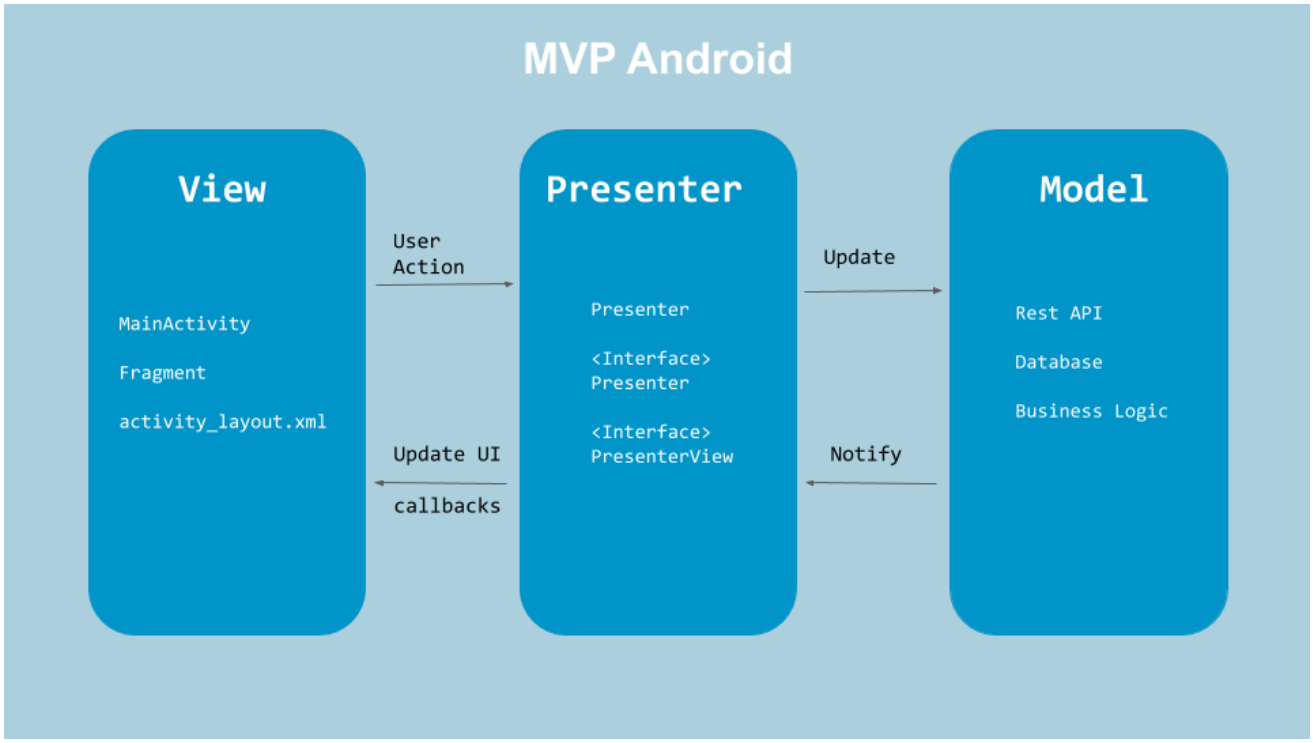


Imagen 12. Modelo Vista presentador.

1. La vista le entrega al presentador las acciones realizadas por el usuario, por ejemplo, le indica el evento onClick de un botón (User Action)
2. El presentador decide las acciones a realizar para ese evento y llama al modelo las veces que sea necesario para completar la respuesta (Update)
3. El modelo procesa el requerimiento y le entrega los resultados al presentador (Notify)
4. El presentador toma los resultados y se los entrega a la vista utilizando los *callbacks* definidos por la interfaz de comunicación Presentador-Vista (Update UI)

El presentador se comunica con la vista usando una interfaz asignada al presentador, de esta forma un Activity, Fragment o cualquier otra vista puede implementar la interfaz y presentar la información a su modo

# Presentador para TicTacToe

El presentador en el juego TicTacToe es el mediador entre la vista y el modelo. Entonces, el presentador está compuesto por:

- Una vista (MainActivity) que implementa la interfaz ITicTacToeView
- El modelo usando la interfaz IBoard

Estas relaciones se pueden ver en el siguiente diagrama de clases

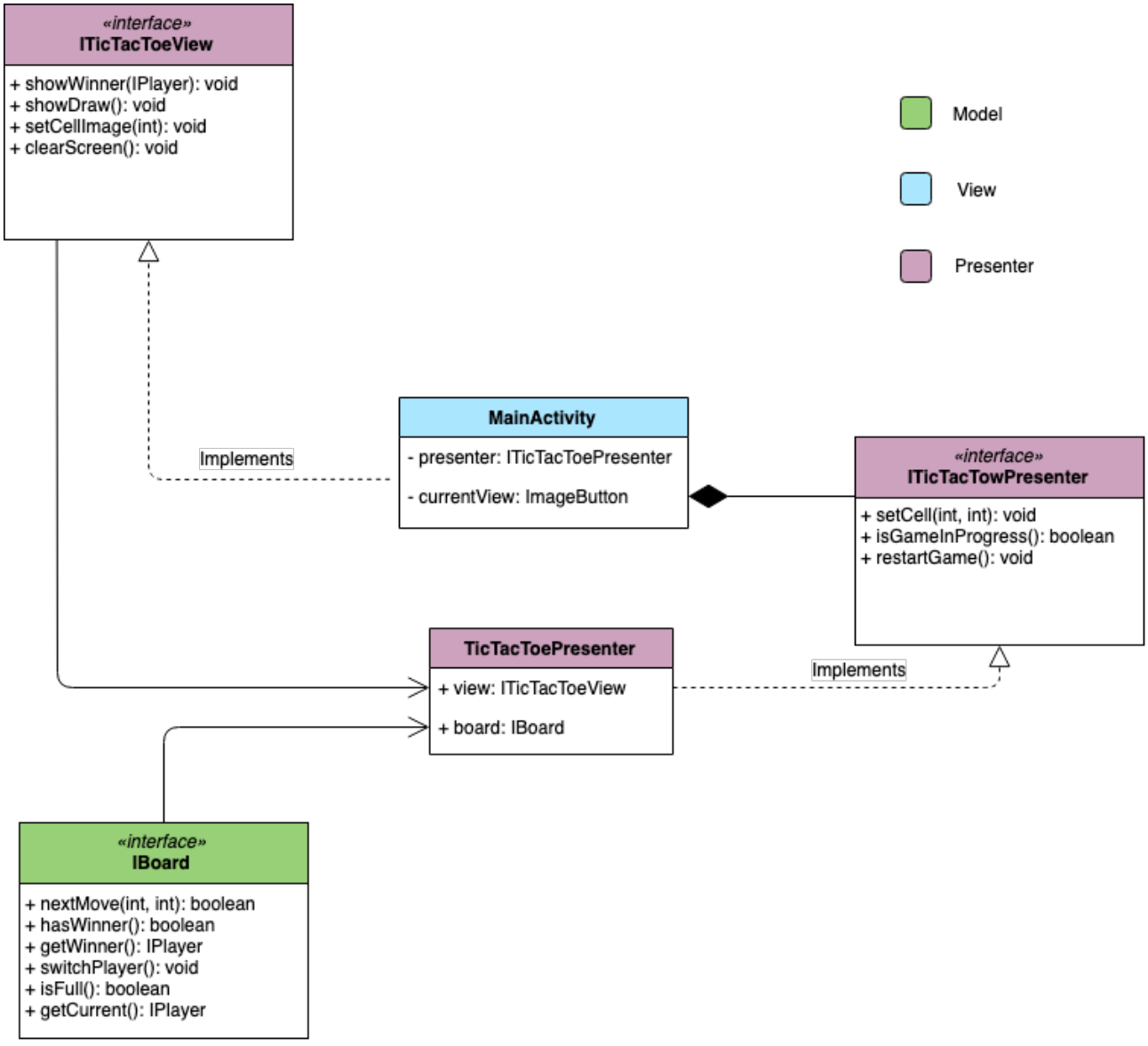


Imagen 13. Diagrama de clases

## Interface ITicTacToeView

La interfaz ITicTacToeView es el enlace entre el presentador y la actividad. Permite al presentador llamar a los callbacks definidos en ella

```
/**
* This interface must be implemented by any view (Activity or Fragment)
* who wants to be notified for changes in its screen for the presenter
*/
public interface ITicTacToeView {

   /**
    * Set the current view with an image resource
    *
    * @param imageResource to shown
    */
   void setCellImage(final int imageResource);

   /**
    * Show the player as winner
    *
    * @param winner
    */
   void showWinner(final IPlayer winner);

   /**
    * Same of showWinner but when it's a draw
    * There is no winner this time
    */
   void showDraw();

   /**
    * Clear all elements in the screen
    */
   void clearScreen();
}
```

La implementación de la interfaz permite al presentador avisar a la vista de ciertos eventos como resultado del procesamiento del modelo. Por ejemplo, al encontrarse un ganador, es el presentador el que llama al callback showWinner(...) de MainActivity para que lo muestre

```
public class MainActivity extends AppCompatActivity implements ITicTacToeView {
    ...

    /*
     * ==========================
     *   ITicTacToeView interface
     * ==========================
     */
    @Override
    public void showWinner(IPlayer winner) {
        GameDialogFragment gameDialogFragment = new
GameDialogFragment(GameDialogFragment.WINNER_DIALOG);
        gameDialogFragment.setWinner(winner);
        showDialog(gameDialogFragment);
    }

    @Override
    public void showDraw() {
        GameDialogFragment gameDialogFragment = new
GameDialogFragment(GameDialogFragment.DRAW_DIALOG);
        this.showDialog(gameDialogFragment);
    }

    @Override
    public void clearScreen() {
        this.currentView = null;

        for(int i=0; i<gridLayout.getChildCount(); i++) {
            ((ImageButton)
gridLayout.getChildAt(i)).setImageResource(R.drawable.background_default_cell);
        }
    }

    @Override
    public void setCellImage(int res) {
        currentView.setImageResource(res);
    }
}
```

## MainActivity

En este modelo, MainActivity es responsable de:

- Implementar la interfaz ITicTacToeView
- Tener una instancia del presentador que se crea junto con el Activity en onCreate()
- Tener las referencias a la vista del tablero. Usando ButterKnife se hace referencia al GridLayout que contiene el tablero en la vista con la anotación @BindView para limpiar los elementos al empezar un nuevo juegoInflar el layout en onCreate() usando *setContentView(R.layout.activity_main)*

```
package cl.desafiolatam.tictactoe.view;

public class MainActivity extends AppCompatActivity implements ITicTacToeView {

   private ITicTacToePresenter presenter;

   @BindView(R.id.main_grid_layout)
   GridLayout gridLayout;

   private ImageButton currentView;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
       this.presenter = new TicTacToePresenter(this);
       ...
   }

   ...
```

## ITicTacToePresenter

Es el contrato con el que se expone el presentador concreto y describe las llamadas que puede hacer la vista al presentador

```
public interface ITicTacToePresenter {

    /**
     * Set a move for a cell (x, y)
     * View does not care the current player
     * @param row x
     * @param col y
     */
    void setCell(int row, int col);

    /**
     *
     * @return {@code true} if there is a game in progress
     * {@code false} otherwise
     */
    boolean isGameInProgress();

    /**
     * Restart the board for a new game
     */
    void restartGame();
}
```

Con estas 3 acciones, la vista puede comunicarse con el presentador. Es luego que TicTacToePresenter implementa la interfaz para ser utilizado por la capa de presentación

## TicTacTowPresenter

Es el presentador concreto que se encarga de implementar las funcionalidades declaradas por la interfaz ITicTacTowPresenter

En su constructor recibe la vista para poder actualizarla usando el contrato dado por la interfaz ITicTacToeView

```
public class TicTacToePresenter implements ITicTacToePresenter {
   public static final int RES_PLAYER_1 = R.drawable.ic_player_1;
   public static final int RES_PLAYER_2 = R.drawable.ic_player_2;

   /**
    * Callbacks for update UI
    */
   private ITicTacToeView view;

   /**
    * Model
    */
   private Board board;

   /**
    * After winner or draw the game is stopped
    */
   private boolean gameInProgress;

   public TicTacToePresenter(@NonNull ITicTacToeView view) {
       this.view = view;
       newGame();
   }

   /************************************************
    ** ITicTacToePresenter interface implementation **
    ************************************************/
   public void setCell(final int row, final int col) {
       if (this.board.nextMove(row, col)) {
           //Update view with the current player image

view.setCellImage(getPlayerResource(this.board.getWinner().getPlayerTurn()));
```

```
            if (this.board.hasWinner()) {
                view.showWinner(board.getWinner());
                gameInProgress = false;
                return;
            }

            if (this.board.isFull()) {
                view.showDraw();
                gameInProgress = false;
            }

            //Ready for the next move
            this.board.switchPlayer();
        } else {
            Timber.d("Invalid cellSelectedAt. Cell already used");
        }
    }

    public boolean isGameInProgress() {
        return gameInProgress;
    }

    public void restartGame() {
        this.newGame();
    }
    public static int getPlayerResource(final int player) {
       return player == IPlayer.TURN_PLAYER_1 ? RES_PLAYER_1 : RES_PLAYER_2;
    }

    /************************************************
     ******* TicTacToePresenter private methods *******
     ************************************************/
    private void newGame() {
        IPlayer player1 = new Player("Player1", IPlayer.TURN_PLAYER_1);
        IPlayer player2 = new Player("Player2", IPlayer.TURN_PLAYER_2);

        this.board = new Board(player1, player2);

        this.gameInProgress = true;
    }
}
```

## Probando el presentador

Uno de los beneficios de usar MVP es que las funcionalidades del presentador pueden ser probadas usando test unitarios

A este nivel no se tienen dependencias al SDK de Android, y solo se quiere probar código propio, así que para la inyección de dependencias mock, Mockito es una buena opción.

La clase TicTacToePresenterTest tiene una interfaz ITicTacToeView con la anotación @Mock que será la responsable de los eventos hacia la vista

```
package cl.desafiolatam.tictactoe.presenter;

import ...

public class TicTacToePresenterTest {

   /**
    * Our presenter ready to test
    */
   private TicTacToePresenter presenter;

   /**
    * A winner player for showWinner() method
    */
   private IPlayer winner;

   @Mock
   private ITicTacToeView view;

   @Before
   public void setUp() {
       MockitoAnnotations.initMocks(this);

       this.presenter = new TicTacToePresenter(view);
       this.winner = new Player("Player1", IPlayer.TURN_PLAYER_1);
   }

   @Test
   public void isGameInProgress_ok() {
       assertTrue(this.presenter.isGameInProgress());
   }
```

```
    @Test
    public void setCell_validMove(){
        this.presenter.setCell(0, 0);
        Mockito.verify(view,
Mockito.only()).setCellImage(TicTacToePresenter.RES_PLAYER_1);
    }

    @Test
    public void setCell_cellAlreadyInUse(){
      this.presenter.setCell(0, 0);
      Mockito.verify(view,
Mockito.only()).setCellImage(TicTacToePresenter.RES_PLAYER_1);

      this.presenter.setCell(0, 0);
      // the view was not updated after setCell
      Mockito.verify(view,
Mockito.never()).setCellImage(TicTacToePresenter.RES_PLAYER_2);

      // there is no winner yet
      Mockito.verify(view, Mockito.never()).showWinner(winner);

      // neither draw
      Mockito.verify(view, Mockito.never()).showDraw();
    }


    @Test
    public void setCell_showWinner(){
        // do moves for player 1 becomes winner
        doMoves(MOVES_P1_WINS);

        Mockito.verify(view, Mockito.atLeastOnce()).showWinner(winner);
    }
    @Test
    public void setCell_showDraw(){
        doMoves(MOVES_TO_DRAW);

        Mockito.verify(view, Mockito.atLeastOnce()).showDraw();
    }
    @Test
    public void restartGame_clearScreen() {
        this.presenter.restartGame();

        Mockito.verify(view, Mockito.only()).clearScreen();
    }
```

Los test verifican que luego de ejecutarse una acción en el presentador se llame a los callbacks necesarios de la vista. Por ejemplo, para *restartGame_clearScreen()*, luego de indicar al presentador que reinicie el juego, la vista tiene que haber sido actualizada usando el callback *clearScreen()*. De no ser así, el presentador está teniendo un comportamiento distinto al esperado

Nota:

doMoves() es un método auxiliar que realiza una lista de movimientos que se le pasa por parámetro. MOVES_TO_DRAW y MOVES_P1_WIN son estas listas con los movimientos necesarios para que ocurra un empate o para que gane el jugador 1.Están disponibles en la branch MVP del proyecto TicTacToe.

| **Pros** | **Pruebas: Con la separación que tiene la vista y el presentador, este puede ser probado usando test unitarios** |
|---|---|
| Contra | Mantenimiento: Al igual que el controlador, el presentador tiende a crecer con el tiempo. Una solución para esto es dividir en varios presentadores acotados por funcionalidad |
| Modularidad y Flexibilidad | El presentador no tiene responsabilidades sobre cómo desplegar la vista. Como lo desconoce, la vista puede intercambiarse sin afectar al presentador |

## Secuencia para el empate en el juego

¿Cuál es el flujo interno de la app cuando hay un empate?

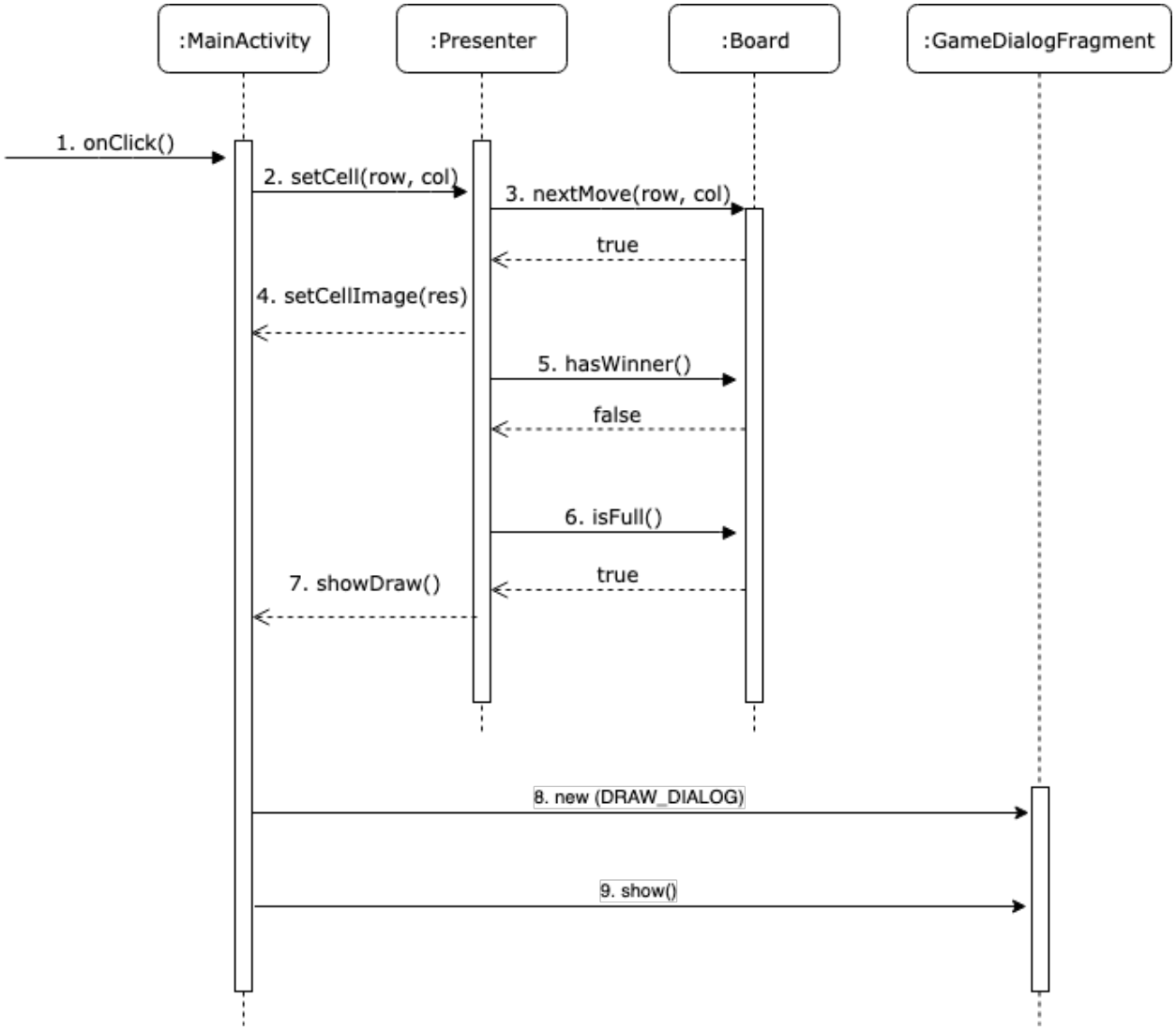


Imagen 14. Diagrama de Flujo para empate.

1. El usuario hace click en una celda del tablero
2. MainActivity le dice al presentador que ocurrió el evento y dónde usando row y col
3. El presentador le indica al modelo que ocupe la celda en las coordenadas indicadas. Luego retorna al presentador indicando si pudo o no hacer el movimiento
4. El presentador actúa en función de la respuesta del modelo. Si se pudo realizar el movimiento, hace el llamado a actualizar la pantalla usando el callback
5. Además de actualizar UI, el presentador revisa si hay un ganador usando el modelo
6. Luego revisa si el tablero está lleno (empate).
7. El presentador utiliza el callback showDraw() para que la vista muestre el resultado
8. Es la actividad la que crea el nuevo diálogo
9. Y se encarga de mostrarlo