Arquitectura de Software y Testing - Parte II

Inyección de Dependencias

Competencias:

- Entender que es una dependencia en android
- Aprender que es la inyección de dependencias
- Conocer e implementar la librería Dagger para inyección de dependencias

Introducción

La inyección de dependencias es un patrón de diseño comúnmente utilizado en el desarrollo moderno de software para facilitar el cumplimiento del principio Solid de la inversión dependencia.

Manejando un framework como Dagger al desacoplar códigos, podrá escribir código más modular y testeable.

Recordemos que el principio de inversión de dependencia indica que aquellas secciones de nuestra aplicación más importantes no deben nunca depender de aquellas menos importantes, es decir, por lo general, abstraemos por ejemplo el modelo o dominio de nuestra lógica de negocio para que esta nunca dependa de las consultas a base de datos o llamados a servicios web, de tal manera que en un modelo de tres capas tanto la vista como la implementación de base de datos o servicios sean dependientes de una lógica de negocio o modelo de datos.

Dependencias

Una dependencia se puede definir como un objeto que depende de otro objeto, por ejemplo, si la clase A depende de la clase B es un dependencia, y si la clase B cambia, entonces la clase A es afectada por dicho cambio y deberá adaptarse a ello; este es el principio de paradigmas de implementación de dependencias en la programación orientada a objetos (OOP), donde se práctica la dependencia por herencia (cuando una clase extiende de otra) o a través de composiciones usando interfaces que permiten una mayor flexibilidad de la dependencia.

A continuación analicemos el siguiente código:

```
public class User {

private IUserService userProfile = new UserProfile();

public String getUserFullName(String userToken){ String userName =
this.userProfile.getName(userToken);
 {....}
}

}
```

Del código anterior entendemos que la clase `User` requiere la clase `UserProfile` para hacer su trabajo y no puede completar su función sin esta, esto quiere decir que la clase “User” depende de la clase `UserProfile` y que la clase `UserProfile` es una dependencia de la clase `User`”. Como información adicional, la clase `IUserService` sería una interfaz que es implementada en la clase `UserProfile`.

Si bien el ejemplo anterior contiene dependencias, la implementación viola el principio **Open/Closed**, el cual indica que tus clases deberían estar abiertas para extensiones pero cerradas para modificaciones.

Se puede mejorar la implementación anterior con un tipo de dependencia digamos visible, a través de un constructor, por ejemplo:

```
public class User {

private IUserService userProfile;

//Constructor
public User(IUserService userProfile){

this.userProfile = userProfile;
}

public String getUserFullName(String userToken){
  String userName = this.userProfile.getName(userToken);
  {....}
}

}
```

Del código anterior, para cuando necesitemos crear una instancia de la clase User, podemos inyectar una implementación diferente de la interfaz `IUserService`.

```
public class EjemploInyeccionDependenciaBasico {
  User user = new User(IUserService miInterfaz .....)
}
```

Es oportuno explicar una dependencia directa y una dependencia indirecta de los ejemplos anteriores.

Dependencia Directa

Una dependencia directa es simplemente un acceso directo a la instancia del objeto que crea esa dependencia sin intermediarios. Ejemplo:

```
public class User {

  private IUserService userProfile;

  //Constructor
  public User(IUserService userProfile){
    this.userProfile = userProfile; //dependencia directa
  }

  public String getUserFullName(String userToken){
    String userName = this.userProfile.getName(userToken);
    {....}
  }

}
```

Dependencia Indirecta

Una dependencia indirecta es aquella dependencia a la cual tenemos acceso desde otra dependencia más directa. Ejemplo, dada la siguiente clase:

```
public class ProfileUser implements IUserService {
  IData data = new Data();  //Dependencia indirecta
}

public String getName(String userToken){
  return this.data.getName(userToken);
}
```

Entendemos que la clase "User" tiene una dependencia indirecta sobre la clase “Data” a través de su dependencia directa con la clase “ProfileUser”.

Inyección de dependencias

La inyección de dependencias se puede definir un patrón de diseño que implementa el principio de control de inversión (IoC) para resolver la dependencia de los objetos. Entendiendo lo que es una dependencia podemos señalar la inyección de dependencias como la configuración de un objeto del cual dependemos, realizada desde otro objeto externo. Buscando una definición aún más sencilla podemos decir que la inyección de dependencias significa entregar a un objeto sus variables de instancia.

La inyección de dependencias surge del principio denominado “**inversión de dependencia”**, el cual sugiere que las capas no dependan de otras capas de menor nivel, sino más bien de abstracciones.

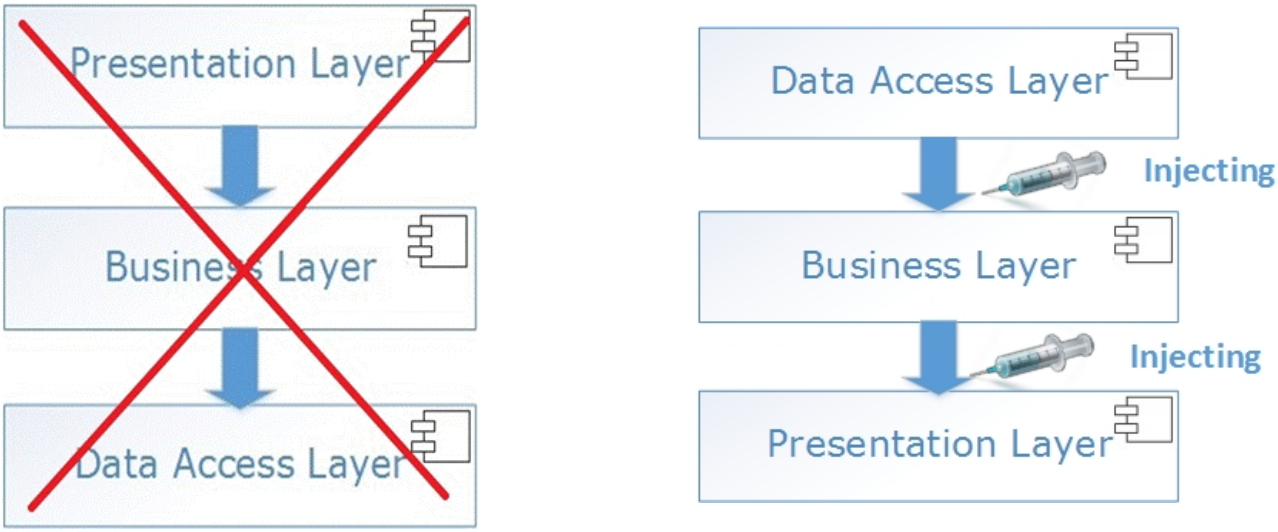

Imagen 1 y 2: Inteccón de dependencias.

El Control de inversión (**IoC**) es un principio de diseño genérico de la arquitectura de software que ayuda a crear marcos de software modulares y reutilizables que son fáciles de mantener. En palabras más digeridas digamos que el IoC se trata de separar lo que hay que hacer del cuando se hace, entregando el cómo se hace a un objeto externo.

El IoC se puede implementar en nuestro código con algunas técnicas conocidas como las siguientes:

- Callbacks
- Observer Pattern
- Factory Pattern
- Dependency Injection

Trataremos la implementación del principio general de control de inversión a través de la inyección de dependencias y utilizaremos un framework que está muy de moda hoy por hoy en el desarrollo de aplicaciones android llamado Dagger2.

Dagger

Dagger es un framework de **inyección de dependencias** totalmente estático y ejecutado en tiempo de compilación, tanto para Java como para Android. Es una adaptación de una versión anterior creada por la empresa Square y en sus versiones más recientes mantenida por Google.

Dagger pretende abordar muchos de los problemas de desarrollo y rendimiento que han plagado los sistemas basadas en la reflexión. Un sistema basado en la reflexión no es más que una analogía de la reflexión humana, en el sentido de que en los sistemas es una actividad computacional que razona sobre su propia computación.

Imagen 3. Logo Dagger.

Beneficios de utilizar el framework Dagger

El uso de dagger en nuestras aplicaciones android nos beneficia principalmente en el hecho de ordenar las dependencias, que están sean utilizadas en el orden adecuado y también, que no tengamos que repetir código de creación de instancias desde múltiples métodos o funciones, por ejemplo un caso sería el siguiente:

```
void onCreate(){
   Columnas columnas = **new** **Columnas**();
   Techo techo = **new** **Techo**();
   Ventanas ventanas = **new** **Ventanas**();
   Puertas puertas = **new** **Puertas**();
   Patio patio = **new** **Patio**();
   Habitaciones habitaciones = **new** **Habitaciones**();
   Baños baños = **new** **Ba**ños();

   Casa casa = **new** **Casa**(columnas, techo, ventanas, puertas, patio,
habitaciones, baños);
   casa.construye();
}
```

Del código anterior podemos reflexionar, que todas esas instancias son necesarias para el objeto Casa, pero si tenemos que crear una casa desde otra sección del código de nuestro sistema, tendremos que repetir dichas instancias. Con dagger quedaría de la siguiente manera:

```
void onCreate(){
      Columnas columnas = new Columnas();
      Techo techo = new Techo();
      Ventanas ventanas = new Ventanas();
      Puertas puertas = new Puertas();
      Patio patio = new Patio();
      Habitaciones habitaciones = new Habitaciones();
     Baños baños = new Baños();

     Casa casa = new Casa(columnas, techo, ventanas, puertas, patio,
habitaciones, baños);
     casa.construye();
}
```

Anotaciones Dagger

Dagger está basado en anotaciones, las cuales generan códigos que son muy fáciles de leer y entender. Las anotaciones de dagger son las siguientes:

- @**Module** y @**Provides**: definen clases y métodos que proporcionan dependencias.
- @**Inject**: solicitar dependencias. Se puede utilizar en un constructor, un campo o un método.
- @**Component**: habilita módulos seleccionados y se usa para realizar la inyección de dependencia.

Dagger 2 utiliza el código generado para acceder a los campos y no a la reflexión (principio de software). Por lo tanto, no está permitido utilizar campos privados para la inyección de estos.

Ejercicio 3:

Crear un proyecto implementando la librería Dagger2, que tenga los modelos de negocio casa, habitaciones, puertas y ventanas, donde al ejecutar la aplicación, se muestre en el LogCat un mensaje "Se está construyendo la casa", seguido de otro mensaje que indica "Son 4 el número de habitaciones", estos logs deben salir desde su modelo.

1. Crearemos a continuación un proyecto llamado `CasaDagger`, seleccionando blank activity, java y target sdk 28. Agregaremos la siguiente librería a nuestro archivo gradle, de la carpeta app en sus dependencias:

```
implementation 'com.google.dagger:dagger:2.16'
annotationProcessor 'com.google.dagger:dagger-compiler:2.16'
```

Implementaremos una estructura similiar a la reflejada en la imagen 4:

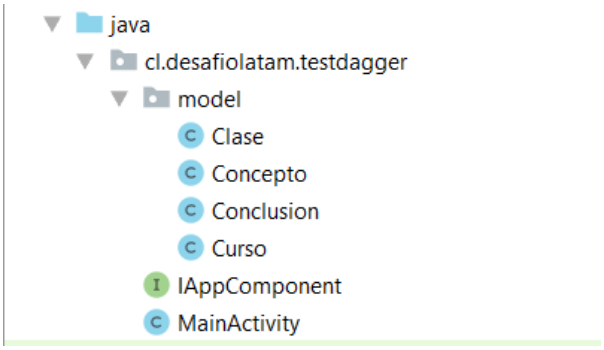

Imagen 4. Estructura de archivos.

2. Crearemos una serie de archivos de negocio para entender uno de los principales beneficios de Dagger para la inyección de dependencias:

- **Casa**

  Modelo de objeto Casa.

```
public class Casa {
   private static final String TAG = "Casa";
   private Puertas puertas;
   private Ventanas ventanas;
   private Habitaciones habitaciones;

   public Casa(Puertas puertas, Ventanas ventanas, Habitaciones habitaciones){
       this.puertas = puertas;
       this.ventanas = ventanas;
       this.habitaciones = habitaciones;
   }
   public void construir(){
       Log.d(TAG, "Estamos construyendo la casa");
       habitaciones.numeroHabitaciones();
   }
}
```

- **Habitaciones** Modelo de objetos habitaciones

```
import android.util.Log;

public class Habitaciones {

   private static final String TAG = "Casa";

   public Habitaciones(){
   }

   public void numeroHabitaciones(){
       Log.d(TAG, "Son 4 habitaciones");
   }
}
```

- **Puertas** Modelo de objetos de puertas.

```
public class Puertas {

   public Puertas(){

   }
}
```

- **Ventanas**: modelo de objetos de ventanas.

```
public class Ventanas {

public Ventanas(){

}
}
```

3. Agregamos los modelos de a la clase MainActivity para validar nuestros objetos de negocio creados:

```
public class MainActivity extends AppCompatActivity {

  private Casa casa;
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Puertas puertas = new Puertas();
    Ventanas ventanas = new Ventanas();
    Habitaciones habitaciones = new Habitaciones();

    casa = new Casa(puertas, ventanas, habitaciones);
    casa.construir();
  }
}
```

4. Al ejecutar nuestra app veremos el siguiente código en el Logcat:

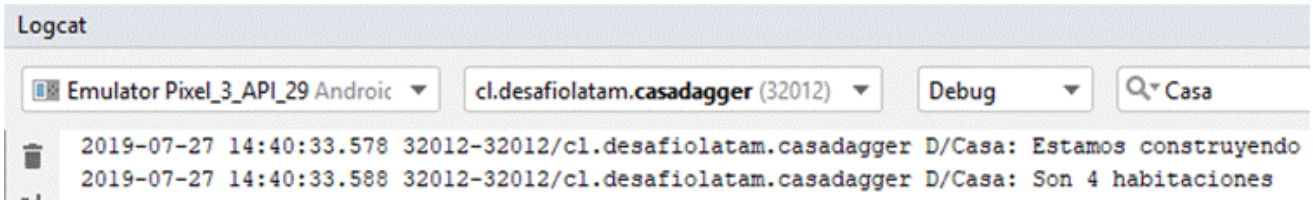

Imagen 5. Logcat del resultado a la ejecucuón.

5. Tenemos un modelo de clases integrado a la actividad y funcionando con sus mensajes. Es momento de utilizar dagger y comprender sus beneficios para este ejemplo. Comenzaremos por crear una interfaz general para nuestra app llamada AppComponent:

```
@Component
public interface AppComponent {
   Casa getCasa();
}
```

6. Este objeto getCasa tendrá la habilidad de ser accedido desde cualquier clase como un componente a futuro en el sistema, sin embargo para que funcione plenamente es necesario inyectar sus dependencias, para ello **vamos a cada uno de los constructores de nuestras clases de negocio (Casa, Puertas, Ventanas, Habitaciones) y agregaremos justo antes del constructor la etiqueta @Inject**, la cual permitirá que las dependencias sean utilizadas desde nuestra "MainActivity" sin necesidad de instanciarlas en su método onCreate.

```
public class Casa {

    private static final String TAG = "Casa";

    private Puertas puertas;
    private Ventanas ventanas;
    private Habitaciones habitaciones;

    @Inject
    public Casa(Puertas puertas, Ventanas ventanas, Habitaciones habitaciones){
        this.puertas = puertas;
        this.ventanas = ventanas;
        this.habitaciones = habitaciones;
    }

    public void construir(){
```

Imagen 6. Utilizando la etiqueta @Inject

7. Realizamos un `MakeBuild` de nuestra aplicación en este punto para que Dagger cree los objetos asociados a nuestras anotaciones.

8. Por último, en nuestra MainActivity reemplazamos el código dentro del método onCreate:

```
Puertas puertas = new Puertas();
Ventanas ventanas = new Ventanas();
Habitaciones habitaciones = new Habitaciones();

casa = new Casa(puertas, ventanas, habitaciones);

casa.construir();
```

Por el siguiente:

```
AppComponent component = DaggerAppComponent.create();
casa = component.getCasa();
casa.construir();
```

La clase DaggerAppComponent fue autogenerada por dagger, y de esta manera podemos utilizar este componente en otras secciones de nuestra aplicación además de evitar el denominado código boilerplate de creación de instancias.

Imaginemos todo lo que nos podríamos ahorrar de código cuando un sistema crece y se crean más y más objetos con esta librería de inyección de dependencias.

La clase autogenerada DaggerAppComponent contiene lo siguiente (Ctrl + mouse over):

```
public final class DaggerAppComponent implements AppComponent {
 private DaggerAppComponent(Builder builder) {}

 public static Builder builder() {
   return new Builder();
 }
 public static AppComponent create() {
   return new Builder().build();
 }
 @Override
 public Casa getCasa() {
   return new Casa(new Puertas(), new Ventanas(), new Habitaciones());
 }
 public static final class Builder {
   private Builder() {}

   public AppComponent build() {
     return new DaggerAppComponent(this);
   }}}
```

9. Ejecutamos nuestra aplicación nuevamente con los cambios realizados, y el resultado es el siguiente:

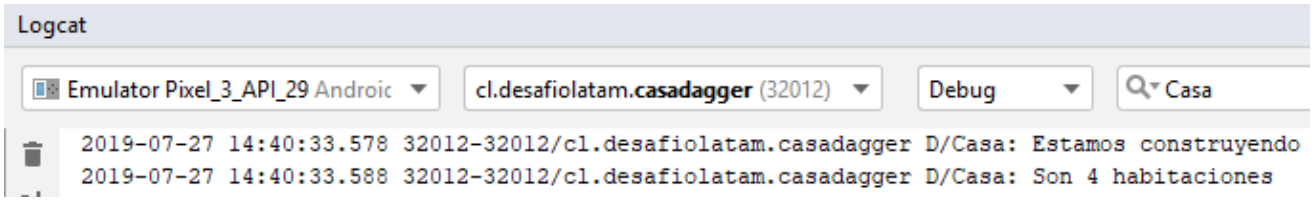

Imagen 7. Finalizando la implementación de Dagger

Hemos aprendido a implementar Dagger y a conocer algunos de sus principales usos y beneficios.

Pruebas de Software

Competencias:

- Aprender qué son los tests de software
- Conocer los beneficios de hacer tests sobre nuestras aplicaciones
- Aprender sobre los test unitarios y aplicarlos en un proyecto

Introducción

Las pruebas de software sigue siendo uno de los temas pendientes en el mundo del desarrollo de aplicaciones de software. Muchos programadores no son conscientes de que desarrollar sin pruebas resulta en acrobacias, en el trapecio sin red de seguridad, sino además una fuente de errores, malas prácticas y ansiedad.

Es importante construir pruebas de software que garanticen un código de calidad y una aplicación, cuyas etapas y posibles aristas, hayan podido ser comprobadas y consideradas, concluyendo en un sistema maduro y óptimo para la entrega.

Origen de las pruebas de software

El Testing de software nace aproximadamente en el año 1960 a partir de la crisis del desarrollo del software, cuando empiezan a desarrollar los primeros sistemas para el Departamento de Defensa de los Estados Unidos. A está época se le llamó así porque el software era muy complicado para elaborar, no se entregaba a tiempo, era muy costoso, y difícil identificar su avance porque no es un tangible. Por está situación se empiezan a generar múltiples soluciones a la crisis del software, como la Calidad de los procesos de desarrollo de software, el mejoramiento a la infraestructura del software, los frameworks, y por supuesto el Testing que nace como una respuesta para poder sobrellevar la llamada crisis del software, que hoy ya no es la crisis del software, es la industria natural del Software.

El desarrollo de software implica conocimiento, experiencia, talento, capacidad intelectual y un poco de magia, es una labor compleja. En la actualidad la rapidez con la que emergen nuevos lenguajes, frameworks, patrones, buenas prácticas, librerías, nuevos dispositivos, entre otros, hace del desarrollo de software una exigente carrera.

Hace décadas atrás, lo principal era la capacidad humana de memorización y abstracción; lo que hoy en día implica con toda esta tecnología, que los fallos y errores son inevitables si los intentamos evitar solo con nuestras capacidades humanas.

Testing

El Testing de Software es toda una disciplina en la ingeniería de software permite tener procesos, métodos de trabajo y herramientas para identificar defectos en el software alcanzando un proceso de estabilidad del mismo. El Testing no es una actividad que se piensa al final del desarrollo del software, va paralelo a este. Permite que lo que se está construyendo, se realice de manera correcta de acuerdo a lo que necesita un usuario final.

La importancia del testing es que fundamentalmente es una forma de prevenir o inclusive de corregir posibles desviaciones del software antes de que sea operable. Se tenía la equivocada idea que el testing se realizaba al final, cuando ya el software estaba codificado y justo antes de entregarlo a la operación, pero actualmente el testing de software debe ir desde el inicio del proceso.

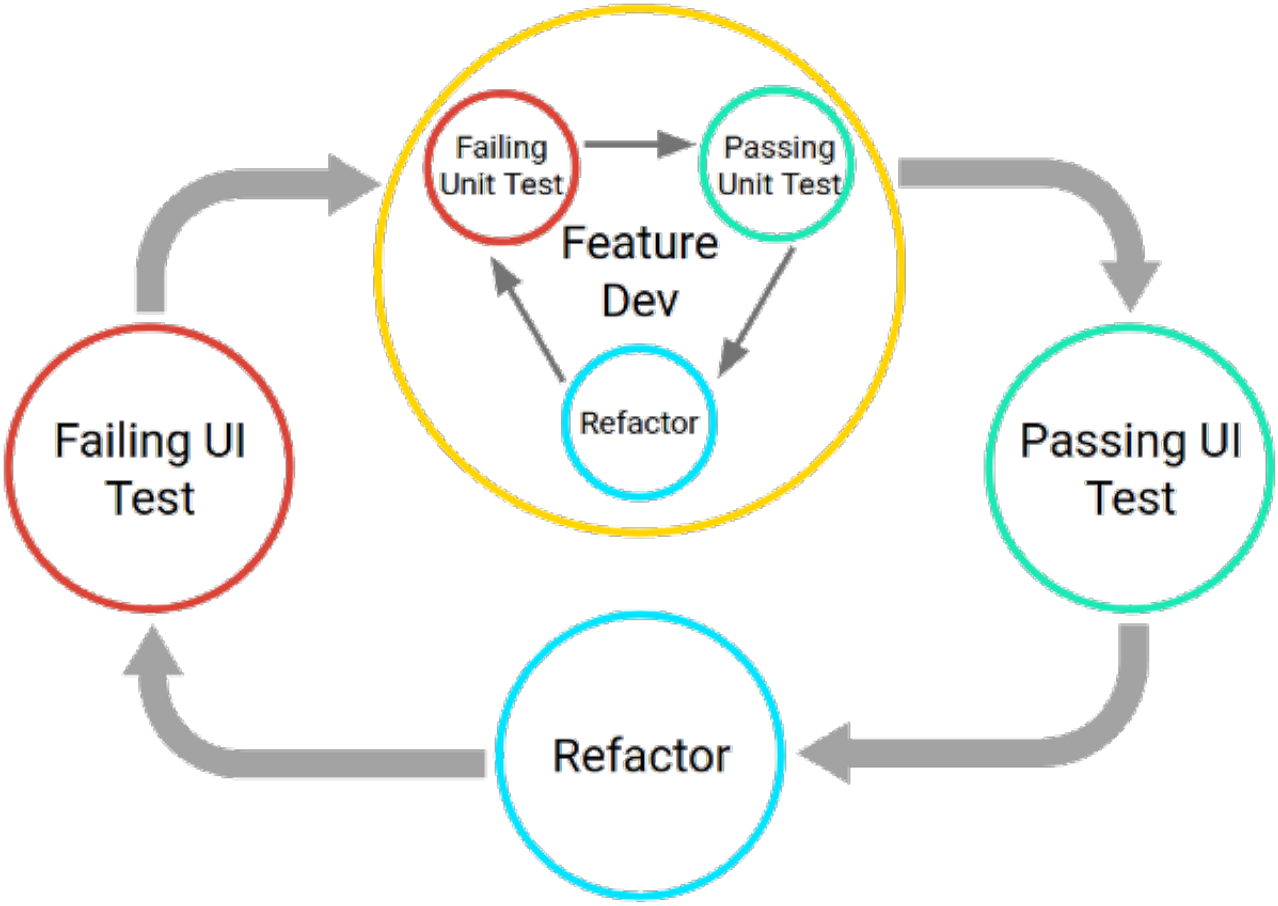

Imagen 8. Diagrama de creación y pruebas de código

Ciclo tipo de pruebas de software

Aunque existen variaciones entre las organizaciones, existe un ciclo típico de prueba. El siguiente ejemplo es común entre las organizaciones, estas prácticas se encuentran comúnmente en otros modelos de desarrollo, pero pueden no ser tan claras o explícitas.

- **Análisis de requisitos**: las pruebas deben comenzar en la fase de requisitos del ciclo de vida del desarrollo del software. Durante la fase de diseño, los evaluadores trabajan para determinar qué aspectos de un diseño se pueden probar y con qué parámetros funcionan esas pruebas.
- **Planificación de prueba**: estrategia de prueba, plan de prueba, creación de banco de pruebas. Dado que muchas actividades se llevarán a cabo durante las pruebas, se necesita un plan.
- **Desarrollo de pruebas**: procedimientos de prueba, escenarios de prueba, casos de prueba, conjuntos de datos de prueba, scripts de prueba para usar en el software de prueba.
- **Ejecución de la prueba**: los evaluadores ejecutan el software basándose en los planes y documentos de prueba y luego informan cualquier error encontrado al equipo de desarrollo. Esta parte puede ser compleja cuando se ejecutan pruebas con una falta de conocimientos de programación.
- **Informe de prueba**: una vez que se completa la prueba, los evaluadores generan métricas y hacen informes finales sobre su esfuerzo de prueba y si el software probado está listo para su lanzamiento.
- **Análisis del resultado de la prueba**: o el Análisis de defectos, lo realiza el equipo de desarrollo, generalmente junto con el cliente, para decidir qué defectos deben asignarse, repararse, rechazarse (es decir, encontrar que el software funciona correctamente) o diferirse para su posterior tratamiento.
- **Reprobación de defectos**: una vez que el equipo de desarrollo ha resuelto un defecto, el equipo de pruebas lo vuelve a probar.
- **Pruebas de regresión**: es común tener un pequeño programa de pruebas construido con un subconjunto de pruebas, para cada integración de software nuevo, modificado o fijo, con el fin de garantizar que la última entrega no haya arruinado nada y que el producto de software como un todo sigue funcionando correctamente.
- **Cierre de la prueba**: Una vez que la prueba cumple con los criterios de salida, las actividades como la captura de los productos clave, las lecciones aprendidas, los resultados, los registros, los documentos relacionados con el proyecto se archivan y se utilizan como referencia para proyectos futuros.

Beneficios de realizar pruebas de software

- **Confianza**: las pruebas de software proporcionaron la confianza en el desarrollo del producto y brindaron la garantía de la calidad. El control de calidad realizado durante las pruebas de software proporcionó la confianza al negocio en cuanto a la implementación de los requisitos comerciales y de los usuarios.
- **Evidencia**: Las pruebas de software proporcionaron información sobre las lagunas en el software y su impacto en los usuarios.
- **Confiabilidad**: durante el transcurso de las pruebas de software, la calidad de un producto se mide en función de varios parámetros, como la funcionalidad, el rendimiento, la seguridad, la usabilidad y la aceptación. La verificación de todos los parámetros con respecto a los resultados de un producto en comentarios positivos hace que un producto sea más confiable.
- **Seguridad**: las pruebas de software también funcionan con los parámetros de seguridad del producto, que tiene la máxima prioridad recientemente para evitar todo tipo de vulnerabilidades y puertas traseras para los piratas informáticos que pueden conducir a la explotación del código fuente.

Test Unitarios

En programación, una prueba unitaria es una forma de comprobar el correcto funcionamiento de una unidad de código. Por ejemplo *en* diseño estructurado *o en diseño funcional* una función o un procedimiento, en **diseño orientado a objetos** una clase.

Esto sirve para asegurar que cada unidad funcione correctamente y eficientemente por separado. Además de verificar que el código hace lo que tiene que hacer, verificamos que sea correcto el nombre, los nombres y tipos de los parámetros, el tipo de dato que se devuelve, si el estado inicial es válido, el estado final es válido, etc… La idea es escribir casos de prueba para cada función no trivial o método en el módulo, de forma que cada caso sea independiente del resto. Luego, con las Pruebas de Integración), se podrá asegurar el correcto funcionamiento del sistema o subsistema en cuestión.

Para que una prueba unitaria tenga la calidad suficiente se deben cumplir los siguientes requisitos:

- **Automatizable**: No debería requerirse una intervención manual. Esto es especialmente útil para[integración continua
- **Completas**: Deben cubrir la mayor cantidad de código.
- **Repetibles o Reutilizables**: No se deben crear pruebas que sólo puedan ser ejecutadas una sola vez. También es útil para[integración continua
- **Independientes**: La ejecución de una prueba no debe afectar a la ejecución de otra.
- **Profesionales**: Las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.

Aunque estos requisitos no tienen que ser cumplidos al pie de la letra, se recomienda seguirlos o de lo contrario las pruebas pierden parte de su función.

Test Unitario en Android

Android Studio está diseñada para simplificar las pruebas. Con solo algunos clics, puedes establecer una prueba JUnit que se ejecute en el JVM local o una prueba instrumental que se ejecute en un dispositivo. Por supuesto, también puedes extender tus capacidades de pruebas integrando frameworks de prueba como Mockito, para probar llamadas de Android API en tus pruebas de unidades locales.

Generalmente al crear un nuevo proyecto en Android Studio, se crea automáticamente una carpeta sobre la raíz del proyecto de tipo *module-name*/src/test/java/.

Estas son pruebas que se ejecutan en la máquina virtual Java (JVM) local de tu máquina. Usa estas pruebas para minimizar el tiempo de ejecución cuando tus pruebas no tengan dependencias de marco de Android o cuando puedas simular las dependencias del marco de Android.

En el tiempo de ejecución, estas pruebas se ejecutan en una versión modificada de android.jar en la que se quitan todos los modificadores final. Esto te permite usar bibliotecas de simulación populares, como Mockito.

Ejercicio 4

Ejecutar pruebas unitarias sobre el proyecto “IndicadoresEconomicosChile” del capítulo “Clean Architecture” de esta unidad.

1. Agregamos las librerías mockito y las librerías de soporte para anotaciones de android a nuestro archivo gradle de app:

```
testImplementation 'org.mockito:mockito-core:1.10.19'
androidTestImplementation 'com.android.support:support-annotations:28.0.0'
androidTestImplementation 'com.android.support.test:rules:1.0.2'
androidTestImplementation 'com.android.support.test:runner:1.0.2'
```

2. Nos ubicamos en la carpeta raíz de las pruebas unitarias y creamos una nueva clase llamada “GetIndicadorEconomicoTest”.

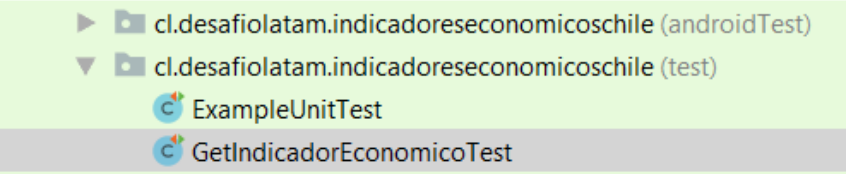

Imagen 9. Creando clase para GetIndicatorEconomicoTest

3. Agregamos una anotación sobre el nombre de la clase para indicar que este test correrá con Mockito Junit de la siguiente forma:

```
@RunWith(MockitoJUnitRunner.class)
public class GetIndicadorEconomicoTest {
```

Imagen 10. Agregando la notación @RunWith

4. Agregamos variables que vamos a requerir para las pruebas, partiendo de la inicialización de nuestra clase GetIndicadoresEconomicosUseCase, la cual será la base de esta prueba:

```
private GetIndicadorEconomicoUseCase getIndicadorEconomicoUseCase;
private String[] params = new String[2];
private static final String tipoIndicador = "dolar";
private static final String fechaIndicador = "18-07-2019";
```

5. Agregamos una variable de la clase que requiere GetIndicadorEconomicoUseCase en su constructor, indicando una anotación @Mock , con la finalidad de que sea inyectada la dependencia en MockitoJunit para la prueba, de la misma forma como lo hace dagger.

```
@Mock
private IIndicadorEconomicoRepository mIIndicadorEconomicoRepository;
```

6. Agregamos un método llamado “setUp” con la etiqueta @Before, que indica será procesado antes de la prueba. Este método tiene la finalidad de inicializar la clase GetIndicadorEconomicoUseCase.

```
@Before
public void setUp() {
   getIndicadorEconomicoUseCase = new
GetIndicadorEconomicoUseCase(mIIndicadorEconomicoRepository);
}
```

7. Finalmente agregamos el método de prueba llamado “testGetIndicadorEconomico” con la anotación @Test de la siguiente manera:

```
@Test
public void testGetIndicadorEconomico() {
   params[0] = tipoIndicador;
   params[1] = fechaIndicador;
   getIndicadorEconomicoUseCase.buildUseCaseObservable(params);

Mockito.verify(mIIndicadorEconomicoRepository).getIndicadorEconomicoFromApiLaye
r(tipoIndicador, fechaIndicador);
   Mockito.verifyNoMoreInteractions(mIIndicadorEconomicoRepository);

   getIndicadorEconomicoUseCase.dispose();
}
```

En el método anterior estamos verificando que funcione el método de la interfaz IIndicadorEconomicoRepository a través de su flujo en el código con la clase GetIndicadorEconomicoUseCase y finalizamos con la verificación de que no hayan más llamados a dichos métodos. El resultado debería ser el siguiente:

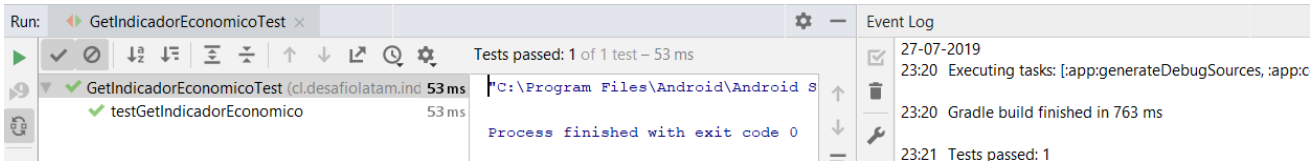

Imagen 11. Resultado final.

Pruebas con Espresso

Competencias:

- Conocer los tests instrumentales en android
- Diferenciar los distintos tipos de test en android
- Aplicar tests instrumentales en un proyecto utilizando Espresso
- Conocer las alternativas más utilizadas para hacer tests instrumentales en android

Introducción

Con la era de la digitalización, las pruebas instrumentadas de software se han hecho más populares debido a la gran cantidad de dispositivos, tamaños, resoluciones y sistemas operativos.

La variedad de plataformas donde pueden operar nuestros sistemas han hecho de las pruebas instrumentales una gran necesidad. Aprendiendo a realizar las pruebas instrumentales tendrás una perspectiva más amplia en tu desarrollo y comprenderás en mayor proporción lo que el usuario espera y desea de la aplicación que desarrollamos, por ende es un valor agregado a tu carrera android.

Test Instrumental en Android

Se usan para automatizar la interacción de usuarios. Generalmente al crear un proyecto nuevo en android studio, el IDE crea automáticamente una carpeta para pruebas instrumentales desde la raíz del proyecto en module-name/src/androidTest/java/.

Son pruebas que se ejecutan en un dispositivo o emulador de hardware. Estas pruebas tienen acceso a las API Instrumentation, y te permiten acceder a información como el Context de la app que pruebas y controlar la app a prueba desde tu código de prueba.

Debido a que las pruebas instrumentadas se compilan en un APK (por separado del APK de tu app), deben tener su propio archivo AndroidManifest.xml. Sin embargo, Gradle automáticamente genera este archivo durante la compilación para que no se vea en el conjunto de fuentes de tu proyecto. Puedes agregar tu propio archivo de manifiesto si es necesario, por ejemplo, a fin de especificar un valor diferente para `minSdkVersion` o registrar receptores de ejecución solo para tus pruebas. Cuando se compila tu app, Gradle combina varios archivos de manifiesto en un manifiesto.

Diferencia entre test instrumental y unitario

A menudo, las **pruebas unitarias** se conocen como "pruebas locales" o "pruebas unitarias locales". La principal razón para esto parece ser que desea poder ejecutar pruebas **sin un dispositivo o un emulador conectado**.

Las pruebas unitarias no pueden probar la interfaz de usuario de su aplicación sin omitir objetos, como una actividad.

Las **pruebas de instrumentación se ejecutan en un dispositivo o un emulador**. En segundo plano, su aplicación se instalará y luego se instalará una aplicación de prueba que controlará su aplicación y ejecutará las pruebas de UI según sea necesario. Las pruebas de instrumentación también se pueden usar para probar la lógica de IU. Son especialmente útiles cuando necesita probar código que depende del contexto.

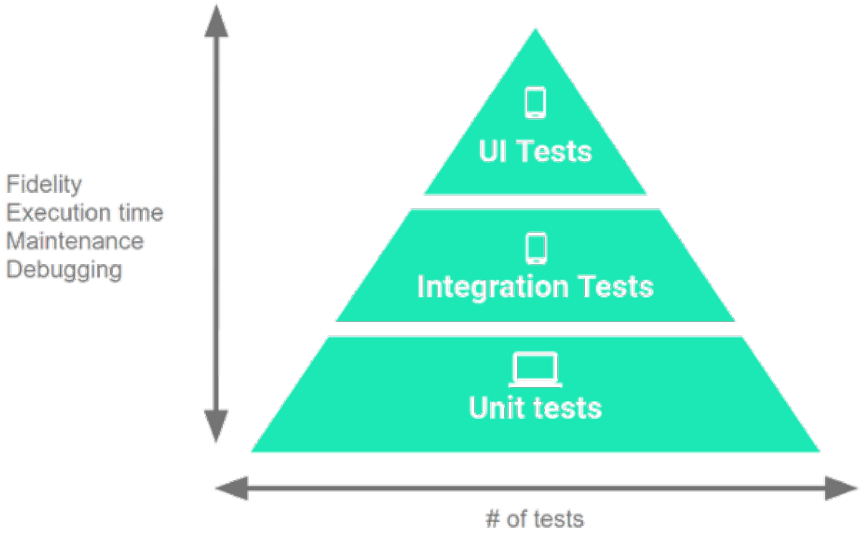

Imagen 12. Diagrama de fidelidad al realizar test.

Espresso

Espresso es un marco de interfaz de usuario de Android de Google. Es una herramienta liviana dirigida a los desarrolladores de aplicaciones, lo que significa que puede usarse en todo su potencial en el desarrollador de pruebas.

Para desarrollar estas pruebas es ideal estar familiarizado con el código de la aplicación y tener acceso a él. Las pruebas de Espresso son muy confiables y se ejecutan rápido. Espresso tiene acceso a las partes internas de los objetos de la interfaz de usuario.

Su punto negativo es que espresso solo puede probar una aplicación a la vez y no tiene acceso a los recursos del dispositivo; sin embargo, esto podría resolverse fácilmente creando pruebas combinadas con UI Automator.

Ejercicio 5

Implementar pruebas instrumentales sobre la aplicación “IndicadoresEconomicosChile” del capítulo “Clean Architecture” de la presente unidad.

1. Nos posicionamos sobre el menú principal superior, escogemos “Run” seguido de “Record Espresso Test”.

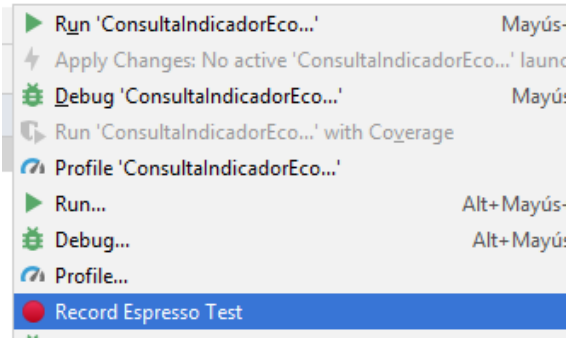

Imagen 12. Incorporando pruebas instrumentales.

En este punto iniciará la aplicación en nuestro emulador, junto con una ventana adicional que grabará todos los pasos que hagamos en el sistema y los guardará.

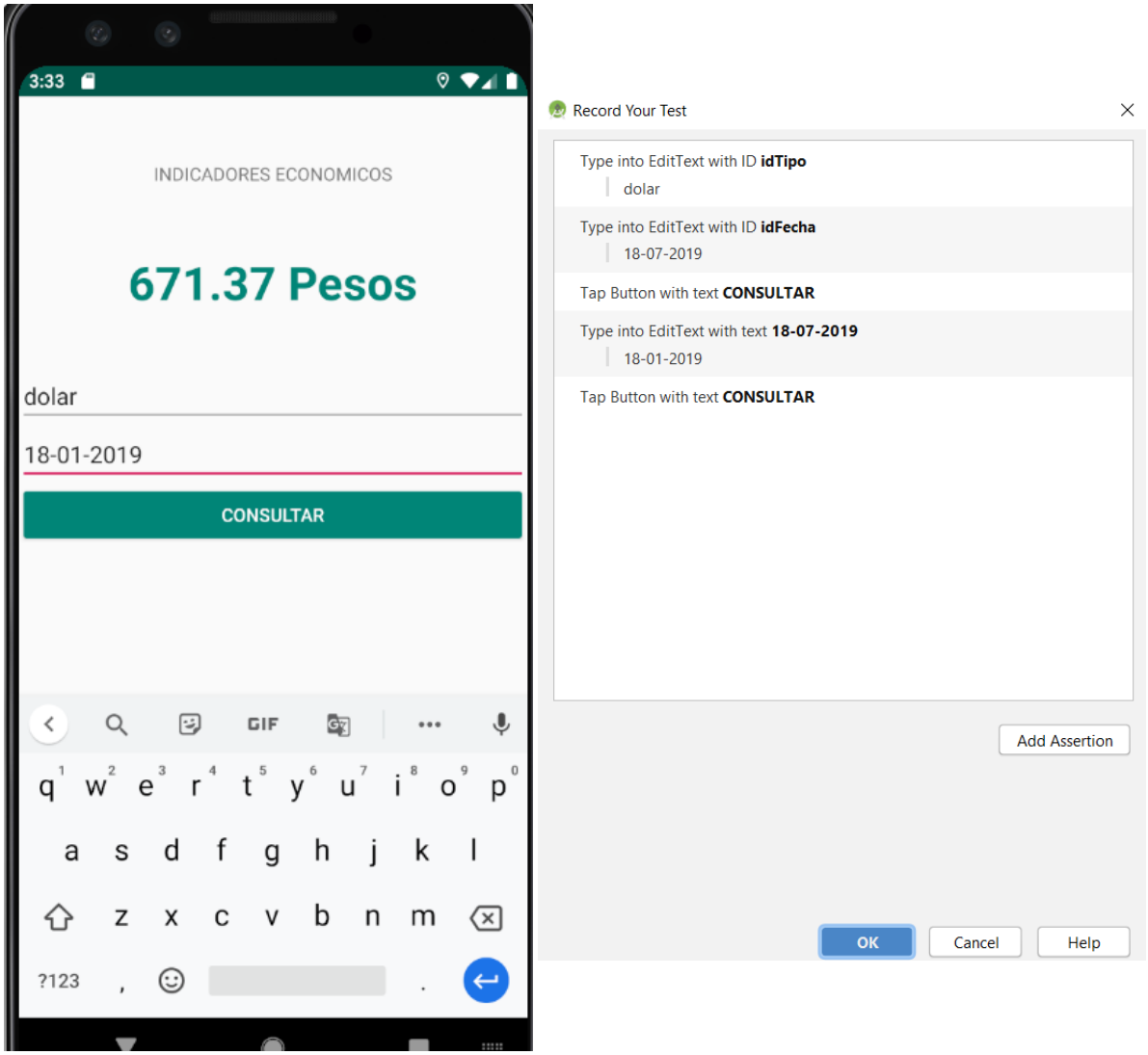

Imagen 13 y 14. Guardando los pasos ejecutados en el emulador.

De lo anterior, hemos realizado dos acciones de consulta con distintas fechas sobre la aplicación que han quedado registradas por Espresso.

3. Seleccionamos “Ok” y a continuación se nos solicitará un nombre de clase para el código que será auto generado.

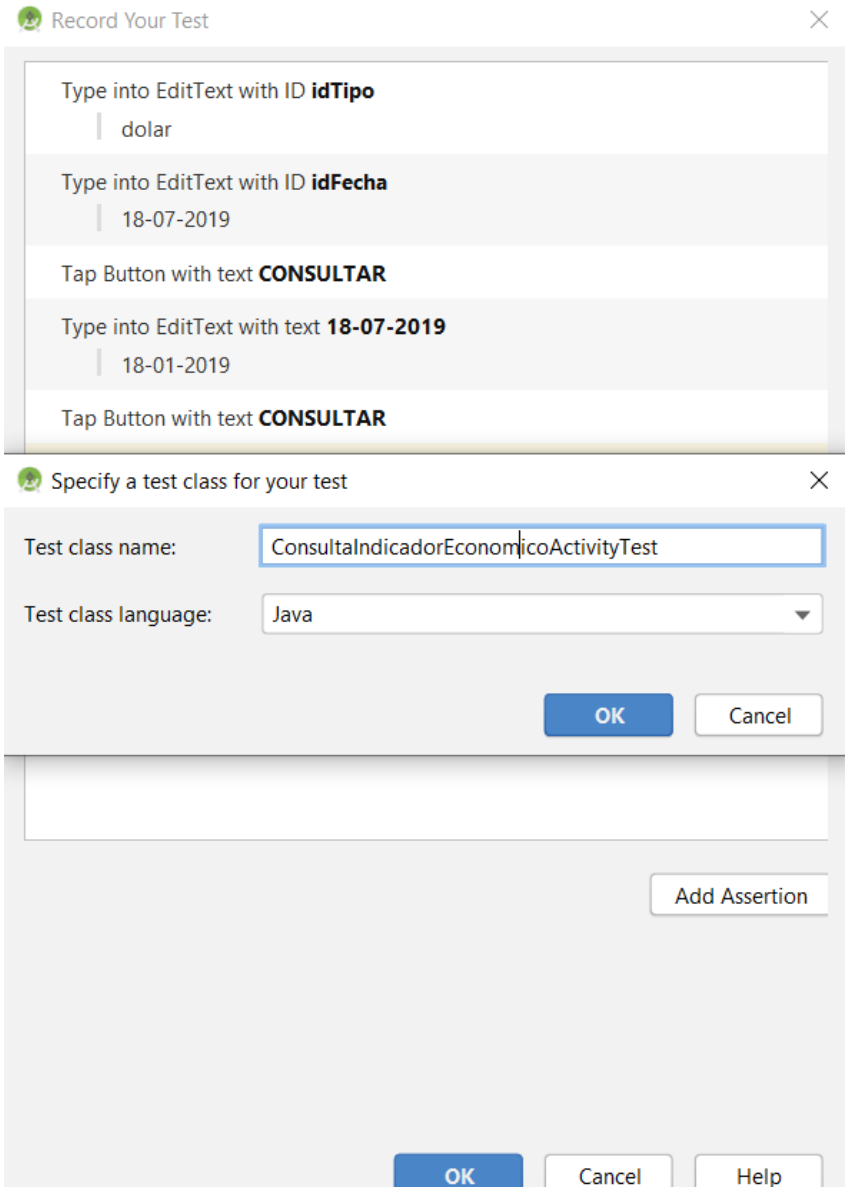

Imagen 15. Autogenerando código para el test.

4. Android y Espresso han creado una clase de pruebas a partir de “Record Espresso” con los pasos que hemos realizado. Este es el código generado:

```
@LargeTest
@RunWith(AndroidJUnit4.class)
public class ConsultaIndicadorEconomicoActivityTest {

   @Rule
   public ActivityTestRule<ConsultaIndicadorEconomicoActivity>
mActivityTestRule = new ActivityTestRule<>
(ConsultaIndicadorEconomicoActivity.class);

   @Test
   public void consultaIndicadorEconomicoActivityTest() {
```

```
        ViewInteraction editText = onView(
                allOf(withId(R.id.idTipo),
                        childAtPosition(
                                childAtPosition(
                                        withId(android.R.id.content),
                                        0),
                                1),
                        isDisplayed()));
        editText.perform(replaceText("dolar"), closeSoftKeyboard());

        ViewInteraction editText2 = onView(
                allOf(withId(R.id.idFecha),
                        childAtPosition(
                                childAtPosition(
                                        withId(android.R.id.content),
                                        0),
                                2),
                        isDisplayed()));
        editText2.perform(replaceText("18-07-2019"), closeSoftKeyboard());

        ViewInteraction button = onView(
                allOf(withId(R.id.btnAction), withText("CONSULTAR"),
                        childAtPosition(
                                childAtPosition(
                                        withId(android.R.id.content),
                                        0),
                                3),
                        isDisplayed()));
        button.perform(click());

        ViewInteraction editText3 = onView(
                allOf(withId(R.id.idFecha), withText("18-07-2019"),
                        childAtPosition(
                                childAtPosition(
                                        withId(android.R.id.content),
                                        0),
                                2),
                        isDisplayed()));
        editText3.perform(replaceText("18-01-2019"));

        ViewInteraction editText4 = onView(
                allOf(withId(R.id.idFecha), withText("18-01-2019"),
                        childAtPosition(
                                childAtPosition(
                                        withId(android.R.id.content),
                                        0),
```

```
                                2),
                        isDisplayed()));
        editText4.perform(closeSoftKeyboard());

        ViewInteraction button2 = onView(
                allOf(withId(R.id.btnAction), withText("CONSULTAR"),
                        childAtPosition(
                                childAtPosition(
                                        withId(android.R.id.content),
                                        0),
                                3),
                        isDisplayed()));
        button2.perform(click());
    }

    private static Matcher<View> childAtPosition(
            final Matcher<View> parentMatcher, final int position) {

        return new TypeSafeMatcher<View>() {
            @Override
            public void describeTo(Description description) {
                description.appendText("Child at position " + position + " in
parent ");
                parentMatcher.describeTo(description);
            }

            @Override
            public boolean matchesSafely(View view) {
                ViewParent parent = view.getParent();
                return parent instanceof ViewGroup &&
parentMatcher.matches(parent)
                        && view.equals(((ViewGroup)
parent).getChildAt(position));
            }
        };
    }
}
```

5. Al ejecutar nuestro test instrumental desde esta clase deberíamos visualizar la aplicación iniciada y recibiendo inputs automáticamente, de la misma manera como lo hicimos anteriormente. El resultado de la prueba debería ser el siguiente:

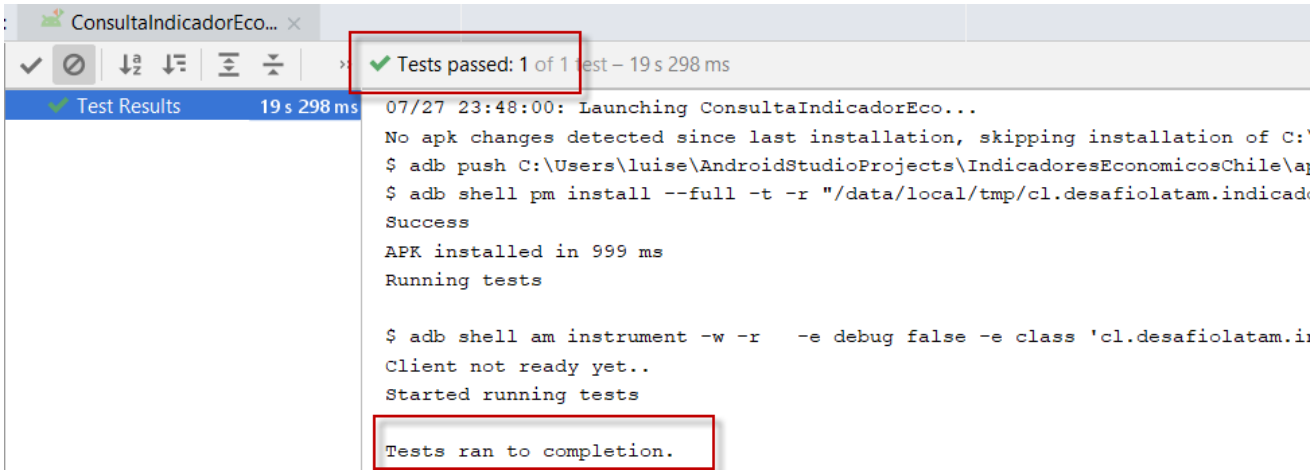

Imagen 15. Aprobando el test.

Alternativas al test instrumental con Espresso

UI Automator

UI Automator es un marco de UI de Android para pruebas móviles desarrollado y mantenido por Google. Sus características clave incluyen pruebas funcionales entre aplicaciones, la capacidad de probar múltiples aplicaciones y cambiar entre las aplicaciones instaladas y las del sistema.

UI Automator es una herramienta de prueba de caja negra (es decir, un desarrollador de pruebas no necesita conocer la estructura interna de la aplicación y puede confiar completamente en los elementos visibles de la interfaz de usuario). No es sorprendente que las pruebas de UI Automator estén escritas en Java, el idioma preferido de Google. UI Automator consta de dos conjuntos de API: UI Automator API, para manipular los componentes de UI de una aplicación, y las API de estado del dispositivo, para acceder y realizar operaciones en el dispositivo (cambiar la rotación del dispositivo, presionar el botón Atrás, Inicio o Menú, etc). También viene con un Visor de UI Automator muy útil, una herramienta GUI para escanear y analizar los componentes de UI que se muestran actualmente en el dispositivo.

La desventaja de UI Automator es que no es compatible con WebView, sobre la cual se construyen las aplicaciones híbridas de Android. Por lo tanto, UI Automator sólo admite aplicaciones nativas de Android.

Robotium:

Robotium es un marco de interfaz de usuario de Android de código abierto. Ha existido desde 2010, y ahora es una herramienta muy madura y estable. La última versión, 5.6.3, se lanzó en septiembre de 2016. En las últimas versiones, la legibilidad de las pruebas y la velocidad de ejecución de las pruebas se han mejorado significativamente.

La mayoría de los blogs técnicos, tutoriales y cursos se refieren al tipo de pruebas utilizadas por Robotium como pruebas de caja negra. Yo diría que es realmente una prueba de caja gris, porque para escribir pruebas de Robotium, necesita conocer la estructura interna de la aplicación, al menos parcialmente (nombres de actividades, por ejemplo).

Robotium es una extensión del marco de prueba de Android y, como tal, utiliza el enlace en tiempo de ejecución a los componentes de la interfaz de usuario, lo que hace que las pruebas sean más sólidas. Robotium viene como un archivo jar que debe compilarse con su proyecto.

Esta es una lista parcial de las principales características de Robotium:

- Soporta aplicaciones tanto nativas como híbridas.
- Puede ejecutar pruebas tanto en dispositivos reales como en emuladores.
- Admite parafernalia completa de la interfaz de usuario de android, actividades, botones, menús, brindis, diálogos, etc.
- Apoya gestos.
- Tiene algunas funciones de control del dispositivo: cambiar la orientación del dispositivo, tomar capturas de pantalla, desbloquear la pantalla, etc.
- El grabador Robotium está disponible como un complemento de Android Studio y Eclipse de pago. Es una gran herramienta para ponerse en marcha con las pruebas rápidamente.
- Se puede ejecutar como parte de la integración continua.
- El idioma elegido por Robotium es Java.

Appium

Appium es una herramienta de prueba móvil de código abierto que es compatible con iOS y Android. Se puede utilizar para probar cualquier tipo de aplicación móvil: nativa, web e híbrida. Appium es una herramienta multiplataforma, lo que significa que puede ejecutar la misma prueba en diferentes plataformas. Para lograr esta capacidad multiplataforma, Appium utiliza el marco de interfaz de usuario de Android proporcionado por el proveedor para las pruebas: XCTest para iOS y UIAutomator o Instrumentation para Android. Envuelve estos marcos de proveedores en Selenium WebDriver. Esto permite a los desarrolladores de Appium escribir pruebas en una amplia variedad de lenguajes de programación: Java, Objective-C, JavaScript, PHP, Ruby, Python, etc. También hace que la prueba de Appium sea muy similar a la de las pruebas de Selenium.

Como WebDriver se creó inicialmente para pruebas web, Appium extendió WebDriver con métodos API adicionales para hacerlo más adecuado para la automatización móvil.

Aquí hay algunas características adicionales útiles de Appium:

- No necesitas instalar nada en el dispositivo.
- No necesitas recompilar o cambiar tu aplicación de ninguna manera para probarla con Appium.
- Appium tiene una comunidad muy grande y activa.
- Appium viene con una herramienta para escanear y analizar los componentes de la interfaz de usuario de una aplicación: Appium UI Inspector. Los desarrolladores también pueden usar UI Automator Viewer de Android Studio.

Si necesitas escribir pruebas tanto para iOS como para Android, y eres un fanático de Selenium, Appium sería una excelente opción para ti.