

Estudio de testeabilidad y realización de tests

Práctica 2 de Calidad del Software

grep

Equipo: Carlos Solsona Álvarez,
Stefano Tomasini Hoefner,
Anna Trofimova,
Sergio Villagarcía Sánchez,
Andreas Wolf Wolf,
Shuheng Ye,
Marcos Ferrer Zalve

T01: Informe sobre los tests en la aplicación	3
T02: Test plan de la aplicación	5
2.1 Introducción	5
2.2 Objetivos	5
2.3 Tareas	5
2.4 Alcance y propósito de los tests	5
2.5 Estrategia para realizar los tests	5
2.6 Tests unitarios	5
2.7 Tests de integración	5
2.8 Requerimientos de Hardware	5
2.9 Requerimientos pre-testing	5
2.10 Programa de tests	5
2.11 Partes que se van a testear	5
2.12 Partes que no se van a testear	5
2.13 Roles	5
2.14 Dependencias	5
2.15 Aprobación del test plan	5
T03: Realizar TDD (Test Driven Development)	6
T04: Realizar test unitario en la aplicación	6
4.1 - Test unitario de clase AdaptadorEventos	6
4.2 - Test unitario de clase AdaptadorMensajes	10
T05: Realizar test de integración en la aplicación	11
5.1 Test de la base de datos	11
5.2 - Test de la clase SignUp	12
T06: Realizar tests de interfaz en la aplicación	13
T07: Automatización de la ejecución de los tests con CircleCI	16

T01: Informe sobre los tests en la aplicación

En esta sección de la memoria vamos a explicar brevemente qué clases partes se pueden testear y qué partes no se pueden testear.

Comenzamos con las partes que sí se pueden testear y que vamos a testear en esta práctica:

- La primera parte a testear, ya sea mediante tests unitarios, integración o de interfaz sería la parte de iniciar sesión y todo lo que sea crear usuarios.
- La siguiente parte a testear sería el propio calendario que es lo que hace la aplicación.

Continuamos con partes que no se pueden testear o que sus tests realmente sirven de poco para la aplicación:

- Las partes de nuestra aplicación que no servirían mucho para hacer tests son el chat/foro (que no tiene mucho sentido tener un chat entre usuarios para una aplicación de calendario) y algunas funcionalidades del calendario, como por ejemplo guardar una actividad nueva en la base de datos, porque no está implementado en la aplicación cuando se nos presentó por primera vez. Aunque si podemos hacer tests unitarios en el calendario que no persistan, como añadir algo a la base de datos, comprobar y comprobar que se ha añadido correctamente aunque luego no se guarde.

Por último, vamos a mostrar los posibles tests a automatizar de nuestra aplicación. Cuando hablamos de automatizar tests, lo primero que se debe hacer es automatizar los tests unitarios, que en general son los más fáciles de implementar, y los tests de integración. Nuestra aplicación no es del todo funcional, ya que se nos presentó una aplicación calendario. Podemos iniciar sesión, registrarse, hablar por un chat, crear una actividad en el calendario (que no funciona porque no persiste) y cerrar sesión. Realmente para lo que supuestamente es y lo que realmente hace pocas cosas se pueden probar por lo que realmente lo poco que se podría automatizar sería el inicio de sesión y registro, ya que guardar una actividad en el calendario no serviría de nada automatizarlo ya que no funciona y automatizar un chat tampoco es demasiado útil.

A continuación vamos a presentar nuestro plan de testing modificado para nuestra aplicación en concreto, ya que debido a la inoperancia de la aplicación en general, realizar todos los tipos de tests deseados va a resultar imposible.

T02: Test plan de la aplicación

2.1 Introducción

El propósito de esta sección es hacer un plan de testing para nuestra aplicación. La aplicación a testear simula un calendario en el que podemos guardar actividades y hablar con otros usuarios mediante un foro. La aplicación no va mucho más allá de esto y no tiene más funcionalidades complejas como estas.

2.2 Objetivos

El objetivo principal es asegurar el correcto funcionamiento de las pocas funcionalidades que tiene esta aplicación. Mediante tests unitarios, de integración e interfaz vamos a probar una a una las funcionalidades de la aplicación para analizar cómo funciona y si la aplicación ejecuta bien todas las clases y devuelve lo que tiene que devolver.

2.3 Tareas

Como ya se ha comentado en el punto anterior, las tareas principales del plan del test es designar e implementar los tests necesarios para asegurar el correcto funcionamiento. Se va a testear la interfaz, probando las pantallas una a una realizando varias tareas para ver si la interfaz cambia según vayamos realizando cambios en la aplicación. También vamos a probar alguna funcionalidad de la base de datos (lo que nos deje porque la base de datos no funciona correctamente), y por último el propio calendario, sus actividades y el foro general.

2.4 Estrategia para realizar los tests

La estrategia que vamos a utilizar para realizar los tests es bastante sencilla, vamos a ir clase por clase apuntando las tareas más importantes y de alto nivel de la aplicación para seguidamente implementar esas tareas en forma de tests

2.5 Tests unitarios

Para los tests unitarios, ya que son de los más “fácil” de implementar, los usaremos para probar las funcionalidades más cortas y rápidas de la aplicación, como mandar un mensaje en el foro o guardar una actividad en el calendario

2.5 Tests de integración

Para los tests de integración, probaremos que la base de datos funciona correctamente y guarda todos los objetos en ella para que persistan después de cada ejecución.

2.7 Requerimientos de hardware

En cuanto a hardware, cualquier portátil que corra Android Studio en sus versiones más recientes será más que suficiente para realizar los tests y ejecutar la aplicación sin ningún problema de recursos.

2.8 Partes que se van a testear

Como ya se ha mencionado en puntos anteriores y en la introducción del plan de testing, las clases MainActivity, MenuActivity, SignUpActivity y CalendarActivity son las que se van a testear. Es decir, todo lo relacionado con iniciar sesión, registrar un usuario, el calendario en general y el foro/chat.

2.9 Partes que no se van a testear

Las partes que no se van a testear son la parte del admin, ya que la aplicación no tiene en el código fuente nada sobre la parte del admin. Al presentarnos la aplicación, encontramos varias clases y métodos vacíos que nos dieron a entender que había un modo Admin, pero sin implementar. Además, tampoco vamos a testear nada que sea persistir cualquier objeto en la base de datos, ya que la que venía con la aplicación no funciona y crear una nueva base de datos de Firebase llevaría demasiado tiempo y supondría cambiar más de la mitad del código.

2.10 Roles

En cuanto a los roles en el test plan, los 7 integrantes del grupo nos vamos a encargar de testear varias partes del código. Todos hemos trabajado por igual y hemos realizado varios tests, los cuales se han ejecutado todos con éxito. Nos hemos dividido de la siguiente manera:

Carlos y Shu se han encargado de los tests unitarios de la aplicación.

Anna y Marcos de los tests de integración y base de datos

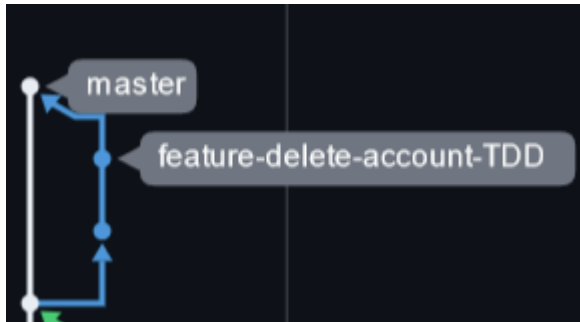
Sergio y Stefano de los tests y funcionalidad TDD

Andreas de los tests de interfaz

2.11 Aprobación del test plan

Para finalizar el test plan, los 7 integrantes del grupo, hemos leído el plan de testing, estamos de acuerdo con él y vamos a seguir el plan al pie de la letra para implementar correctamente los tests de la aplicación y asegurar que la aplicación funciona correctamente, es fácil de usar, accesible y completa

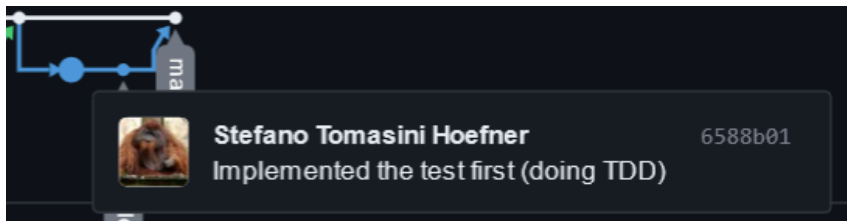
T03: Realizar TDD (Test Driven Development)



Para implementar la funcionalidad adicional de la aplicación, en este caso la opción de poder borrar la cuenta desde la base de datos, se ha usado la técnica de TDD (Test Driven Development). La característica principal del desarrollo por TDD es que antes de implementar cualquier funcionalidad, se implementa la prueba que falle, y después se implementa la propia funcionalidad que consiga pasar el test definido.

En la imagen de abajo se observa el test que se realizó antes de la implementación de la propia clase encargada de realizarlo.

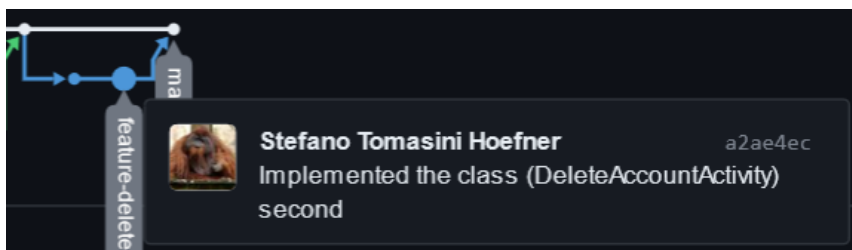
```
40  Stefano Tomasini Hoefner
41  @Before
42  public void setUp() {
43      mAuth = FirebaseAuth.getInstance();
44      mDatabase = FirebaseDatabase.getInstance();
45      dbUsersRef = mDatabase.getReference("Users");
46
47      mAuth.createUserWithEmailAndPassword(email: "testDelete@gmail.com", PSW);
48  }
49
50  2 usages
51  private final String PSW = "123456";
52
53  Stefano Tomasini Hoefner
54  @Test
55  public void testDelete() {
56
57      DatabaseReference dbUserRef = dbUsersRef.child("testDelete");
58
59      dbUserRef.setValue(new User());
60
61      dbUserRef.addListenerForSingleValueEvent(new ValueEventListener() {
62          2 usages Stefano Tomasini Hoefner
63          @Override
64          public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
65              assertNull(dataSnapshot.getValue());
66          }
67
68          Stefano Tomasini Hoefner
69          @Override
70          public void onCancelled(@NonNull DatabaseError databaseError) { fail(); }
71      });
72
73      Espresso.onView(ViewMatchers.withId(R.id.psw_delete_confirm)).perform(ViewActions.typeText(PSW));
74      closeSoftKeyboard();
75      Espresso.onView(ViewMatchers.withId(R.id.delete_button)).perform(ViewActions.click());
76
77      assertNull(mAuth.getCurrentUser());
78  }
```



[Commit de la creación del test para borrar la cuenta](#)

Posteriormente, se fue implementando la propia clase funcional. Al pasar la prueba definida anteriormente, se considera que ya está en un estado de funcionamiento correcto y listo para ser fusionado con la rama principal.

```
63  
1 usage  Stefano Tomasini Hoefner  
64 private void deleteAccount(String confirmPassword) {  
65     FirebaseUser currentUser = mAuth.getCurrentUser();  
66     if (currentUser != null) {  
67         AuthCredential credential = EmailAuthProvider.getCredential(currentUser.getEmail(), confirmPassword);  
68         currentUser.reauthenticate(credential).addOnCompleteListener(task -> {  
69             if (task.isSuccessful()) {  
70                 currentUser.delete().addOnCompleteListener(task1 -> {  
71                     if (task1.isSuccessful()) {  
72                         Toast.makeText(DeleteAccountActivity.this, "Se borró tu cuenta", Toast.LENGTH_SHORT).show();  
73                         mAuth.signOut();  
74                         Intent mainActivityIntent = new Intent(getApplicationContext(), MainActivity.class);  
75                         startActivity(mainActivityIntent);  
76                         finish();  
77                     } else {  
78                         Toast.makeText(DeleteAccountActivity.this, "No se pudo borrar tu cuenta", Toast.LENGTH_SHORT).show();  
79                     }  
80                 });  
81             } else {  
82                 Toast.makeText(DeleteAccountActivity.this, "Contraseña incorrecta", Toast.LENGTH_SHORT).show();  
83             }  
84         });  
85     }  
86 }  
87 }  
88 }
```



[Commit de la implementación de la clase DeleteAccountActivity](#)

La mayor ventaja del TDD es que te aseguras con certeza de que el código que se escribe es testable y de bajo acoplamiento, lo cual facilita mucho las tareas de mantenimiento y debugging en fases más tardías del desarrollo, cuando el sistema se vuelve más complejo, ya que se vuelve bastante más fácil identificar dónde se pueden encontrar los errores del programa.

T04: Realizar test unitario en la aplicación

4.1 - Test unitario de clase AdaptadorEventos

Este es un test unitario en Java utilizando el framework JUnit. Tiene como objetivo verificar si los métodos de la clase AdaptadorEventos se comportan correctamente.

La clase AdaptadorEventos es una clase extendida de la clase RecyclerView.Adapter y se utiliza para mostrar una lista de eventos en una vista de lista.

```
public class AdaptadorEventosTest {

    private AdaptadorEventos adaptador;

    @Before
    public void setUp() {
        adaptador = new AdaptadorEventos(c: null);
    }

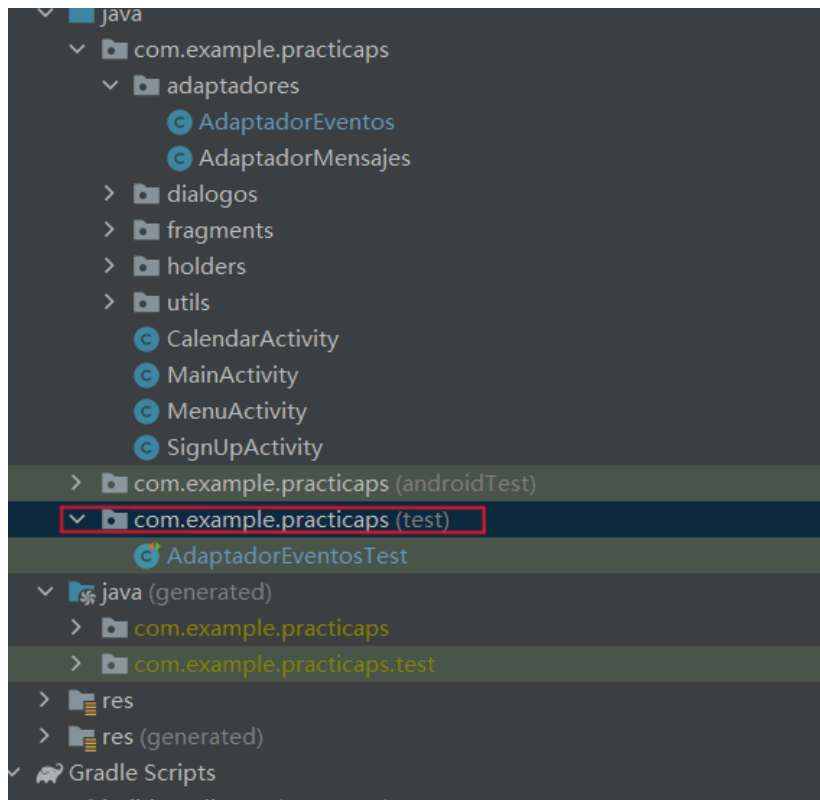
    @Test
    public void addEventoTest() {
        // Check if addEvento method increases item count
        int itemCount = adaptador.getItemCount();
        adaptador.addEvento(new EventInfo( info: "Event 1", hour: 10, minute: 30));
        assertEquals( expected: itemCount + 1, adaptador.getItemCount());

        // Add Event 2
        itemCount = adaptador.getItemCount();
        adaptador.addEvento(new EventInfo( info: "Event 2", hour: 14, minute: 0));
        assertEquals( expected: itemCount + 1, adaptador.getItemCount());
    }

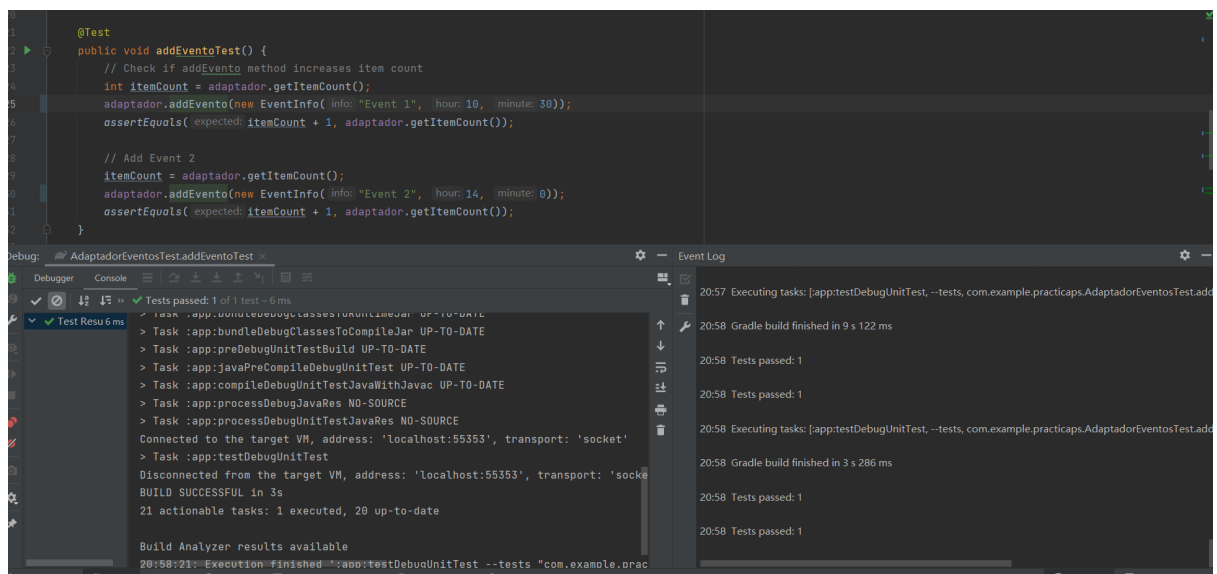
    @Test
    public void getItemCountTest() {
        // Check if getItemCount() returns correct count
        adaptador.addEvento(new EventInfo( info: "Event 1", hour: 10, minute: 30));
        adaptador.addEvento(new EventInfo( info: "Event 2", hour: 14, minute: 0));

        assertEquals( expected: 2, adaptador.getItemCount());
    }
}
```

Este test unitario ha sido ejecutado en la JVM sin la necesidad del emulador Android. Esto permite a los desarrolladores ejecutar test más rápido, y es más fácil de depurar y de mantener, ya que no están sujetas a las limitaciones de la plataforma android. También más independiente del entorno.

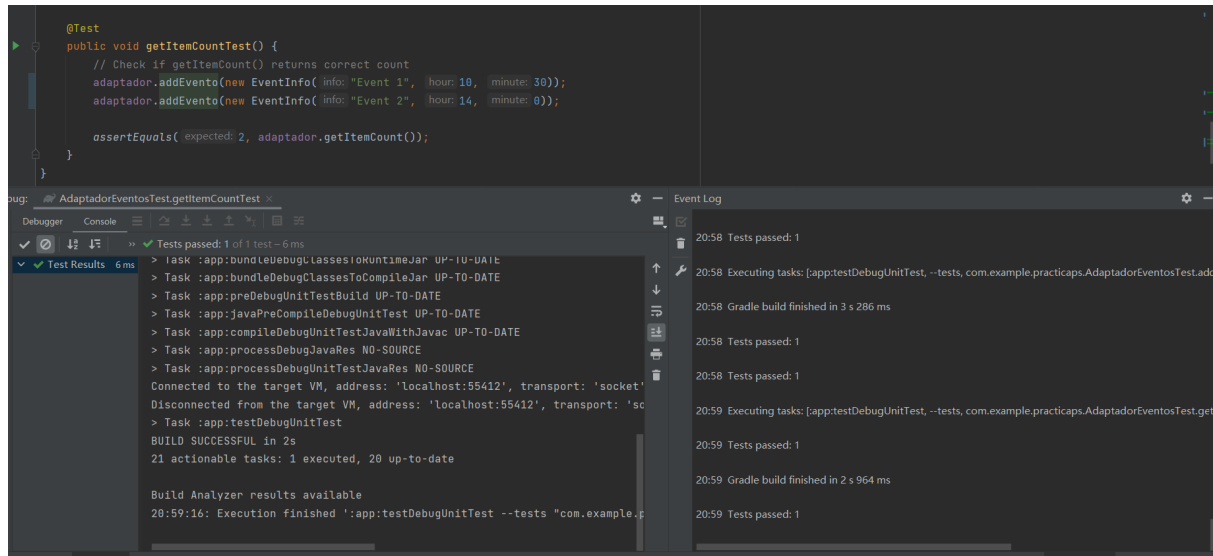


El primer test unitario, `addEventoTest()` verifica si el método `addEvento()` de la clase `AdaptadorEventos` funciona correctamente al agregar un evento. Primero se almacena el número de elementos en el adaptador. Luego se agrega un evento nuevo usando el método `addEvento` y se comprueba si el número de elementos(0) ha aumentado en uno. Luego se agrega otro evento y se comprueba si el número de elementos ha aumentado en uno de nuevo.



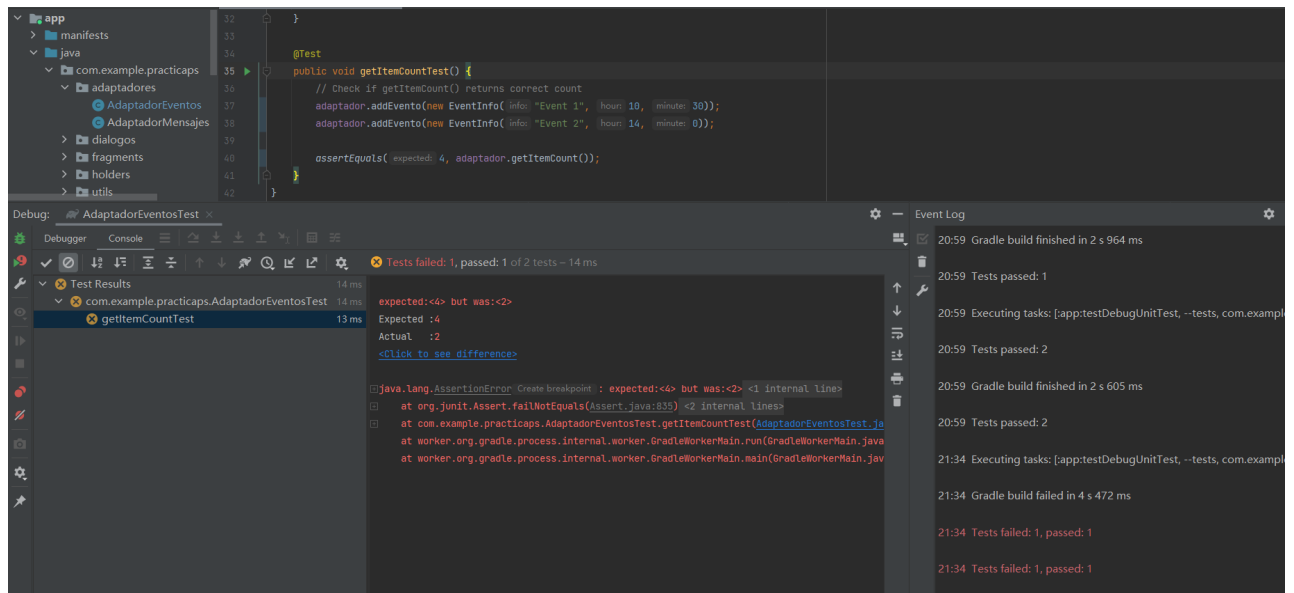
(addEventosTestResult)

El segundo test, `getItemCountTest()` verifica si el método `getItemCount()` funciona correctamente. Se agregan dos eventos utilizando el método `addEvento` y se comprueba si el número de elementos devuelto por `getItemCount` es igual a 2.



(`getItemCountTestResult`)

Si cambia el valor 2 del `assertEquals(2, adaptador.getItemCount())` a 4, el test espera que se hayan agregado cuatro elementos al adaptador, lo que no es cierto, porque se han agregado solamente dos elementos, y por lo tanto el test fallará porque el valor esperado (4) no coincide con el valor real devuelto por el método `getItemCount()`.



(`getItemCountTestResult`)

4.2 - Test unitario de clase `AdaptadorMensajes`

Este test unitario, al igual que el anterior, utiliza el framework JUnit. Su objetivo es verificar los métodos de la clase `AdaptadorMensajes`.

La clase `AdaptadorEventos` es una clase extendida de la clase `RecyclerView.Adapter` y se utiliza para mostrar una lista de eventos en una vista de lista.

```
package com.example.practicaps;

import ...

public class MessageTest {
    5 usages
    private AdaptadorMensajes adaptador;

    @Before
    public void setUp() { adaptador = new AdaptadorMensajes( null); }

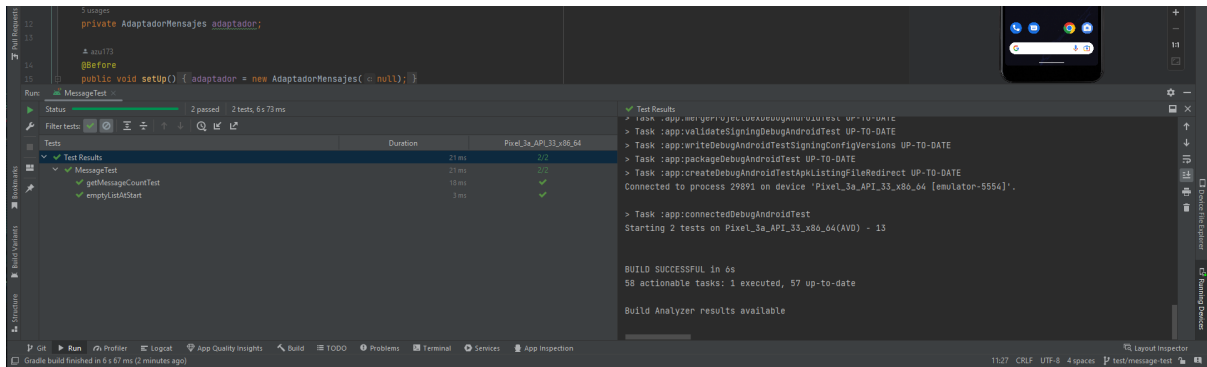
    @Test
    // Checks if the adapter is empty at start
    public void emptyListAtStart() {
        int itemCount = adaptador.getItemCount();
        assertEquals(itemCount, 0);
    }

    @Test
    // Checks if the adapter item count goes up when storing a new message
    public void getMessageCountTest() {
        int itemCount = adaptador.getItemCount();
        adaptador.addMensaje(new MensajeRecibir( nombre: "test_name", mensaje: "test_content", type_mensaje: "1", System.currentTimeMillis()));
        assertEquals( expected: itemCount + 1, adaptador.getItemCount());
    }
}
```

El primer test unitario, `emptyListAtStart()`, simplemente verifica que la lista del adaptador se encuentra vacía al crear una instancia de la clase. Aunque puede parecer bastante obvio a primera vista que no, si se complica el programa, es posible que en algún punto esta condición no se cumpla y genere problemas al no tener conocimiento de este comportamiento. El contenido del test es una comprobación sencilla de que `adaptador.getItemCount()` es igual a 0.

El segundo test unitario, `getMessageCountTest()`, comprueba que al añadir un objeto a la lista mediante el método proporcionado por el adaptador, se está aumentando el número de elementos de la propia lista del objeto. Para ello, primero obtenemos el número de objetos de la lista antes de iniciar el propio test (en principio debería ser 0) y añadimos un nuevo mensaje al adaptador. Por último, comprobamos que la longitud inicial de la lista más 1 es igual a `adaptador.getItemCount()`.

Se consideró realizar un tercer test que comprobase si coincidía el contenido del mensaje creado con el almacenado en la lista, pero al no disponer la clase del adaptador de ningún método para extraer mensajes directamente de la lista se ha decidido no implementar el método para mantener la simplicidad y la funcionalidad de la clase.



T05: Realizar test de integración en la aplicación

El test de integración es una técnica de testing de software que se utiliza para verificar que los diferentes componentes o módulos de un sistema interactúan de manera correcta y coordinada entre sí.

El objetivo es probar cómo se relacionan los distintos componentes y cómo se comportan en conjunto, implicando la creación de un ambiente de prueba que simula el ambiente real de producción.

5.1 Test de la base de datos

El primer test de integración realizado fue sobre la base de datos firebase que utiliza la aplicación para guardar registro de los usuarios y eventos presentes.

La prueba consiste en insertar un nuevo usuario en la base de datos y comprobar que este ha sido, en efecto, insertado. Por tanto, su objetivo es comprobar el correcto funcionamiento de la base de datos con las operaciones que se realizan sobre ella.

Durante la producción del test nos encontramos con varios problemas relativos al funcionamiento de la base de datos. Entre ellos, la selección de usuarios nulos y la imposibilidad de probar la base de datos debido a un problema relativo a quién era el propietario de esta. Finalmente, se pudieron arreglar todos estos problemas, con lo que obtuvimos el código mostrado a continuación:

```

@RunWith(AndroidJUnit4.class)
public class DatabaseTest {
    @Rule
    public ActivityTestRule<MainActivity> mActivityRule = new ActivityTestRule<>(MainActivity.class);

    private DatabaseReference mDatabaseReference;

    @Before
    public void setup() {
        // Initialize FirebaseApp and DatabaseReference
        FirebaseApp.initializeApp(mActivityRule.getActivity());
        mDatabaseReference = FirebaseDatabase.getInstance().getReference("users");
    }

    @Test
    public void testDatabase() {
        // Insert a record into the Firebase database
        String key = mDatabaseReference.push().getKey();
        mDatabaseReference.child(key).child("name").setValue("John Doe");

        // Verify that the record was inserted correctly
        Task<DataSnapshot> task = mDatabaseReference.child(key).get();
        task.addSuccessListener(snapshot -> {

```

5.2 - Test de la clase SignUp

Para la segunda prueba, seleccionamos la clase de aplicación responsable de registrar nuevos usuarios. Esta clase también usa una base de datos.

Al comenzar a hacer la prueba, encontramos algunos problemas en el código fuente y el funcionamiento de la base de datos. El programa seleccionaba a un usuario cuyo valor era nulo, lo que provocó que toda la prueba colapsara y arrojase un error. Después de cambiar las condiciones del usuario seleccionado para realizar la prueba, pudimos lograr nuestro objetivo y realizar una prueba exitosa.

La siguiente imagen representa el código del test de integración de la clase SignUp:

```
@After
public void tearDown() {
    FirebaseAuth user = FirebaseAuth.getInstance().getCurrentUser();
    assert user != null;
    user.delete();
}

@Test
public void testSignUp() {
    String email = "test@gmail.com";
    String password = "123456";
    String name = "Test";
    String surname = "User";

    // Fill in the EditText fields with the test data
    Espresso.onView(ViewMatchers.withId(R.id.edit_email_sig)).perform(ViewActions.typeText(email));
    Espresso.onView(ViewMatchers.withId(R.id.edit_pass_sig)).perform(ViewActions.typeText(password));
    Espresso.onView(ViewMatchers.withId(R.id.edit_confirm_pass_sig)).perform(ViewActions.typeText(password));
    Espresso.onView(ViewMatchers.withId(R.id.edit_name_sig)).perform(ViewActions.typeText(name));
    Espresso.onView(ViewMatchers.withId(R.id.edit_lastnames_sig)).perform(ViewActions.typeText(surname));

    closeSoftKeyboard();
    // Click on the Sign Up button
    Espresso.onView(ViewMatchers.withId(R.id.button_sign)).perform(ViewActions.click());

    // Wait for the authentication to complete
    try {
        Thread.sleep( millis: 5000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    // Check if the user is successfully authenticated
    assertNotNull(mAuth.getCurrentUser());

    // Check if the user data is successfully saved in the Firebase database
    assertNotNull(dbUserRef.child(mAuth.getCurrentUser().getUid()));
}
```

T06: Realizar tests de interfaz en la aplicación

La función de los tests de interfaz de nuestra aplicación es probar que al realizar cambios que vayan a modificar la interfaz o la pantalla, ya sea al iniciar sesión e ir al menú, añadir alguna actividad o recordatorio al calendario o cualquier otra actividad que cambie la interfaz, esta se muestre correctamente y sin ningún tipo de error o fallo.

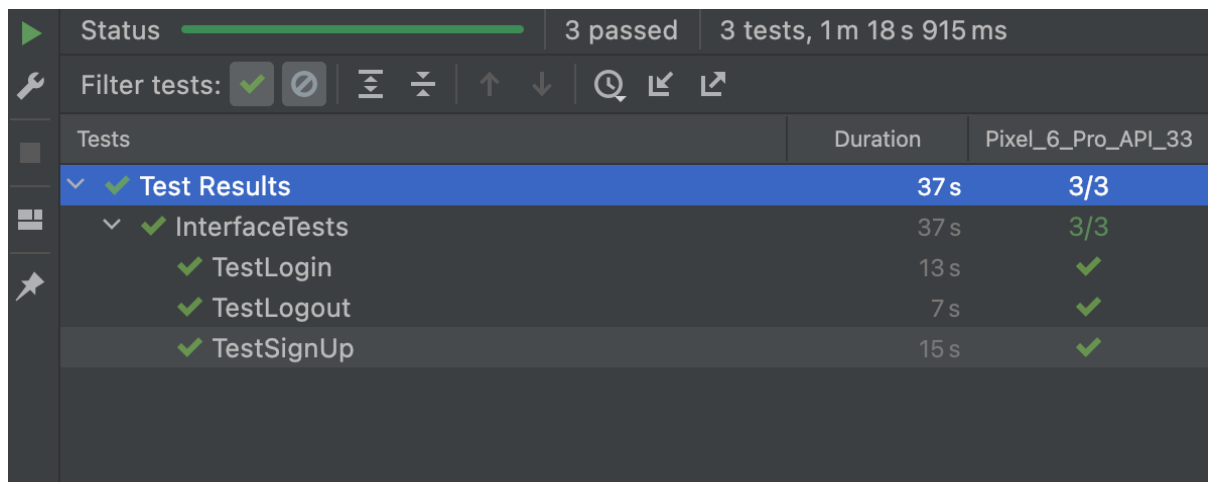
A parte, queremos asegurar que la interfaz sea fácil de usar, accesible e intuitiva. Cabe destacar que la aplicación que se nos presentó estaba, además de incompleta y poco intuitiva, nada usable. Por ello ha sido bastante complicado realizar los tests que vamos a presentar a continuación:

Los 3 tests de interfaz que se han implementado son los siguientes:

- Test de inicio de sesión
- Test de registro
- Test de cierre de sesión

Con estos tres tests, hemos probado lo más importante de la interfaz de la app, que al iniciar sesión te muestre el menú, que al registrarte vuelva a la página inicial para iniciar sesión y que al cerrar sesión vuelva a la pantalla de inicio de sesión.

Como podemos ver en la foto, los tres tests han pasado sin ningún tipo de problema:



Status	3 passed	3 tests, 1 m 18 s 915 ms
Filter tests:	[Icons]	
Tests	Duration	Pixel_6_Pro_API_33
✓ Test Results	37 s	3/3
✓ InterfaceTests	37 s	3/3
✓ TestLogin	13 s	✓
✓ TestLogout	7 s	✓
✓ TestSignUp	15 s	✓

El primer test es el de Iniciar Sesión, hemos usado el Espresso para rellenar los datos automáticamente y hemos comprobado que al iniciar sesión vaya al menú principal:

```
@Test
public void TestLogin(){
    mActivityRule.launchActivity(new Intent());
    String email = "testInterfaz@gmail.com";
    String password ="123456";

    //Rellenamos los campos de email y contraseña con los datos de la cuenta existente
    Espresso.onView(ViewMatchers.withId(R.id.edit_usuario_log)).perform(ViewActions.typeText(email));
    Espresso.onView(ViewMatchers.withId(R.id.edit_pass_log)).perform(ViewActions.typeText(password));
    Espresso.onView(ViewMatchers.withId(R.id.button_log)).perform(ViewActions.click());
}
```

```

Andreas4122002
@Test
public void TestSignUp() throws InterruptedException {
    mActivityRule.launchActivity(new Intent());
    Espresso.onView(ViewMatchers.withId(R.id.button_registro)).perform(ViewActions.click());
    String name = "testInterfaz";
    String apellido = "testInterfaz";
    String email = "prueba@gmail.com";
    String pwd = "prueba12345";
    Espresso.onView(ViewMatchers.withId(R.id.edit_name_sig)).perform(ViewActions.typeText(name));
    Espresso.onView(ViewMatchers.withId(R.id.edit_lastnames_sig)).perform(ViewActions.typeText(apellido));
    Espresso.onView(ViewMatchers.withId(R.id.edit_email_sig)).perform(ViewActions.typeText(email));
    Espresso.onView(ViewMatchers.withId(R.id.edit_pass_sig)).perform(ViewActions.typeText(pwd));
    Espresso.onView(ViewMatchers.isRoot()).perform(ViewActions.closeSoftKeyboard());
    Espresso.onView(ViewMatchers.withId(R.id.edit_confirm_pass_sig)).perform(ViewActions.typeText(pwd));
    Espresso.onView(ViewMatchers.isRoot()).perform(ViewActions.closeSoftKeyboard());
    Espresso.onView(ViewMatchers.withId(R.id.button_sign)).perform(ViewActions.click());
    Espresso.onView(ViewMatchers.withId(R.id.edit_usuario_log)).check(ViewAssertions.matches(ViewMatchers.isDisplayed()));
    Espresso.onView(ViewMatchers.withId(R.id.edit_pass_log)).check(ViewAssertions.matches(ViewMatchers.isDisplayed()));
    Espresso.onView(ViewMatchers.withId(R.id.button_log)).check(ViewAssertions.matches(ViewMatchers.isDisplayed()));
}

```

Por último el test de cierre de sesión, simplemente abrimos una instancia del menú principal y hacemos click en el botón superior derecho para cerrar sesión. Comprobamos que nos devuelve a la pantalla de iniciar sesión por lo que el test funciona correctamente:

```

Andreas4122002 +1
@Test
public void TestLogout() {
    //Comprobamos que se cierra sesión y vuelve a la pantalla de inicio de sesión
    mMenuActivityRule.launchActivity(new Intent());
    Espresso.onView(ViewMatchers.withId(R.id.cerrar_sesion)).perform(ViewActions.click());

    //Comprobar que se ha cambiado a la interfaz de la pantalla de inicio de sesión y registro
    Espresso.onView(ViewMatchers.withId(R.id.edit_usuario_log)).check(ViewAssertions.matches(ViewMatchers.isDisplayed()));
    Espresso.onView(ViewMatchers.withId(R.id.edit_pass_log)).check(ViewAssertions.matches(ViewMatchers.isDisplayed()));
    Espresso.onView(ViewMatchers.withId(R.id.button_log)).check(ViewAssertions.matches(ViewMatchers.isDisplayed()));
}

```

T07: Automatización de la ejecución de los tests con CircleCI

La automatización de la ejecución de los tests está en el GitHub. Hemos usado la herramienta CircleCI integrada en nuestro repositorio para poder llevar a cabo esta tarea:

