Arquitectura de Software y Testing - Parte I

Arquitectura de Software

Competencias:

- Aprender qué es la arquitectura de software
- Beneficios de aplicar arquitectura de software
- Aprender principios de arquitectura en android y las más aplicadas

Introducción

A continuación aprenderemos los conceptos principales de desarrollo de arquitectura de software, porque existen y cómo se aplican en los proyectos de aplicaciones android.

Es importante conocer estos conceptos y fundamentos de arquitectura de software en nuestra vida laboral, dado que nos permite realizar un mejor trabajo, potenciar nuestra carrera, brindándonos la oportunidad de no solo hacer mejor el trabajo sino de optar por mejores cargos profesionales.

Qué es la arquitectura de software

La arquitectura de software es el diseño de más alto nivel de la estructura de un sistema.

En los inicios de la informática, la programación se consideraba un arte y se desarrollaba como tal, debido a la dificultad que representaba para la mayoría de las personas. Con el tiempo, se han ido descubriendo y desarrollando formas y guías generales, como bases para la resolución de los problemas. A estas, se les ha denominado **arquitectura de software**, porque, a semejanza de los planos de construcción, estas indican la estructura, funcionamiento e interacción entre las partes del software.

Generalmente, no es necesario inventar una nueva arquitectura de software para cada sistema de información. Lo habitual es adoptar una arquitectura conocida en función de sus ventajas e inconvenientes para cada caso en concreto. Así, las arquitecturas más universales son:

- **Descomposición Modular**: donde el software se estructura en grupos funcionales muy acoplados.
- **Cliente-servidor**: Donde el software reparte su carga de cómputo en dos partes independientes pero sin reparto claro de funciones.
- **Arquitectura de tres niveles**: Especialización de la arquitectura cliente-servidor donde la carga se divide en tres partes (o capas) con un reparto claro de funciones: una capa para la presentación (interfaz de usuario), otra para el cálculo (donde se encuentra modelado el negocio) y otra para el almacenamiento (persistencia). Una capa solamente tiene relación con la siguiente.

Otras arquitecturas:

- Modelo vista Controlador.
- En pipeline.
- Entre pares.
- En pizarra.
- Orientada a servicios (SOA del inglés Service-Oriente-Architecture).
- Arquitectura de microservicios (MSA del inglés MicroServices Architecture).Algunos consideran que es una especialización de una forma de implementar SOA.
- Dirigida por eventos.
- Máquinas virtuales.

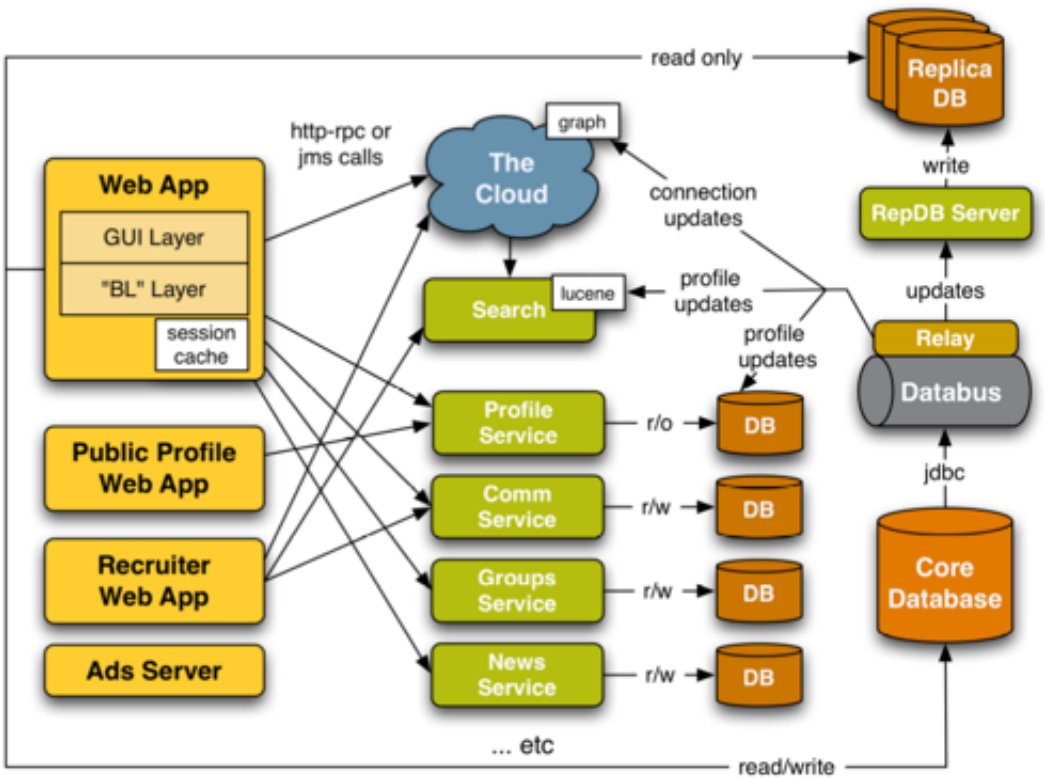

Imagen 1. Diagrama ejemplo de arquitectura de software.

El desarrollo de una arquitectura de software en esencia levanta los requisitos de un cliente (quién necesita el sistema) y garantiza un diseño que cumpla con lo requerido por el negocio solicitado, esto por supuesto también contempla un diseño de fácil adaptación a los cambios que se pueden producir por nuevas reglas de venta u operacionales.

Es muy oportuno comparar la construcción de un sistema con la construcción de una casa, es decir, ambos escenarios contemplan levantamiento de requerimientos, pensar en una arquitectura que garantice lo construido en el tiempo, calcular costos, proyectar días de trabajo necesarios para completar la obra. Como sugerencia, es fundamental comparar con ejemplos prácticos de la vida real lo que conlleva desarrollar un sistema para hacer entender el esfuerzo y dedicación que se necesita para su concreción.

Importancia de la arquitectura de software

Al tomar los requerimientos de una idea de sistema, el siguiente paso es analizar el cómo se debería construir este sistema para cumplir con su finalidad, lo cual nos lleva a un análisis de arquitectura, el cual contempla las siguientes situaciones:

- Definir acciones en base a los límites en el presupuesto y los tiempos de entrega pactados.
- Determinar cómo sería la implementación del sistema en distintos escenarios y sistemas operativos.
- Facilitar el entendimiento del código escrito para los cambios futuros que posiblemente realizarán otras personas.
- Asegurar el corazón del negocio dentro del sistema, para que no sea afectado de manera general por actualizaciones mal pensadas.
- Diseñar una estructura y ordenamiento de todas las clases y objetos del sistema.
- Establecer un standard de construcción basado en patrones de diseño y buenas prácticas.
- Seleccionar y asociar librerías y software de terceros en caso de ser requeridos.

Un buen diseño de arquitectura de software promueve la mantención, soporte y administración adecuada de los sistemas, garantizando su duración en el tiempo, sin importar los cambios que deban ocurrir en su lógica. Con una arquitectura bien diseñada, los costos operativos y financieros de los cambios que han de suceder en el sistema son menores que en aquellos con mal diseño de arquitectura.

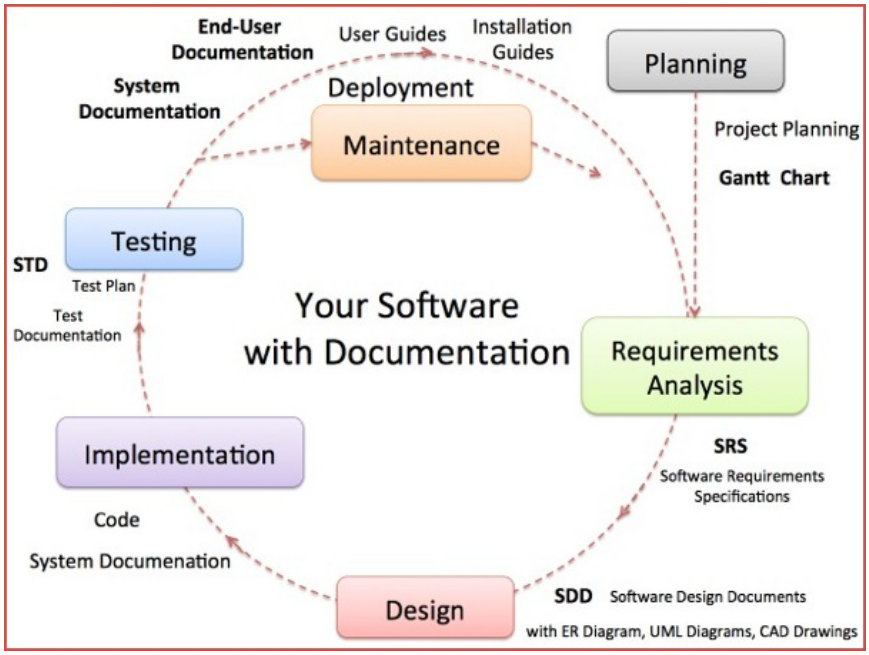

Imagen 2. Diagrama representativo de las etapas de levantamiento y documentación de un sistema en sus ciclos de maduración como parte de un todo en la arquitectura

Los requerimientos del software moderno son cada vez más complejos puesto que los usuarios esperan más de sus aplicaciones. Hoy en día, las aplicaciones deben interactuar con otras aplicaciones y servicios y ejecutarse en una serie de entornos, como la nube, y en dispositivos móviles. Los diseños comunes en el pasado se han reemplazado por software basado en componentes orientados a los servicios.

Objetivos de la arquitectura de software

El objetivo principal de la arquitectura es identificar los requisitos que afectan la estructura de la aplicación. Una arquitectura bien diseñada reduce los riesgos comerciales asociados con la creación de una solución técnica y crea una conexión entre los requisitos comerciales y técnicos.

Objetivos secundarios:

- Exponer la estructura del sistema, ocultando los detalles de su implementación.
- Desarrollar todos los casos de uso y escenarios requeridos.
- Implementar los requisitos funcionales asegurando la calidad del sistema.
- Mejorar la confianza externa en la organización o el sistema.

El papel de un arquitecto de software

Un arquitecto de software facilita una solución que puede contemplar documentos, maquetas, ejemplos, reglas y herramientas a utilizar, todo esto, para que un equipo técnico desarrollador pueda crear el sistema. Un arquitecto de software debe tener entre otras, las siguientes aptitudes:

- Capacidad de negociación sobre los requerimientos del sistema.
- Definir expectativas en cuanto al tiempo que conlleva la construcción del sistema.
- Determinar el número de desarrolladores necesarios para cumplir los tiempos y realizar el trabajo.
- Participar los costos financieros que pueden tener el proyecto para su materialización.
- Guiar al equipo de desarrollo en la construcción del sistema sin mayor protagonismo.
- Ser especialista en las tecnologías a utilizar para el desarrollo del sistema.

Principios comunes de arquitectura en android

Separation of Concerns

En el desarrollo android, el principio más importante según google que debes seguir, es el de la separación de problemas. Debemos evitar escribir todo el código en una sola actividad o fragmento. Las clases basadas en la interfaz de usuario solo deberían contener la lógica que se ocupa de interacciones del sistema operativo y del hilo principal de interfaz de usuario. Si mantienes estas clases tan limpias como sea posible, puedes evitar muchos problemas relacionados con el ciclo de vida.

El sistema operativo android puede finalizar una mala implementación en cualquier momento si se presenta por ejemplo un error como memoria insuficiente.

Controlar la IU a partir de un modelo

Otro principio importante es que debes controlar la IU (interfaz de usuario) a partir de un modelo, preferentemente uno de persistencia. Los modelos son los componentes responsables de manejar los datos de una aplicación. Son independientes de los componentes de la aplicación y los objetos tipo vista, de modo que no se ven afectados por el ciclo de vida de la app y los problemas asociados.

Desarrollar un modelo de persistencia

El modelo de persistencia es ideal debido a los siguientes motivos:

- Tus usuarios no perderán datos si el sistema operativo android cierra tu app para liberar recursos.
- Tu aplicación continuará funcionando cuando la conexión de red sea inestable o no esté disponible.
- Desarrollando clases de modelos con una responsabilidad bien definida para la administración de datos, tu aplicación será más consistente y será más fácil realizar pruebas en ella.

MVx - Patrones de arquitectura de software en android

Todos los patrones Mvx son implementaciones de diseño que comparten un objetivo principal, el cual es desacoplar, separar la vista (UI) de los objetos de negocio , lógicas, apis y bases de datos. Listemos razones para cumplir con la separación de la interfaz de usuario:

- La lógica de IU generalmente tiene los requisitos más detallados y precisos.
- La tasa de cambio de la interfaz de usuario de la aplicación suele ser mucho mayor que la de otras partes de la aplicación.
- La lógica de la interfaz se tiene que mantener lo más simple y básica posible a nivel de lógica.
- Buena parte de las malas prácticas en el código de las aplicaciones de android, están relacionados con el código en la interfaz de usuario.
- Realizando la separación se puede probar fácilmente la lógica de la interfaz de usuario con casos unitarios.
- Es extremadamente difícil probar la lógica de UI con pruebas automatizadas sin dicha separación.

Es por esto que son MVx, Model, View y la obligatoria separación o desacoplamiento de la interfaz de usuario expresada en VM (View Model), P(Presenter), C(Controller).

Es importante destacar que estos patrones de diseño pueden seguir los principios SOLID y Clean Architecture.

A continuación detallaremos dos de los patrones más comunes en el diseño de arquitectura de software android.

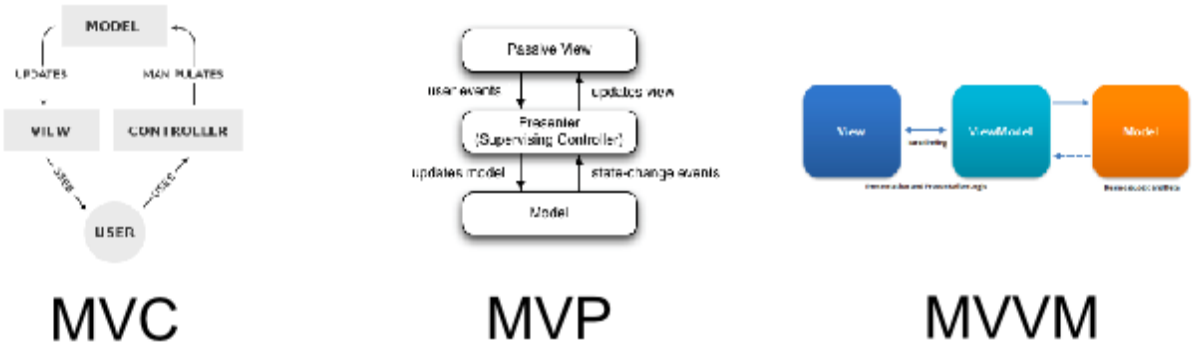

Imagen 3. Diagrama representativo MVC, MVP y MVVM

Model View ViewModel (MVVM)

El patrón de arquitectura de software recomendado por google android hoy por hoy es la arquitectura basada en el patrón de diseño ViewModel, específicamente MVVM (Model-View-ViewModel). Es importante admitir que es relativamente nuevo el patrón y es recomendable implementarlo con kotlin, y este es recomendado por android dado que se adapta perfectamente a la tendencia reactive de la programación moderna, ajustándose a los componentes de arquitectura que han tenido un gran avance durante los últimos tiempos.

Imagen 4. Model View ViewModel

MVVM consta de tres capas: Model, View y ViewModel. Cada una de estas capas o componentes tiene sus propias responsabilidades para hacer su tarea.

Modelo: en la capa de modelo, escribimos nuestra lógica de negocios.

ViewModel: esta capa interactúa con la vista y el modelo, proporciona datos observables (Live Data por ejemplo) monitoreadas por la vista que puede ser un fragmento de actividad o una actividad.

Vista: la vista es un fragmento de actividad o una actividad. La vista observa el el ViewModel para actualizar sus objetos.

La página oficial de android developers tiene un caso práctico explicando este particular que vale la pena seguir a continuación:

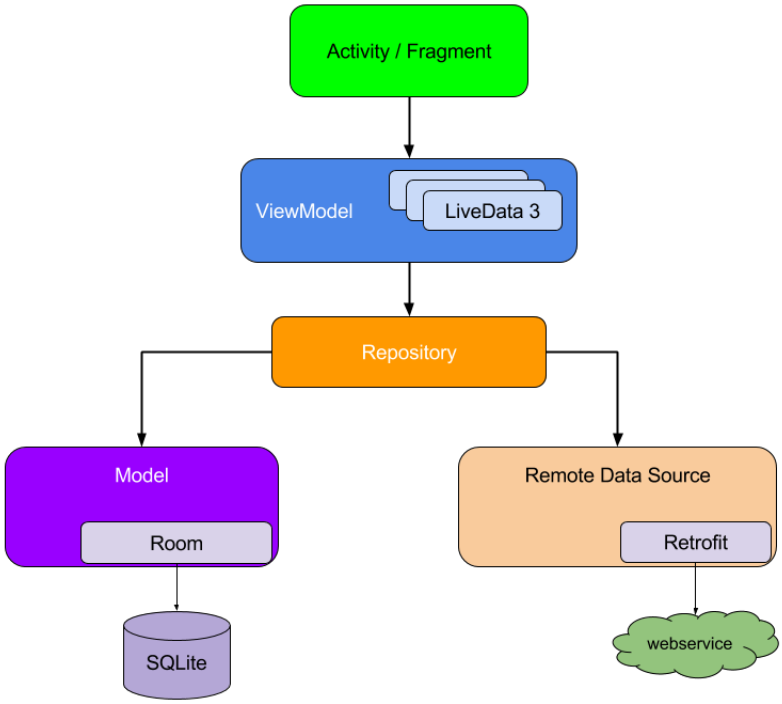

Imagen 5. Representación gráfica de implementación MVVM con repositorio

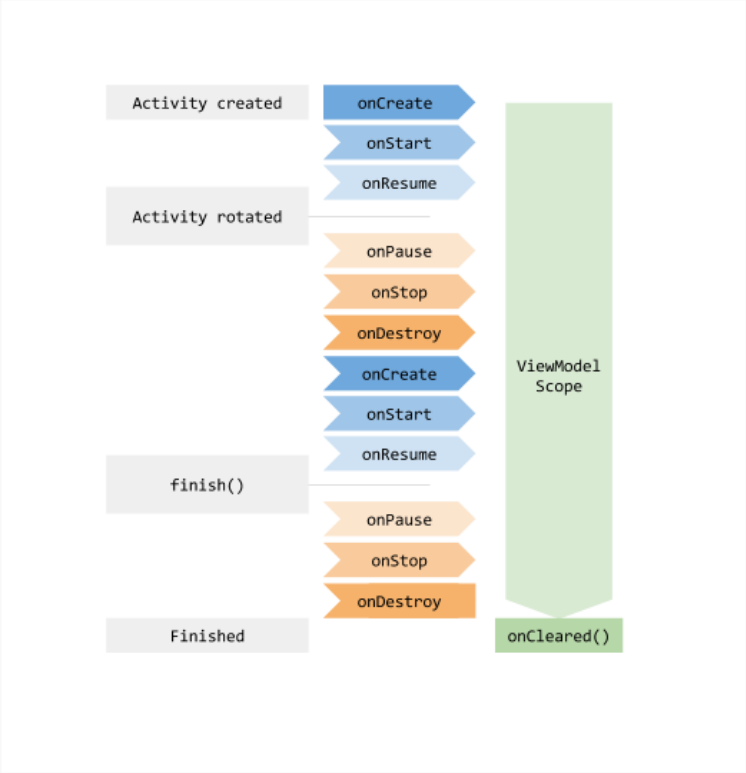

Imagen 6. Adaptación del ViewModel al ciclo de vida de una actividad

Model View Presenter (MVP)

Es una derivación del patrón arquitectónico modelo–vista–controlador (MVC), y es utilizado mayoritariamente para construir interfaces de usuario.

En MVP el presentador asume la funcionalidad del "intermediario". En MVP, toda lógica de presentación es colocada al presentador.

- **Model**: la capa de datos, responsable de manejar la lógica de negocios y la comunicación con las capas de la red y la base de datos.
- **View**: la capa de interfaz de usuario: muestra los datos y notifica al presentador sobre las acciones del usuario.
- **Presenter**: recupera los datos del modelo, aplica la lógica de la IU (interfaz de usuario/View) y administra el estado de la vista, decide qué mostrar y reacciona ante las notificaciones de entrada del usuario desde la vista.

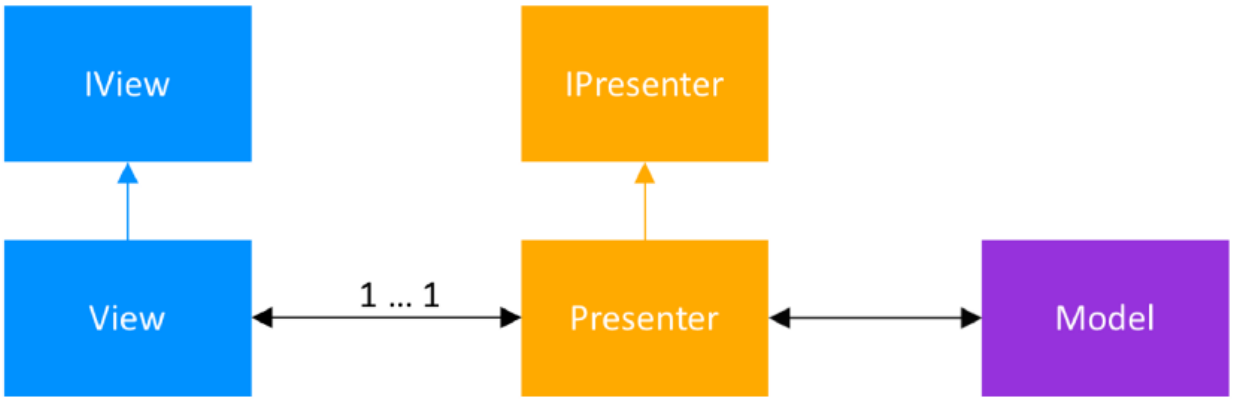

Imagen 7. Diagrama de patrón de diseño MVP - Capas e interfaces

Model View Controller (MVC)

Es de los patrones de diseño más maduros y comprobados de la arquitectura de software, lo podemos describir en este caso para android como un MVP donde el presentador pasa a ser un controlador que no utiliza las operaciones de las vistas para acoplar, sino más bien se basa en listeners.

- **Model**: la capa de datos, responsable de manejar la lógica de negocios y la comunicación con las capas de la red y la base de datos.
- **View**: la capa de interfaz de usuario: muestra los datos y notifica al presentador sobre las acciones del usuario.
- **Controller**: recupera los datos del modelo, administra los eventos por medio de composiciones listeners, decide qué mostrar y reacciona ante las notificaciones de entrada del usuario desde la vista.

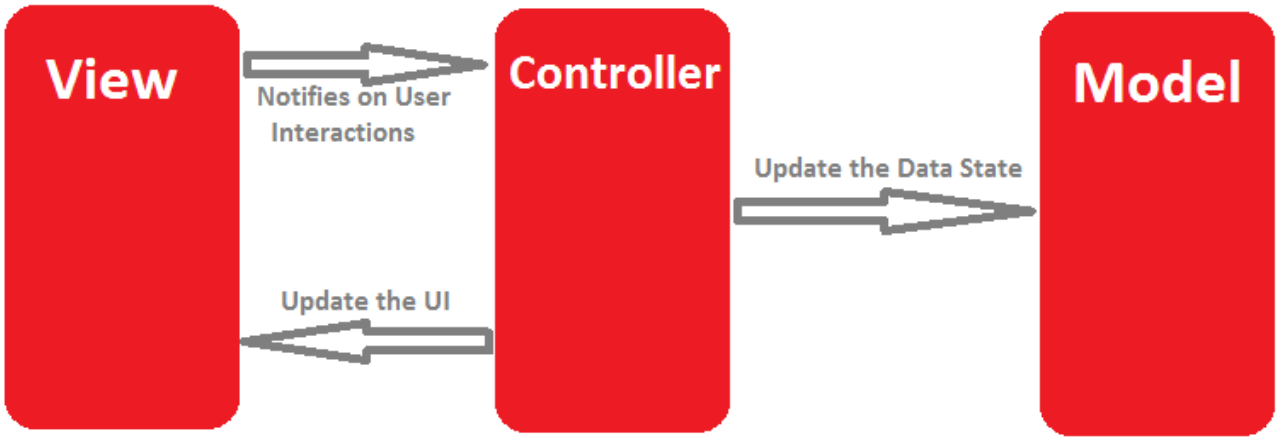

Imagen 8. Representación de patrón de diseño MVC

Principios SOLID

Competencias:

- Aprender qué son los principios SOLID
- Aplicar principios SOLID en un proyecto

Introducción

El objetivo de cada desarrollador es escribir un código que consiga cumplir en su totalidad los requisitos del negocio y que pueda satisfacer fácilmente los requisitos futuros. Evolucionar con el tiempo es el único factor que puede ayudarlo a continuar. Los principios SOLID te ayudan a escribir dicho código.

La calidad del código a construir en nuestras aplicaciones depende en gran medida de la capacidad que tengamos de implementar estos principios SOLID, junto con las buenas prácticas y recomendaciones asociadas que a lo largo de nuestra carrera con la experiencia adaptamos.

Conceptos y Aplicación de principios SOLID

SOLID son siglas que corresponden a cinco principios que explicaremos a continuación.

Single Responsibility Principle (S)

Este principio sugiere que una clase, método, función debería tener una sola responsabilidad, es decir, si la clase conecta realiza una operación de conexión a base de datos, la clase sólo debería dedicarse a ello, dejando en otra clase la declaración de alguna query.

Open/Closed Principle (O)

Sugiere que las clases deberían estar abiertas para ser extendidas pero estar cerradas para ser modificadas, es decir, una clase A desarrollada por un programador, para hacer alguna modificación por un segundo programador, esta debe realizarse en una clase B que extienda de la clase A sin llegar a modificar directamente la clase A.

Liskov´s Substitution Principle (L):

establece que las clases deberían de ser sustituidas fácilmente por sus clases hijas sin afectar el sistema. Consideremos el siguiente ejemplo:

- Una clase padre llamada Animal:

```
public class Animal {
    public void makeNoise() {
        System.out.println("I am making noise");
    }
}
```

- Dos clases hijas de "Animal" llamadas "Perro y Gato":

```
public class Perro extends Animal {
  @Override
  public void makeNoise() {
  System.out.println("bow wow");
  }
}

public class Gato extends Animal {
  @Override
  public void makeNoise() {
  System.out.println("meow meow");
  }
}
```

- En cualquier parte de nuestro código deberíamos estar habilitados para reemplazar nuestra clase Animal por la de Perro o Gato sin que falle nuestro sistema, es decir, una clase hija que extiende de la clase Animal no debe implementar código que al ser reemplazado el método makeNoise() colapse el sistema por incongruencia. Por ejemplo:

```
class AlgunAnimal extends Animal {
    @Override
    public void makeNoise() {
        throw new RuntimeException("No puedo hacer ruido");
    }
}
```

- En este último caso nuestra aplicación fallará y se detendrá.

En android uno de los ejemplos más oportunos para implementar el principio de Liskov´s es una clase adapter que funcione con la clase RecyclerView, evitando escribir código que pudiera llevar a fallar al RecyclerView.

Interface Segregation Principle (I)

Este principio sugiere que es mejor tener varias interfaces específicas que una sola interfaz general. Revisemos el siguiente ejemplo:

- En android tenemos múltiples click listeners como el OnClickListener o el OnLongClickListener.

```
/**
* Interface definition for a callback to be invoked when a view is clicked.
*/
public interface OnClickListener {
    /**
     * Called when a view has been clicked.
     *
     * @param v The view that was clicked.
     */
    void onClick(View v);
}

/**
* Interface definition for a callback to be invoked when a view has been
clicked and held.
*/
public interface OnLongClickListener {
    /**
     * Called when a view has been clicked and held.
     *
     * @param v The view that was clicked and held.
     *
     * @return true if the callback consumed the long click, false otherwise.
     */
    boolean onLongClick(View v);
}
```

Nos podemos preguntar en este caso el porque necesitaríamos tener dos interfaces para la misma acción con pequeñas diferencias. ¿Por qué no una sola interfaz?

```
public interface MyOnClickListener {
  void onClick(View v);
  boolean onLongClick(View v);
}
```

Si tenemos una sola interfaz, nos obligaría a implementar métodos que posiblemente no usaremos, lo que haría nuestro código desordenado y menos óptimo.

Dependency Inversion Principle (D)

promueve que las clases dependen de la abstracción y no de lo concreto; esto lo que quiere decir es que deberíamos tener objetos de interfaz que nos ayuden a comunicarnos con las clases. El objetivo principal de este principio es ocultar la implementación de la clase A de la clase B, de tal manera que si alguna de las dos cambia, la otra clase no tiene porque enterarse de ello.

Un ejemplo en android sería la implementación del patrón de diseño MVP (Model View Presenter) donde necesitamos mantener la referencia del Presenter en nuestra View. Siguiendo el principio de inversión de dependencia, creamos una interfaz que oculte la implementación de nuestro Presenter y la clase de la View hace referencia a la interfaz del Presenter.

El objetivo de cada desarrollador es escribir un código que consiga cumplir en su totalidad los requisitos del negocio y que pueda satisfacer fácilmente los requisitos futuros. Evolucionar con el tiempo es el único factor que puede ayudarlo a continuar. Los principios SOLID te ayudan a escribir dicho código.

Ejercicio 1

Crear un proyecto aplicando los principios SOLID desde su raíz principio por principio, que se conecte a un api de consulta de indicadores económicos de Chile, permitiendo en nuestra actividad el ingreso de un string indicando el tipo de moneda y un string indicando la fecha en formato dd-mm-yyyy, y por supuesto mostrar el resultado de la consulta sobre los inputs anteriores. Esta aplicación debe consultar cada vez que se presione un botón "Consultar" que haga el llamado al api de indicadores con los parámetros tipoMoneda y fecha.

1. Creamos un nuevo proyecto llamado "PruebaDinamicaSolid" con api mínima 19, seleccionando la plantilla "Blank activity".
2. Usaremos una api libre y gratuita de indicadores económicos en chile.

   Página https://mindicador.cl/ utilizando el método GET que solicita tipo de indicador y fecha https://mindicador.cl/api/dolar/16-07-2019, como lo muestra la imagen 9:

Imagen 9. Api indicadores económicos.

Un ejemplo de respuesta invocando esta api sería el siguiente:

```
{
  version: "1.5.0",
  autor: "mindicador.cl",
  codigo: "dolar",
  nombre: "Dólar observado",
  unidad_medida: "Pesos",
  serie: [{
    fecha: "2019-07-18T04:00:00.000Z",
    valor: 681.69
  }]
}
```

3. Implementamos la librería retrofit para poder consumir esta api de indicadores en nuestro gradle app del proyecto:

```
implementation 'com.squareup.retrofit2:retrofit:2.6.0'
implementation 'com.squareup.retrofit2:converter-gson:2.6.0'
```

4. De acuerdo al principio de **Single Responsability**, crearemos clases con una única responsabilidad:

- **utils/RetrofitClient.java**

  Clase cuya única función será crear una instancia del cliente retrofit para ser usada desde otra clase.

```
public class RetrofitClient {

        private static Retrofit retrofit;
        private static final String BASE_URL = "https://mindicador.cl/api/";

        public static Retrofit getRetrofitInstance(){
            if (retrofit == null) {
                retrofit = new retrofit2.Retrofit.Builder()
                        .baseUrl(BASE_URL)
                        .addConverterFactory(GsonConverterFactory.create())
                        .build();
            }
            return retrofit;
        }
}
```

- **pojo/indicadoreconomico/indicadorEconomico**

  Clase que mantiene los datos del objeto de respuesta principal.

```
public class IndicadorEconomico {

    private String nombre;
    private String unidad_medida;
    private ArrayList<SerieIndicadorEconomico> serie;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getUnidad_medida() {
        return unidad_medida;
    }

    public void setUnidad_medida(String unidad_medida) {
        this.unidad_medida = unidad_medida;
    }

    public ArrayList<SerieIndicadorEconomico> getSerie() {
        return serie;
    }

    public void setSerie(ArrayList<SerieIndicadorEconomico> serie) {
        this.serie = serie;
    }
}
```

- **pojo/indicadoreconomico/SerieIndicadorEconomico**

  Clase que mantiene los datos del objeto de respuesta dentro del arreglo "serie" del objeto principal.

```
public class SerieIndicadorEconomico {

   private String fecha;
   private double valor;

   public String getFecha() {
       return fecha;
   }

   public void setFecha(String fecha) {
       this.fecha = fecha;
   }

   public double getValor() {
       return valor;
   }

   public void setValor(double valor) {
       this.valor = valor;
   }
}
```

5. De acuerdo con el principio de **Interface Segregation**, crearemos una interfaz de clase llamada IndicadorEconomicoApi que contenga métodos únicamente asociados a la indicadores económicos. Si existiese la necesidad de consultar otros métodos distintos a indicadores económicos debemos crear una nueva interfaz para tal fin y así sucesivamente.

```
public interface IndicadorEconomicoApi {

   @GET("{tipoIndicador}/{fechaIndicador}")
   Call<IndicadorEconomico> getIndicadorEconomico(@Path("tipoIndicador") String
tipoIndicador, @Path("fechaIndicador") String fechaIndicador);
}
```

6. Siguiendo el principio de **Interface Segregation** y el principio **Open/Closed** crearemos una nueva interfaz para controlar la respuesta del llamado en la carpeta `/api (cl.desafiolatam.pruebadinamicasolid.api)` de la siguiente manera:

```
public interface RequestInterfaceApi {
       void Response(Response response);
}
```

7. La interfaz anterior tiene un método Response() que debe instanciar una clase propia llamada "Response" para que tenga un trato propio la respuesta del api en nuestro sistema, para ello, crearemos una nueva clase "Response" siguiendo el principio **Single Responsability**:

```
public class Response {
  public ResponseState state;
  public boolean hasError;
  public RequestType requestType;
  public IndicadorEconomico result;
  public String errorMessage;

  public enum ResponseState {
       SUCCESS,
       FAILURE,
       NO_CONNECTION
   }
   public enum RequestType {
       SEARCH_FOR_INDICADOR
   }
}
```

8. Requerimos crear una clase `ClassHandle` en la carpeta api, para el manejo del callback de la llamada de retrofit y asociar los objetos resultantes a nuestra interfaz RequestInterfaceApi que a su vez instancia nuestra clase Response. **Single Responsability, Loskov's Substitution**.

```
public class CallHandle {
   private static String TAG = "CallHandle";
   public void handleCallBack(final RequestInterfaceApi
objRestRequestInterface, Call objCall, final Response objResponse) {
       objCall.enqueue(new Callback() {
           @Override
           public void onResponse(Call call, retrofit2.Response response) {
               try {
                   Log.d(TAG, response.message());

                   if (response.isSuccessful()) {
                       objResponse.state = Response.ResponseState.SUCCESS;
                       objResponse.hasError = false;
                       objResponse.result = response.body();
                       assert objResponse.result != null;
                   } else {
                       Log.d(TAG, "Error: "+response.message());
                       assert response.errorBody() != null;
                       objResponse.errorMessage = response.message();
                   }
                   objRestRequestInterface.Response(objResponse);

               } catch (Exception objException) {
                   objResponse.errorMessage = "Error en la respuesta";
                   objRestRequestInterface.Response(objResponse);
               }
           }
           @Override
           public void onFailure(Call call, Throwable objThrowable) {
               String errorMessage = "";
               if (objThrowable instanceof IOException) {
                   errorMessage = "No connection error";
               } else {
                   errorMessage = "Server error";
               }
               objResponse.errorMessage = errorMessage;
               objRestRequestInterface.Response(objResponse);
           }
       });
   }
}
```

9. A continuación crearemos un handler en la carpeta utils/handlers llamado IndicadorEconomicoHandler extendiendo de CallHandler, que contendrá todos los métodos asociados a indicadores económicos, indicando parámetros únicos, la instancia al cliente retrofit con la interfaz necesaria, en este caso la IndicadorEconomicoApi, para finalmente usar el método handleCallBack() de la clase extendida. **Liskov's Substitution**.

```
public class IndicadorEconomicoHandler extends CallHandle {

   private String tipoIndicador, fechaIndicador;

   public IndicadorEconomicoHandler(String tipoIndicador, String
fechaIndicador){
       this.tipoIndicador = tipoIndicador;
       this.fechaIndicador = fechaIndicador;
   }

       public void getIndicadorEconomico(RequestInterfaceApi
objRestRequestInterface) {
           Call<IndicadorEconomico> objCall;
           Response objResponse = new Response();
           IndicadorEconomicoApi indicadorEconomicoApiEndPoint;

           objResponse.state = Response.ResponseState.FAILURE;
           objResponse.hasError = true;
           objResponse.requestType = Response.RequestType.SEARCH_FOR_INDICADOR;

           indicadorEconomicoApiEndPoint =
RetrofitClient.getRetrofitInstance().create(IndicadorEconomicoApi.class);
           objCall =
indicadorEconomicoApiEndPoint.getIndicadorEconomico(tipoIndicador,
fechaIndicador);

           handleCallBack(objRestRequestInterface, objCall, objResponse);
       }
}
```

De lo anterior, podemos crear N handlers instanciando otras interfaces y apis, o N métodos de indicadores económicos en esta misma clase utilizando el mismo procedimiento que reutiliza RetrofitClient, CallHandle y la interfaz asociada, además de nuestra interfaz genérica RequestInterfaceApi para las respuestas.Al ocurrir esto, no tenemos que modificar la clase IndicadorEconomicoHandler para agregar un nuevo call a un api o método distinto a un indicador económico, esto se conoce como **Dependency Inversion Principle**, lo cual nos indica crear un nuevo handler y una nueva interfaz que contengan la lógica de la operación.

10. Finalmente modificaremos nuestra actividad por defecto la "MainActivity" , usando los principios de **Single Responsability (dedicada a las vistas), Open/Closed, Liskov's Substitution, Interface Segregation y Dependency Inversion**, implementando la interfaz RequestInterfaceApi sobrescribiendo su método `Response()`, en el cual además tomaremos la respuesta que viene como objeto y le realizamos un parse hacia nuestro pojo particular `IndicadorEconomico` de la siguiente manera:

```
public class MainActivity extends AppCompatActivity implements
RequestInterfaceApi {

    private static String TAG = "MainActivity";
    private TextView resultadoIndicadores;
    private EditText tipoIndicador, fechaIndicador;

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

        initializeViews();
    }

    private void initializeViews(){
        resultadoIndicadores = findViewById(R.id.resultado);
        tipoIndicador = findViewById(R.id.idTipo);
        fechaIndicador = findViewById(R.id.idFecha);
    }

    public void consultarIndicador(View v){
        try {

            new IndicadorEconomicoHandler(tipoIndicador.getText().toString(),
fechaIndicador.getText().toString()).getIndicadorEconomico(this);
        }catch (Exception e){
            Log.e(TAG, "Error: ", e);
        }
    }

    @SuppressLint("SetTextI18n")
    @Override
    public void Response(Response response) {
        try {
            if (response.state == Response.ResponseState.SUCCESS &&
!response.hasError) {
```

```
                if (response.requestType ==
Response.RequestType.SEARCH_FOR_INDICADOR) {
                    try {

                        Gson gson = new GsonBuilder().create();
                        String jsonObject = gson.toJson(response.result);
                        IndicadorEconomico indicadorEconomico = new
Gson().fromJson(jsonObject, IndicadorEconomico.class);
                        if (indicadorEconomico.getSerie() != null) {
                            if (indicadorEconomico.getSerie().size() > 0) {

resultadoIndicadores.setText(indicadorEconomico.getSerie().get(0).getValor() +
" " + indicadorEconomico.getUnidad_medida());
                            } else {
                                Toast.makeText(getApplicationContext(), "El api
no tiene resultados para esta fecha o tipo de moneda",
Toast.LENGTH_SHORT).show();
                                resultadoIndicadores.setText("");
                            }

                        } else {
                            Toast.makeText(getApplicationContext(), "El api no
tiene resultados", Toast.LENGTH_SHORT).show();
                            resultadoIndicadores.setText("");
                        }
                    }catch (Exception e){
                        Log.e(TAG, "Error: "+e);
                    }
                }

            } else {
                String errorMsg = response.hasError ? response.errorMessage :
"No connection error";
                Toast.makeText(getApplicationContext(), errorMsg,
Toast.LENGTH_SHORT).show();
                Log.e(TAG, errorMsg);
            }
        } catch (Exception objException) {
            Log.e(TAG, objException.getMessage());
        }
    }

}
```

11. El archivo de layout `activity_main.xml` para el funcionamiento de este ejemplo quedaría de la siguiente manera:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.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=".MainActivity">

    <TextView
        android:id="@+id/indicadoresEconomicosTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="INDICADORES ECONOMICOS"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:layout_marginTop="50dp"/>

    <EditText
        android:id="@+id/idTipo"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Ingresa un tipo de moneda por su nombre"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <EditText
        android:id="@+id/idFecha"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Ingresa una fecha"
        app:layout_constraintTop_toBottomOf="@id/idTipo"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent" />
```

```
    <Button
        android:id="@+id/btnAction"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:backgroundTint="@color/colorPrimary"
        android:textColor="#FFFFFF"
        android:text="CONSULTAR"
        android:onClick="consultarIndicador"
        app:layout_constraintTop_toBottomOf="@id/idFecha"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"/>

    <TextView
        android:id="@+id/resultado"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="36dp"
        android:textStyle="bold"
        android:textColor="@color/colorPrimary"
        android:layout_marginTop="50dp"
        app:layout_constraintTop_toBottomOf="@id/indicadoresEconomicosTitle"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        />

</android.support.constraint.ConstraintLayout>
```

12. En nuestro archivo androidManifest.xml nos aseguramos de agregar los permisos de internet, el `usesClearTextTraffic` para versiones nuevas de android dentro de nuestro tag "application" y agregaremos un parámetro dentro de nuestro tag activity que permita ajustar la pantalla cuando ingresemos datos de tipo de moneda y fecha. Veamos de qué se trata en orden secuencial:

```
<uses-permission android:name="android.permission.INTERNET" />
```

```
<application
    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"
    android:usesCleartextTraffic="true">
```

```
<activity android:name=".MainActivity"
    android:windowSoftInputMode="stateVisible|adjustResize">
```

13. Nuestro proyecto al ser iniciado debería de mostrarse como se ven en las siguientes imágenes:

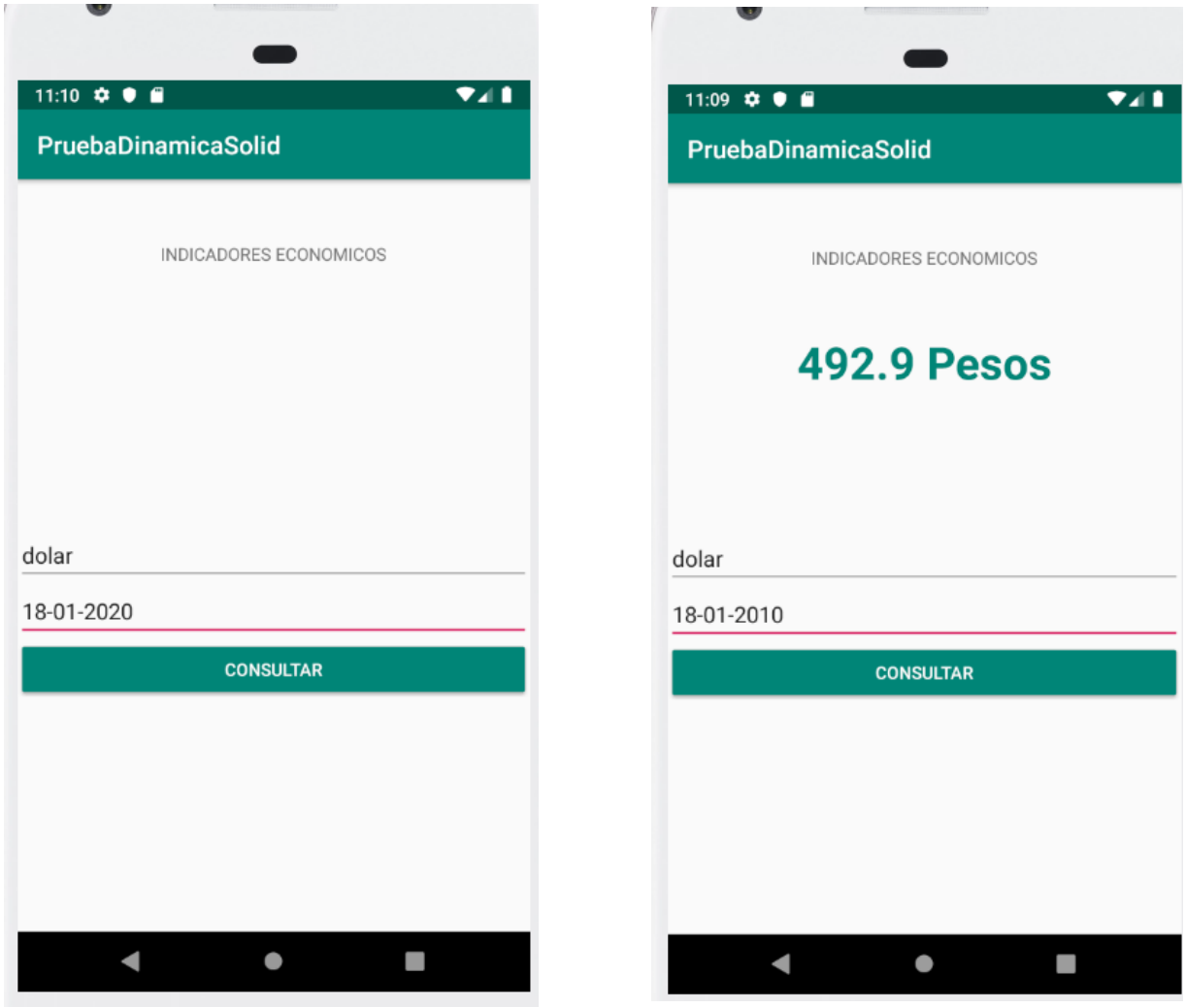

Imagen 10 y 11. Resultado de la aplicación.

El ejercicio anterior, fue un acercamiento a los principios **SOLID**, dónde en resumen, construimos clases con responsabilidades únicas como los pojos, la handlers, interfaces, actividades, etc, aplicamos principios de segregación de interfaces e inversión de dependencias con el manejo de la respuesta y del api a través de sus interfaces y los objetos que las implementan, y también aplicamos el principio Liskov's substitution y open/closed con la clase Callhandle. Si bien hay algunos elementos que quedan por desarrollar, como por ejemplo la lógica en el método Response de la actividad no es precisamente una **Single Responsability**, estaremos profundizando más aún estos principios en el próximo ejercicio conjuntamente con la Clean Architecture.

Clean Architecture

Competencias:

- Aprender qué es Clean Architecture
- Comprender los beneficios de Clean Architecture
- Utilizar un tipo de arquitectura Clean en un proyecto

Introducción

Anticipamos que este contenido es el más importante de esta unidad y en el cual desarrollaremos un ejercicio robusto para conseguir el objetivo de crear un proyecto basado en la arquitectura clean. El ejercicio tendrá un mayor nivel de complejidad y superando este, estaremos en conocimiento y capacidad de crear una arquitectura capaz de funcionar con distintos equipos de trabajo, con separación de capas y con modificaciones y cambios que no alteren en lo absoluto la lógica de nuestro sistema. También una arquitectura clean permite un código más escalable, entendible y mantenible.

No existe una regla específica de la mejor implementación por capas con clean architecture, esto dependerá de los requerimientos del cliente, el alcance del proyecto, los tiempos de entrega, recursos, equipo, entre otros. Clean architecture busca siempre que escribamos un código de alta calidad basados indirectamente en los principios SOLID.

Clean architecture

El patrón Clean architecture es bastante maduro y nos entrega una forma de separación por capas dependendiendo de su responsabilidad en el sistema. La arquitectura nos proporcionará una estructura modular para que diferentes equipos de trabajo puedan trabajar en la misma aplicación sin problemas gracias a esta separación de capas que mantendrá la lógica de negocio protegida, la idea de Clean Architecture es que las capas exteriores solo conozcan la capa interior que la precede.

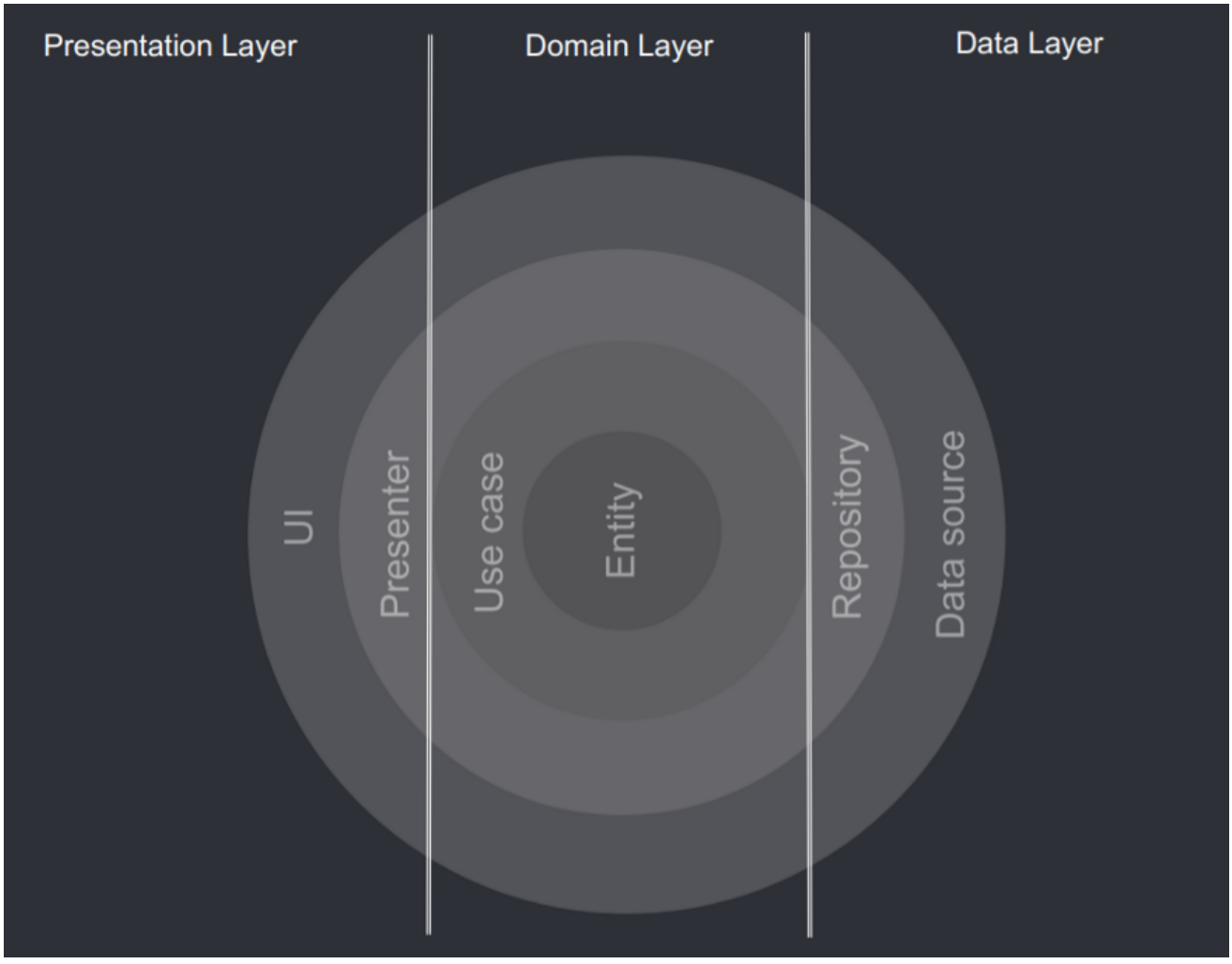

Imagen 12. Separación de capas Clean Architecture

Es importante destacar que siempre se ha de evaluar si se requiere o no de una arquitectura robusta como esta, dependiendo de lo siguiente:

- Si el sistema será trabajado por varias personas y/o equipos es imprescindible aplicar este principio arquitectónico.
- Si el sistema evolucionará rápidamente, su lógica es objeto de cambios continuos, deberás implementar este principio arquitectónico, así seas el único(a) que trabaje con el sistema (por tu bien).

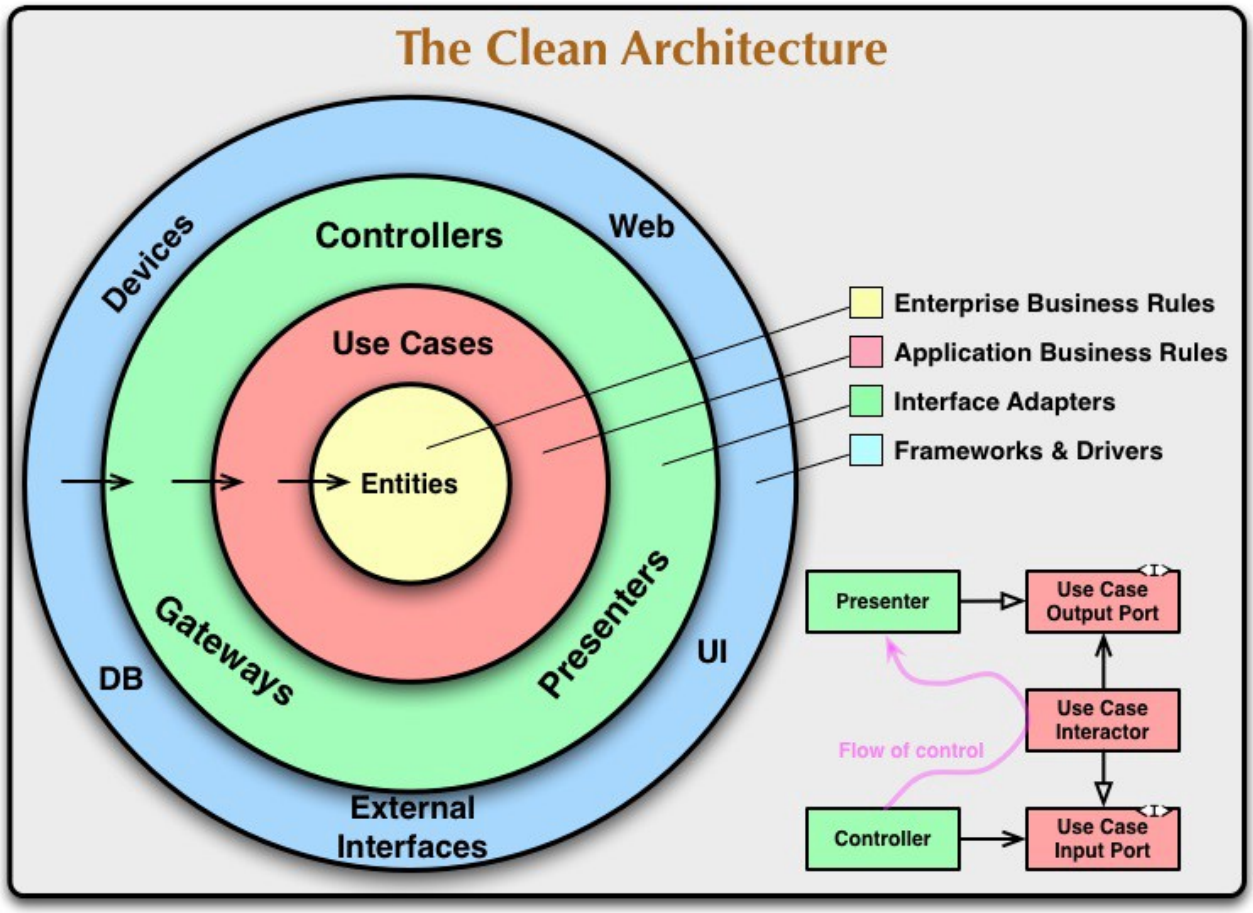

Imagen 13.Diagrama de la arquitectura clean.

Al aplicar Clean architecture en nuestro proyecto, la mejor forma de hacerlo es usando el sentido común para entender lo que estamos haciendo, separa nuestro proyecto de sistema en tres principales capas:

- **Capa de presentación:** aquí se considera toda la lógica de la vista. Se puede implementar un patrón MVP (Model View Presenter), MVC (Model View Controller) o MVVM (Model View ViewModel). Dentro del modelo reside la lógica de negocio, como por ejemplo, la clase IndicadorEconomico, o la clase de Usuario, asegurándonos de que los resultados sean consumido por las vistas. Las clases se prueban utilizando Espresso, Robolectric y muchos otros marcos de prueba de unidades instrumentadas.

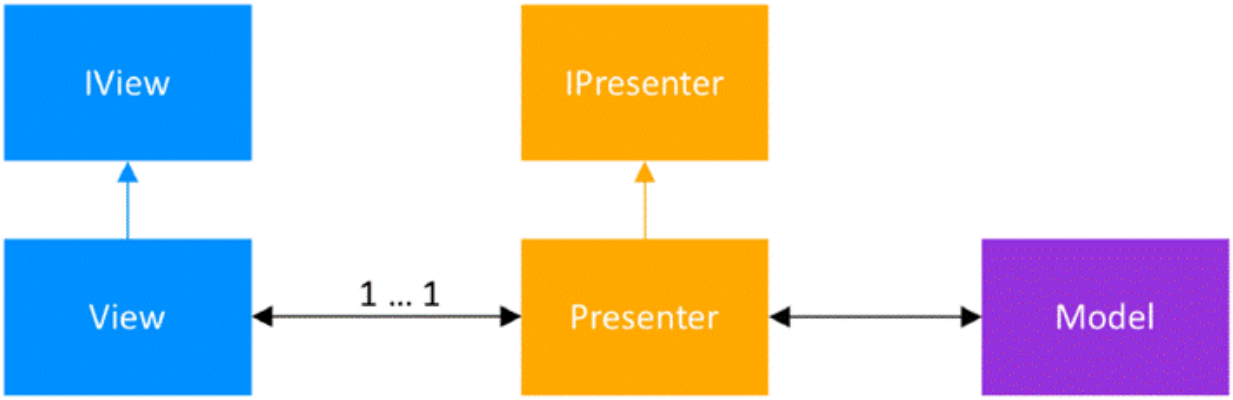

Imagen 14. Capa presentación.

- **Capa de dominio:** es la capa más importante ya que declara y define toda la lógica de negocios que ha de permanecer en el tiempo, los principios y fundamentos con los que opera el sistema. Esta capa es totalmente independiente de cualquier API o framework de Android, es decir, es como desarrollar su aplicación como un simple proyecto de Java (o Kotlin). Puede probar todas las clases aquí con pruebas de unidad locales.
- **Capa de datos:** todos los datos se obtienen en esta capa, además de implementar estrategias sobre el caché, comunicarnos con repositorios de archivos o scripts. SQLite, Realm, Retrofit, Sugar ORM y muchas otras bibliotecas se utilizan en la capa de datos.

A continuación explicaremos en detalle la imagen 13, que no es otra cosa que un diagrama que envuelve distintas arquitecturas en principios generales.

Entidades

Las entidades encapsulan las reglas de negocio de toda la empresa. Una entidad puede ser un objeto con métodos, o puede ser un conjunto de funciones y estructuras de datos, esto no importa, siempre y cuando las entidades puedan ser utilizadas por muchas aplicaciones diferentes en la empresa.

Si no se trata de una empresa y solo estamos escribiendo una aplicación, estas entidades son los objetos comerciales de la aplicación. Encapsulan las reglas más generales y de alto nivel. Es menos probable que cambien cuando algo externo cambia, por ejemplo, no esperamos que estos objetos se vieran afectados por un cambio en la navegación de la página o por la seguridad. Ningún cambio operacional en ninguna aplicación en particular debe afectar la capa de la entidad.

Casos de uso

El software en esta capa contiene reglas de negocio específicas de la aplicación. Encapsula e implementa todos los casos de uso del sistema. Estos casos de uso orquestan el flujo de datos hacia y desde las entidades, y dirigen a esas entidades a utilizar sus reglas de negocios en toda la empresa para lograr los objetivos del caso de uso.

No esperamos que los cambios en esta capa afecten a las entidades, tampoco esperamos que esta capa se vea afectada por cambios en el software externo, como la base de datos, la interfaz de usuario o cualquiera de los marcos comunes. Esta capa está aislada de tales preocupaciones.

Sin embargo, esperamos que los cambios en el funcionamiento de la aplicación afecten los casos de uso y, por lo tanto, el software en esta capa. Si los detalles de un caso de uso cambian, entonces algunos códigos en esta capa ciertamente se verán afectados.

Adaptadores de interfaz

El software en esta capa es un conjunto de adaptadores que convierten los datos del formato más conveniente para los casos de uso y las entidades, al formato más conveniente para otro software externo, como la base de datos o la web. Los presentadores, vistas y controladores pertenecen todos aquí. Es probable que los modelos sean solo estructuras de datos que pasan de los controladores a los casos de uso, y luego regresan de los casos de uso a los presentadores y las vistas.

De manera similar, los datos se convierten, en esta capa, de la forma más conveniente para las entidades y casos de uso, a la forma más conveniente para cualquier estructura de persistencia (base de datos) que se esté utilizando. Si la base de datos es una base de datos SQL, entonces todo el SQL debe estar restringido a esta capa, y en particular a las partes de esta capa que tienen que ver con la base de datos.

También en esta capa hay cualquier otro adaptador necesario para convertir los datos de algún formulario externo, como un servicio externo, al formulario interno utilizado por los casos de uso y las entidades.

Marcos y controladores

La capa más externa generalmente se compone de marcos y herramientas como la base de datos, el marco web (apis de servicios), etc. Generalmente, no se escribe mucho código en esta capa, solo se escribe el código que se comunica con los adaptadores de interfaz.

Esta capa es donde va toda la comunicación con el software externo. La web es un ejemplo de ello si estamos consumiendo un servicio desde una dirección url externa a nuestra aplicación. La base de datos también es un software externo, tiene sus propias reglas de conexión y acceso.

Beneficios de usar Clean Architecture

Cumplir con estas reglas simples de la Clean Architecture no es difícil, y nos ahorrará muchos dolores de cabeza en el futuro. Al separar el software en capas, desarrollaremos sistemas eficientes, escalables y fácil de auditar, con todos los beneficios que esto implica.

Cuando cualquiera de las partes externas del sistema se vuelven obsoletas, como la base de datos o el marco web, puede reemplazar esos elementos obsoletos con un mínimo de esfuerzo.

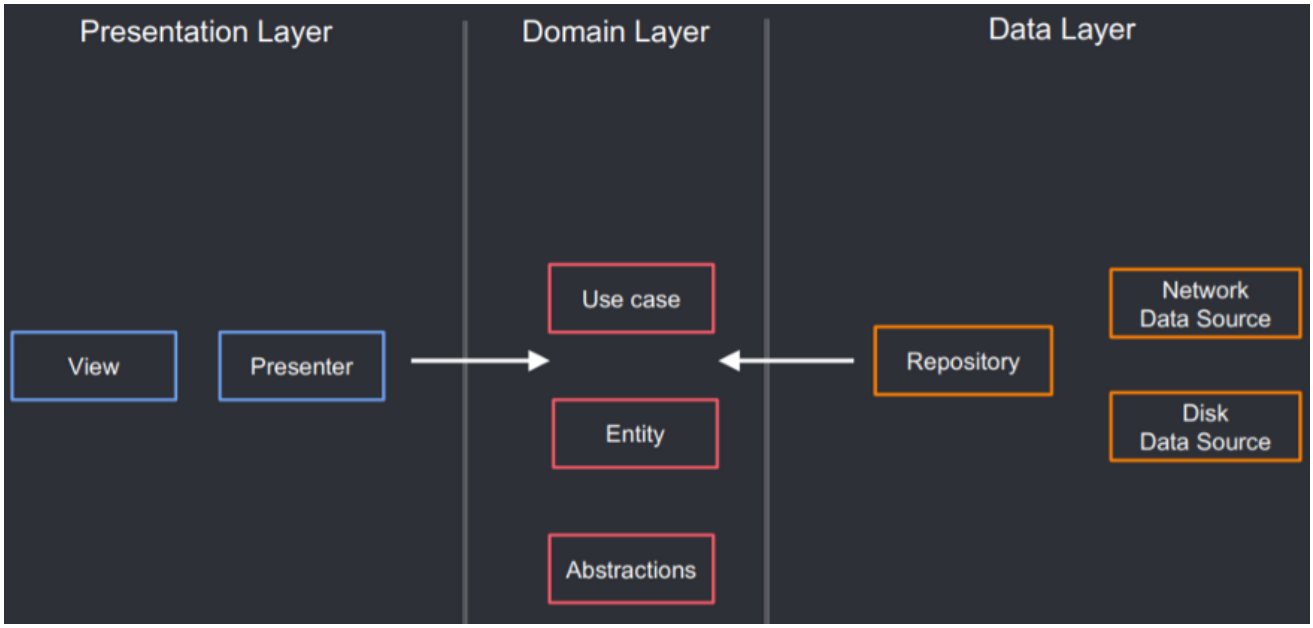

Imagen 15. Máxima abstracción basada en composición.

Ejercicio 2

Creación de proyecto "IndicadoresEconomicosChile" basados en el ejercicio 1, implementando la arquitectura clean con el patrón de diseño MVP.

1. Creamos un nuevo proyecto llamado "IndicadoresEconomicosChile" el cual tendrá una organización y estructura basada en la Clean Architecture junto con el patrón de diseño MVP, reutilizando la lógica de nuestro proyecto del capítulo anterior Solid.

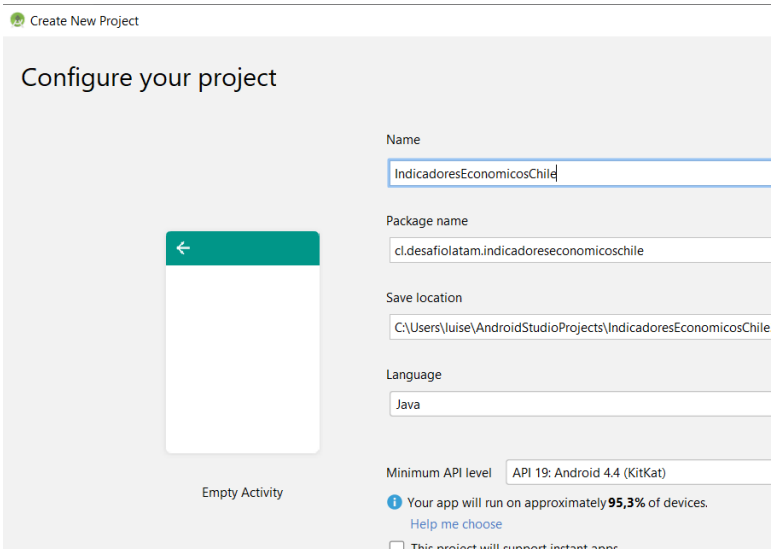

Imagen 16.Nuevo proyecto

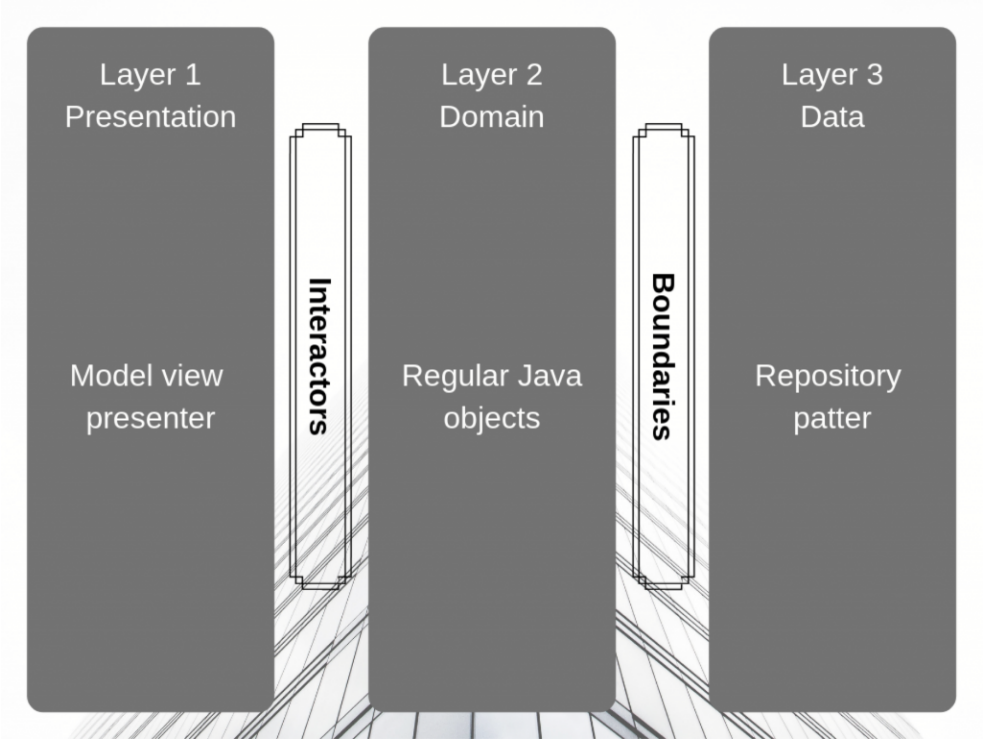

Imagen 17. Arquitectura clean Propuesta para el ejercicio

2. En el proyecto desde nuestra carpeta raíz "cl.desafiolatam.indicadoresclean" crearemos tres nuevos package, data, domain y presentation. Cada uno de estos package harán una representación de separación de capas.

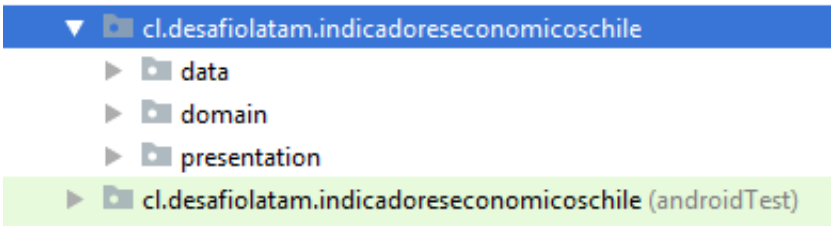

Imagen 18. Agregando los paquetes nuevos

3. Agregamos nuestras librerías retrofit y gson al app gradle:

```
implementation 'com.squareup.retrofit2:retrofit:2.6.0'
implementation 'com.squareup.retrofit2:converter-gson:2.6.0'
```

4. Agregamos reactiveX a nuestro proyecto, librerías muy utilizadas en la actualidad con los patrones MVx, esto nos brinda un mayor acercamiento a la arquitectura clean ya que nos facilita las abstracciones entre las capas, la transformación de datos entre objetos de modelo y nos ahorra usar muchos listeners.

```
implementation 'com.jakewharton.retrofit:retrofit2-rxjava2-adapter:1.0.0'
implementation 'io.reactivex.rxjava2:rxjava:2.0.2'
implementation 'io.reactivex.rxjava2:rxandroid:2.0.1'
```

5. Editamos nuestro archivo de layout `activity_main.xml` con el mismo contenido del ejercicio del capítulo "Principios SOLID". En esta ocasión agregaremos un ProgressBar.

```
<android.support.constraint.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=".presentation.view.MainActivity">
   <TextView
       android:id="@+id/indicadoresEconomicosTitle"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="INDICADORES ECONOMICOS"
       app:layout_constraintLeft_toLeftOf="parent"
       app:layout_constraintRight_toRightOf="parent"
       app:layout_constraintTop_toTopOf="parent"
       android:layout_marginTop="50dp"/>
```

```
<EditText
    android:id="@+id/idTipo"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Ingresa moneda, ej: dolar, euro"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<EditText
    android:id="@+id/idFecha"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Ingresa una fecha dd-mm-yyyy"
    app:layout_constraintTop_toBottomOf="@id/idTipo"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent" />

<Button
    android:id="@+id/btnAction"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:backgroundTint="@color/colorPrimary"
    android:textColor="#FFFFFF"
    android:text="CONSULTAR"
    app:layout_constraintTop_toBottomOf="@id/idFecha"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"/>

<TextView
    android:id="@+id/resultado"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="36dp"
    android:textStyle="bold"
    android:textColor="@color/colorPrimary"
    android:layout_marginTop="50dp"
    app:layout_constraintTop_toBottomOf="@id/indicadoresEconomicosTitle"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    />
```

```
    <ProgressBar
        android:id="@+id/progress"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="50dp"
        app:layout_constraintTop_toBottomOf="@id/indicadoresEconomicosTitle"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent" />

</android.support.constraint.ConstraintLayout>
```

6. **Data Layer**. En este punto comenzamos el trabajo sobre la capa **data**, la cual será la encargada de la captura de datos de nuestra api http://www.indicadores.cl , creando los siguientes objetos:

- **SerieIndicadorSchema**

  Objeto tipo modelo de negocios que guarda los datos en la estructura de la respuesta del api. Lo llamamos Schema para acercarlo más a su origen de datos a partir de servicios web y no de una base de datos tipo entidad.

```
public class SerieIndicadorEconomicoSchema {
    @SerializedName("fecha")
    private String fecha;
    @SerializedName("valor")
    private double valor;
    public SerieIndicadorEconomicoSchema(String fecha, double valor){
        this.fecha = fecha;
        this.valor = valor;
    }
    public String getFecha() {
        return fecha;
    }
    public void setFecha(String fecha) {
        this.fecha = fecha;
    }
    public double getValor() {
        return valor;
    }
    public void setValor(double valor) {
        this.valor = valor;
    }
}
```

- **IndicadorEconomicoSchema**

  La misma definición que el anterior, solo que este objeto de negocio pertenece a la capa más alta de la respuesta del api.

```
public class IndicadorEconomicoSchema {

   @SerializedName("nombre")
   private String nombre;
   @SerializedName("unidad_medida")
   private String unidadMedida;
   @SerializedName("serie")
   private ArrayList<SerieIndicadorEconomicoSchema> serie;

   public IndicadorEconomicoSchema(String nombre, String unidadMedida,
ArrayList<SerieIndicadorEconomicoSchema> serie){
        this.nombre = nombre;
        this.unidadMedida = unidadMedida;
        this.serie = serie;
   }
   public String getNombre() {
        return nombre;
   }
   public void setNombre(String nombre) {
        this.nombre = nombre;
   }
   public String getUnidadMedida() {
        return unidadMedida;
   }
   public void setUnidadMedida(String unidadMedida) {
        this.unidadMedida = unidadMedida;
   }
   public ArrayList<SerieIndicadorEconomicoSchema> getSerie() {
        return serie;
   }
   public void setSerie(ArrayList<SerieIndicadorEconomicoSchema> serie)    {
        this.serie = serie;
   }
}
```

- **IndicadorEconomicoMapper**

  Objeto responsable de transformar los objetos de datos entre capas para retornar a la capa de más alto nivel (Domain) el objeto esperado a través de su interfaz de composición.

```
public class IndicadorEconomicoMapper {

   public IndicadorEconomicoMapper(){}

   public IndicadorEconomico transform(IndicadorEconomicoSchema
indicadorEconomicoSchema){
       IndicadorEconomico indicadorEconomico = null;

       try {
           if (indicadorEconomicoSchema != null) {
               SerieIndicadorEconomico serieIndicadorEconomico = null;
               serieIndicadorEconomico = new
SerieIndicadorEconomico(indicadorEconomicoSchema.getSerie().get(0).getFecha(),
                       indicadorEconomicoSchema.getSerie().get(0).getValor());
               ArrayList<SerieIndicadorEconomico> listSerieIndicadorEconomico =
new ArrayList<SerieIndicadorEconomico>();
               listSerieIndicadorEconomico.add(serieIndicadorEconomico);

               indicadorEconomico = new
IndicadorEconomico(indicadorEconomicoSchema.getNombre(),
                       indicadorEconomicoSchema.getUnidadMedida(),
                       listSerieIndicadorEconomico);
           }
       }catch (Exception e){
           return new IndicadorEconomico("", "", null);
       }
       return indicadorEconomico;
   }
}
```

- **IIndicadoresEconomicosApi**

  Interfaz de composición que completa la ruta web del servicio del api con datos de entrada y define el objeto de respuesta `Call<IndicadorEconomicoSchema>`.

```
public interface IIndicadoresEconomicosApi {
       @GET("{tipoIndicador}/{fechaIndicador}")
       Observable<IndicadorEconomicoSchema>
getIndicadorEconomico(@Path("tipoIndicador") String tipoIndicador,
@Path("fechaIndicador") String fechaIndicador);
}
```

- **RetrofitClient**

  Clase en cargada de crear y devolver la instancia del cliente que construye el llamado http al api con la librería retrofit.

```
public class RetrofitClient {

       private static Retrofit retrofit;
       private static final String BASE_URL = "https://mindicador.cl/api/";

       public static Retrofit getRetrofitInstance(){
           if (retrofit == null) {
               retrofit = new retrofit2.Retrofit.Builder()
                       .baseUrl(BASE_URL)
.addCallAdapterFactory(RxJava2CallAdapterFactory.create())
.addConverterFactory(GsonConverterFactory.create())
                       .build();
           }
           return retrofit;
       }
}
```

- **IndicadoresEconomicosService**

  Por último, crearemos esta clase con una sencilla responsabilidad de crear o regresar una instancia de la interfaz del api indicadores, a través de su cliente retrofit.

```
public class IndicadoresEconomicosService {

       private static IIndicadoresEconomicosApi mIindicadoresEconomicosApi;

       public static IIndicadoresEconomicosApi
getIndicadoresEconomicosService(){
           if(mIindicadoresEconomicosApi==null){
               mIindicadoresEconomicosApi =
RetrofitClient.getRetrofitInstance().create(IIndicadoresEconomicosApi.class);
           }
           return mIindicadoresEconomicosApi;
       }
}
```

7. **Domain Layer**Esta capa contendrá la lógica de más alto nivel asociada a nuestros objetos de negocio y casos de uso. A continuación sus objetos:

- **IndicadorEconomico**

  Al igual que el objeto IndicadorEconomicoSchema, pero de uso exclusivo de la capa de negocio. Creamos un package llamado "model" dentro de domain.

```
public class IndicadorEconomico {

        private String nombre;
        private String unidadMedida;
        private ArrayList<SerieIndicadorEconomico> serie;

        public IndicadorEconomico(String nombre, String unidadMedida,
ArrayList<SerieIndicadorEconomico> serie) {
            super();
            this.nombre = nombre;
            this.unidadMedida = unidadMedida;
            this.serie = serie;
        }

        public String getNombre() {
            return nombre;
        }
```

```
        public void setNombre(String nombre) {
            this.nombre = nombre;
        }

        public String getUnidadMedida() {
            return unidadMedida;
        }

        public void setUnidadMedida(String unidad_medida) {
            this.unidadMedida = unidad_medida;
        }

        public ArrayList<SerieIndicadorEconomico> getSerie() {
            return serie;
        }

        public void setSerie(ArrayList<SerieIndicadorEconomico> serie) {
            this.serie = serie;
        }
    }
```

- **SerieIndicadorEconomico**

  Al igual que el objeto SerieIndicadorEconomicoSchema de data, pero en este caso, de uso exclusivo de la capa domain.

```
public class SerieIndicadorEconomico {

   private String fecha;
   private double valor;

   public SerieIndicadorEconomico(String fecha, double valor) {
       super();
       this.fecha = fecha;
       this.valor = valor;
   }

   public String getFecha() {
       return fecha;
   }

   public void setFecha(String fecha) {
       this.fecha = fecha;
   }
```

```
    public double getValor() {
        return valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }
}
```

- **IndicadorEconomicoRepository**

  Creamos una interfaz observable que nos asegura la no dependencia de la capa domain de la capa data, siguiendo los principios de la clean architecture. Esto nos asegura que a futuro, podemos cambiar la implementación en la capa "data" sin tener que cambiar la capa "domain" ya que esta mantiene su abstracción a través de esta interfaz, la cual obtiene los datos del api para el negocio.

```
public interface IIndicadorEconomicoRepository {
       Observable<IndicadorEconomico> getIndicadorEconomicoFromApiLayer(String
tipoIndicador, String fechaIndicador);
}
```

- **UseCase**

  Clase base de tipo abstracta que implementa los métodos firma como standard para los casos de uso a crear por funcionalidad específica.

```
abstract class UseCase<T, Params> {

    private final CompositeDisposable disposables;

    UseCase() {
        this.disposables = new CompositeDisposable();
    }

    abstract Observable<T> buildUseCaseObservable(Params params);


    /**
     * ejecuta el use case en curso.
     * @param observer {@link DisposableObserver} parametro que queda escuchando
el build del observer
     * by {@link #buildUseCaseObservable(Params)} ()} metódo de la clase.
     * @param params Parameters (Optional) usados para ejecutar el caso de uso
     */
```

```
    public void execute(DisposableObserver<T> observer, Params params) {

        final Observable<T> observable = this.buildUseCaseObservable(params)
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread());

        addDisposable(observable.subscribeWith(observer));
    }

    public void dispose() {
        if (!disposables.isDisposed()) {
            disposables.dispose();
        }
    }

    private void addDisposable(Disposable disposable) {
        disposables.add(disposable);
    }
}
```

- **GetIndicadorEconomicoUseCase**

  Clase que encierra la funcionalidad de captura de los datos de indicador económico, extendiendo de la base UseCase e implementando un método que observa cuando la interfaz IIndicadorEconomicoRepository obtiene datos desde el api.

```
public class GetIndicadorEconomicoUseCase extends UseCase<IndicadorEconomico,
String[]>{

       private final IIndicadorEconomicoRepository
iIndicadorEconomicoRepository;

       public GetIndicadorEconomicoUseCase(IIndicadorEconomicoRepository
iIndicadorEconomicoRepository){
           this.iIndicadorEconomicoRepository = iIndicadorEconomicoRepository;
       }
       @Override
       Observable<IndicadorEconomico> buildUseCaseObservable(String[] input) {
           return
iIndicadorEconomicoRepository.getIndicadorEconomicoFromApiLayer(input[0],
input[1]);
       }
}
```

Con esto concluímos la implementación de los objetos de la capa Domain, sin embargo, vamos a necesitar realizar una última implementación basada en el patrón Repository para que sus objetos sean los encargados de convertir de una capa a otra los datos según sean sus modelos y de transferir la respuesta del api.

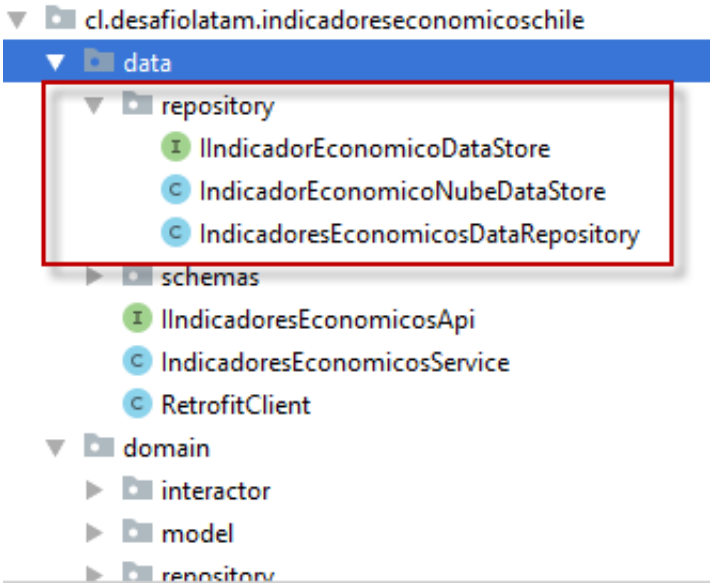

Imagen 19. Capa de datos

En la capa Data para que funcione adecuadamente la arquitectura clean, listamos los objetos a crear a continuación:

- IIndicadorEconomicoDataStore Interfaz de apoyo a la composición para la inversión de dependencia.

```
public interface IIndicadorEconomicoDataStore {
   Observable<IndicadorEconomicoSchema> getIndicadorEconomicoDataSchema(String
tipoIndicador, String fechaIndicador);
}
```

- **IndicadorEconomicoNubeDataStore**

  Este objeto implementa la interfaz anterior, retornando el el objeto asíncrono que devuelve el cliente retrofit contra el api.

```
public class IndicadorEconomicoNubeDataStore implements
IIndicadorEconomicoDataStore{

   @Override
   public Observable<IndicadorEconomicoSchema>
getIndicadorEconomicoDataSchema(String tipoIndicador, String fechaIndicador) {
       return IndicadoresEconomicosService.getIndicadoresEconomicosService()
               .getIndicadorEconomico(tipoIndicador, fechaIndicador);
   }
}
```

- **IndicadoresEconomicosDataRepository**

  Este objeto es muy importante ya que realiza la implementación de la interfaz IIndicadorEconomicoRepository de la capa Domain en su abstracción de composición y realiza la consulta de los datos a través de IndicadorEconomicoNubeDataStore, para finalizar retornando un objeto ya transformado a lo requerido por la capa Domain, en este caso el objeto `IndicadorEconomicoSchema` de Data , se mapeo o transformo a "IndicadorEconomico" de Domain.

```
public class IndicadoresEconomicosDataRepository implements
IIndicadorEconomicoRepository {
   private final IndicadorEconomicoNubeDataStore
indicadorEconomicoNubeDataStore;
   private final IndicadorEconomicoMapper indicadorEconomicoMapper;

   public IndicadoresEconomicosDataRepository(IndicadorEconomicoNubeDataStore
indicadorEconomicoNubeDataStore,
                                              IndicadorEconomicoMapper
indicadorEconomicoMapper){
       this.indicadorEconomicoNubeDataStore = indicadorEconomicoNubeDataStore;
       this.indicadorEconomicoMapper = indicadorEconomicoMapper;
   }
```

```
    @Override
    public Observable<IndicadorEconomico>
getIndicadorEconomicoFromApiLayer(String tipoIndicador, String fechaIndicador){

        return indicadorEconomicoNubeDataStore
                .getIndicadorEconomicoDataSchema(tipoIndicador, fechaIndicador)
                .map(new Function<IndicadorEconomicoSchema, IndicadorEconomico>
() {

                @Override
                public IndicadorEconomico apply(IndicadorEconomicoSchema
indicadorEconomicoSchema) throws Exception {
                    return
indicadorEconomicoMapper.transform(indicadorEconomicoSchema);
                }
            });
    }
}
```

Con el código anterior hemos finalizado el trabajo sobre la capa de negocio de más alto nivel la Domain. En este punto aclararemos porque se construyen tantos objetos y se ha incrementado la complejidad respecto al ejercicio 1; pues al construir una Clean Architecture, aseguramos la división de las capas, para que puedan ser trabajadas por múltiples equipos sobre el mismo proyecto sin afectarse entre sí. Además, toda implementación que se deba agregar o cambiar, se consigue con cambiar una o dos líneas de código, cosa que no sucede con un sistema sin arquitectura clean.

8. A continuación, implementaremos el patrón MVP sobre la capa View (presentación) de nuestra arquitectura Clean.

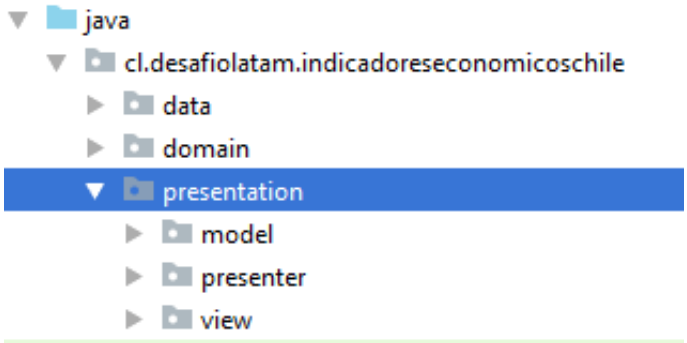

Imagen 20. Capa presentación

9. Comencemos por el model, donde crearemos los objetos siguientes:

- **IndicadorEconomicoModel**

  Objeto que retiene los datos desde otras capas en la actual.

```
public class IndicadorEconomicoModel {

        private String nombre;
        private String unidadMedida;
        private ArrayList<SerieIndicadorEconomicoModel> serie;

        public IndicadorEconomicoModel(String nombre, String unidadMedida,
ArrayList<SerieIndicadorEconomicoModel> serie) {
            super();
            this.unidadMedida = unidadMedida;
            this.serie = serie;
        }

        public String getNombre() {
            return nombre;
        }

        public void setNombre(String nombre) {
            this.nombre = nombre;
        }

        public String getUnidadMedida() {
            return unidadMedida;
        }

        public void setUnidadMedida(String unidad_medida) {
            this.unidadMedida = unidad_medida;
        }

        public ArrayList<SerieIndicadorEconomicoModel> getSerie() {
            return serie;
        }

        public void setSerie(ArrayList<SerieIndicadorEconomicoModel> serie) {
            this.serie = serie;
        }
}
```

- **SerieIndicadorEconomicoModel**

  Objeto que al igual que el anterior retiene los datos que vienen de las otras capas.

```
public class SerieIndicadorEconomicoModel {

    private String fecha;
    private double valor;

    public SerieIndicadorEconomicoModel(String fecha, double valor) {
        super();
        this.fecha = fecha;
        this.valor = valor;
    }

    public String getFecha() {
        return fecha;
    }

    public void setFecha(String fecha) {
        this.fecha = fecha;
    }

    public double getValor() {
        return valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }
}
```

- **IndicadorEconomicoModelMapper**

  Objeto que transforma el objeto de datos que llega de la capa Domain a los objetos locales de la capa creados recientemente.

```
public class IndicadorEconomicoModelMapper {

   public IndicadorEconomicoModel transform(IndicadorEconomico
indicadorEconomicoDomain){
       IndicadorEconomicoModel indicadorEconomico = null;
       try {
           if (indicadorEconomicoDomain != null) {
               SerieIndicadorEconomicoModel serieIndicadorEconomico = null;
               serieIndicadorEconomico = new
SerieIndicadorEconomicoModel(indicadorEconomicoDomain.getSerie().get(0).getFech
a(),
                       indicadorEconomicoDomain.getSerie().get(0).getValor());
               ArrayList<SerieIndicadorEconomicoModel>
listSerieIndicadorEconomico = new ArrayList<SerieIndicadorEconomicoModel>();
               listSerieIndicadorEconomico.add(serieIndicadorEconomico);

               indicadorEconomico = new
IndicadorEconomicoModel(indicadorEconomicoDomain.getNombre(),
                       indicadorEconomicoDomain.getUnidadMedida(),
                       listSerieIndicadorEconomico);
           }
       }catch (Exception e){
           return new IndicadorEconomicoModel("", "", null);
       }
       return indicadorEconomico;
   }
}
```

10. A continuación crearemos una interfaz en la subcapa de vista del MVP, que vamos a usar para la implementación de nuestros objetos presenter.

- **IIndicadorEconomicoView**

  interfaz que define tres eventos, la respuesta de los datos consultados que viajaron desde las otras capas, y mostrar o no un spinner de carga al momento de consultar.

```
public interface IIndicadorEconomicoView {

   void showResponseIndicadorEconomico(IndicadorEconomicoModel
indicadorEconomicoModel);
   void showLoadSpinner();
   void hideLoadSpinner();
}
```

11. SubCapa Presenter, a continuación los objetos a desarrollar para conseguir el patrón MVP:

- **Presenter**

  Interfaz que define dos eventos, startConsulta y stopListeners.

```
public interface Presenter {

   void startConsulta(String tipoIndicador, String fechaIndicador);
   void stopListeners();
}
```

- **IndicadorEconomicoPresenter**

  Implementación del presenter en nuestro modelo MVP, que no es más que una especie de controlador sin serlo, porque nos basamos en las Views de android, a diferencia del patrón MVC.

```
public class IndicadorEconomicoPresenter implements Presenter{

   private IIndicadorEconomicoView mIIndicadorEconomicoView;
   private final GetIndicadorEconomicoUseCase getIndicadorEconomicoUseCase;
   private final IndicadorEconomicoModelMapper indicadorEconomicoModelMapper;

   public IndicadorEconomicoPresenter(
       GetIndicadorEconomicoUseCase getIndicadorEconomicoUseCase,
             IndicadorEconomicoModelMapper indicadorEconomicoModelMapper){
       this.getIndicadorEconomicoUseCase = getIndicadorEconomicoUseCase;
       this.indicadorEconomicoModelMapper = indicadorEconomicoModelMapper;
   }
```

```
    public void bindIIndicadorEconomicoView(IIndicadorEconomicoView
iIndicadorEconomicoView){
        this.mIIndicadorEconomicoView = iIndicadorEconomicoView;
    }

    @Override
    public void startConsulta(String tipoIndicador, String fechaIndicador){
        this.showLoading();
        String[] params = new String[2];
        params[0] = tipoIndicador;
        params[1] = fechaIndicador;
        this.getIndicadorEconomico(params);
    }

    @Override
    public void stopListeners() {
        this.getIndicadorEconomicoUseCase.dispose();
        this.mIIndicadorEconomicoView.hideLoadSpinner();
        this.mIIndicadorEconomicoView = null;
    }

    private void getIndicadorEconomico(String[] params) {
        this.getIndicadorEconomicoUseCase.execute(new
IndicadorEconomicoObserverForPresenter(this), params);
    }

    public void showIndicadorEconomicoResponse(IndicadorEconomico
indicadorEconomico) {
        this.hideLoading();
        final IndicadorEconomicoModel indicadorEconomicoModel =

this.indicadorEconomicoModelMapper.transform(indicadorEconomico);

this.mIIndicadorEconomicoView.showResponseIndicadorEconomico(indicadorEconomico
Model);
    }

    public void showLoading() {
        this.mIIndicadorEconomicoView.showLoadSpinner();
    }

    public void hideLoading() {
        this.mIIndicadorEconomicoView.hideLoadSpinner();
    }
}
```

- **IndicadorEconomicoObserverForPresenter**

  Una clase de apoyo para la lectura de eventos asíncronos.

```
public class IndicadorEconomicoObserverForPresenter extends
DisposableObserver<IndicadorEconomico> {

   private IndicadorEconomicoPresenter indicadorEconomicoPresenter;

           public
IndicadorEconomicoObserverForPresenter(IndicadorEconomicoPresenter
indicadorEconomicoPresenter) {
               this.indicadorEconomicoPresenter = indicadorEconomicoPresenter;
           }
       @Override public void onComplete(){
           indicadorEconomicoPresenter.hideLoading();
       }
       @Override public void onError(Throwable e) {
           e.printStackTrace();
           indicadorEconomicoPresenter.hideLoading();
       }
       @Override public void onNext(IndicadorEconomico indicadorEconomico) {

indicadorEconomicoPresenter.showIndicadorEconomicoResponse(indicadorEconomico);
       }
}
```

12. Ya nos queda solo la subcarpeta view de nuestro patrón MVP, el cual contendrá los siguientes objetos de vista:

- **MainActivity**

  una actividad base abstracta para servir la mesa de una implementación de la vista de consulta de indicadores económicos.

```
public abstract class MainActivity extends Activity {

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(getContentLayoutView());

       ProgressBar progressBar = findViewById(R.id.progress);
       progressBar.setVisibility(View.INVISIBLE);
       EditText tipoIndicador = findViewById(R.id.idTipo);
       EditText fechaIndicador = findViewById(R.id.idFecha);
       Button consultarIndicador = findViewById(R.id.btnAction);
       TextView resultadoConsulta = findViewById(R.id.resultado);

       initializeComponentsForView(progressBar,
               tipoIndicador,
               fechaIndicador,
               consultarIndicador,
               resultadoConsulta);
   }

   public abstract int getContentLayoutView();

   public abstract void initializeComponentsForView(ProgressBar
progressBar,  EditText tipoIndicador,  EditText fechaIndicador, Button
consultarIndicador, TextView resultadoConsulta );

}
```

- **ConsultaIndicadorEconomicoActivity**

  Nuestra implementación de la vista en donde básicamente se inicializan los objetos que componen nuestra arquitectura para la vista.

```
public class ConsultaIndicadorEconomicoActivity extends MainActivity implements
IIndicadorEconomicoView, View.OnClickListener {

   private ProgressBar progressBar;
   private EditText tipoIndicador, fechaIndicador;
   private TextView resultadoConsulta;
   private IndicadorEconomicoPresenter mIndicadorEconomicoPresenter;
   private IndicadoresEconomicosDataRepository
mIndicadoresEconomicosDataRepository;
   private IndicadorEconomicoNubeDataStore mIndicadorEconomicoNubeDataStore;
   private IndicadorEconomicoMapper mIndicadorEconomicoMapper;
   private GetIndicadorEconomicoUseCase mGetIndicadorEconomicoUseCase;
   private IndicadorEconomicoModelMapper mIndicadorEconomicoModelMapper;

   @Override
   public void showLoadSpinner() {
       resultadoConsulta.setText("");
       progressBar.setVisibility(View.VISIBLE);
   }

   @Override
   public void hideLoadSpinner() {
      progressBar.setVisibility(View.GONE);
   }

   @Override
   public int getContentLayoutView() {
       return R.layout.activity_main;
   }

   @Override
   public void initializeComponentsForView(ProgressBar progressBar,
                                           EditText tipoIndicador,
                                           EditText fechaIndicador,
                                           Button btnConsultarIndicador,
                                           TextView resultadoConsulta) {
       this.progressBar = progressBar;
       this.tipoIndicador = tipoIndicador;
       this.fechaIndicador = fechaIndicador;
       this.resultadoConsulta = resultadoConsulta;
       btnConsultarIndicador.setOnClickListener(this);
```

```
        mIndicadorEconomicoModelMapper = new IndicadorEconomicoModelMapper();
        mIndicadorEconomicoMapper = new IndicadorEconomicoMapper();
        mIndicadorEconomicoNubeDataStore = new
IndicadorEconomicoNubeDataStore();
        mIndicadoresEconomicosDataRepository = new
IndicadoresEconomicosDataRepository(mIndicadorEconomicoNubeDataStore,

         mIndicadorEconomicoMapper);
        mGetIndicadorEconomicoUseCase = new
GetIndicadorEconomicoUseCase(mIndicadoresEconomicosDataRepository);
        mIndicadorEconomicoPresenter = new
IndicadorEconomicoPresenter(mGetIndicadorEconomicoUseCase,
mIndicadorEconomicoModelMapper);
        mIndicadorEconomicoPresenter.bindIIndicadorEconomicoView(this);
    }

    @Override
    public void showResponseIndicadorEconomico(IndicadorEconomicoModel
indicadorEconomicoModel) {
        resultadoConsulta.setText(indicadorEconomicoModel.getSerie() != null ?
                indicadorEconomicoModel.getSerie().get(0).getValor()
                        + " "+indicadorEconomicoModel.getUnidadMedida()
                : "Fecha sin datos");
    }

    @Override
    public void onClick(View v) {

mIndicadorEconomicoPresenter.startConsulta(tipoIndicador.getText().toString(),
fechaIndicador.getText().toString());
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        mIndicadorEconomicoPresenter.stopListeners();
    }
}
```

13. No olvidemos agregar los siguientes atributos de permisos y ajustes de pantalla con el teclado, a nuestro archivo Manifest:

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

   <uses-permission android:name="android.permission.INTERNET" />

   <application
       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"
       android:usesCleartextTraffic="true">
       <activity
android:name=".presentation.view.ConsultaIndicadorEconomicoActivity"
           android:windowSoftInputMode="adjustResize">
           <intent-filter>
               <action android:name="android.intent.action.MAIN" />

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

</manifest>
```

De lo anterior importante que apunten a la implementación de la vista desde su archivo manifest.

El ejercicio anterior ha sido un esfuerzo por desarrollar un ejemplo de arquitectura clean, basada en sus principios de separación de capas y de dependencia inversa, en donde las capas de más bajo nivel dependen de las de más alto nivel y no viceversa. Además de esto, implementamos en conjunto el patrón MVP para la capa de la vista, siendo un ejercicio muy completo para el aprendizaje y acercamiento a los principios arquitectónicos implementados en este caso con android.