Las cosas que no vemos

Ana María Martínez Valdés
David Chaparro García
Eros Guerrero Sosa
Iñigo Rolando García
Ignacio Villegas de Miquel
Johan Cruz Huertas
Javier Gil Caballero
Luis Enrique Barrero Peña
Santiago Gabriel y Galán Puelles
Shuyi Wang

Índice

1. Introducción

Como es usual, para la realización de esta práctica, el coordinador, Santiago Gabriel y Galán, dividió el trabajo entre todos los miembros del grupo teniendo en cuenta su rol y la afinidad entre los mismos. De esta manera, al equipo de desarrollo le fue asignada la parte del plan de pruebas e integración continua y al Scrum Master (SM) y Product Owner (PO) la corrección del Product Backlog (PB) en Jira, rellenar el apartado del Product Backlog en la memoria y la introducción. Además, el SM se encargó de la dinámica grupal y como también es coordinador, realizó una tabla (tabla 1.1) para que la división del trabajo quedase más clara.

	PARTE	actividad 1	actividad 2	DESCRIPCIÓN
Ana	Pruebas	apoyo prueba	2.a memoria	apoyo prueba y rellenar en memoria
David	Pruebas	apoyo prueba	2.b memoria	apoyo prueba y rellenar en memoria
Eros	Memoria	parte 1 memoria	PB	realizar revision PB y rellenar memoria
Íñigo	Pipeline	apoyo pipeline	apoyo prueba	apoyo en prueba y pipeline
Javi	Pipeline	hacer pipeline	3.a y 3.b	realizar pipeline y rellenar memoria
Johan	Pruebas	hacer prueba		encargado principal prueba
Kike	Pipeline	hacer pipeline	3.b y 3.c	realizar pipeline y rellenar memoria
Nacho	Pruebas	apoyo prueba	4 memoria	apoyo en pruebas y rellenar memoria
Santi	Memoria	parte 5 memoria	PB	realizar revision PB y rellenar memoria
Shuyi	Pipeline	hacer pipeline	3.b (pipeline)	encargada principal pipeline

Tabla 1.1 “Tabla con el reparto de trabajo”

2. Plan de Pruebas

En este punto dos de la memoría se explica el plan de pruebas que hemos decidido para el desarrollo del proyecto. Se detalla qué pruebas hemos decidido realizar y quienes se han encargado del desarrollo de estas, a parte de una explicación breve de su finalidad y de qué resultados deben dar para saber si son o no correctas. Por otro lado en la segunda parte de este punto explicamos la prueba piloto que hemos llevado a cabo para probar las herramientas de automatización que hemos seleccionado.

2.1. Pruebas

Las **pruebas de sistema** las llevarán a cabo Ana María Martínez y Javier Gil de manera manual. Tendremos cuatro tipos de pruebas diferenciadas:

Las primeras serán las pruebas de funcionalidad, que consistirán en probar todas las acciones que tenga la aplicación y asegurarse de que todo sale tal y como se esperaba. En nuestra idea de aplicación, un ejemplo muy sencillo podría ser comprobar si al introducir un nombre al hacer la identificación, si este ya existe, debería mostrar un mensaje de error advirtiendo de ello.

Por otro lado estarán las pruebas de carga, estas valorarán la funcionalidad del sistema con una carga de usuarios variables, pero siempre dentro de los valores esperados por la aplicación. En nuestro caso, valoramos que estén haciendo uso de ella tanto alumnos de un colegio como personas externas interesadas en esta al mismo tiempo. Estas pruebas las llevaremos a cabo al final de cada sprint para comprobar que la aplicación sigue funcionando correctamente y que está pensada para aguantar la carga de trabajo con la que contará normalmente.

En tercer lugar están las pruebas de estrés, en las que se someterá al sistema a una gran carga de trabajo hasta que falle. Como estas pruebas simulan situaciones casi irreales, no les daremos gran importancia, por lo que no serán realizadas con mucha frecuencia, y, en caso de que se hagan, normalmente se realizan al final de cada sprint.

Por último las de usabilidad, unas pruebas que consideramos de gran importancia ya que la aplicación está pensada para todo tipo de públicos, incluidos niños más pequeños o personas con problemas, por lo que buscamos que sea lo más sencilla posible tanto para entender lo que se pide, como para moverse por la aplicación. Las pruebas de usabilidad las realizaremos con mucha frecuencia, ya que si no, no conseguiremos nuestro objetivo de tener una aplicación que todo el mundo pueda manejar. Como ejemplo de este tipo de pruebas estaría mirar que todo se entiende bien, y que nada da lugar a equivocación. Cada vez que se haga una nueva funcionalidad, o se escriba un nuevo texto de teoría a tratar, se realizarán este tipo de pruebas.

En todas estas pruebas de sistema se documentará todas las variables que se han tenido en cuenta en su ejecución y la salida esperada (en caso de que la prueba tenga una salida concreta) además de distintas anotaciones que se consideren necesarias para mejorar la funcionalidad propuesta.

Las **pruebas de unidad** las realizarán Ignacio Villegas e Íñigo Rolando y se implementarán cada vez que se termine una funcionalidad que necesite ser testeada. Estarán implementadas con JUnit, por lo que se harán de forma automática, ejecutando la clase que los implemente podremos saber si la aplicación actual pasa o no los test. Dado que prueban funcionalidades muy específicas, estos test estarán en su mayoría incluidos dentro de otros test que abarquen una función más grande. Es por esto que solo se ejecutarán de manera aislada en ocasiones concretas, como por ejemplo, cuando se ha cambiado una funcionalidad que ya estaba testeada.

De cada test se deberá documentar si ha sido correcto o incorrecto y, en caso de ser incorrecto, se documentará cuál es el error o excepción que se ha producido para poder arreglarlo.

Las **pruebas de aceptación** también serán pruebas manuales que pasarán por tres fases:

En la primera fase, Alpha, los desarrolladores probarán la aplicación siguiendo una serie de pasos que les harán pasar por todas las nuevas funcionalidades que se hayan implementado. Se documentará cómo se comporta la aplicación tras la prueba, teniendo en cuenta lo que ha hecho y lo que debería hacer siguiendo la especificación.

La segunda fase, Betha, la tendrá que realizar un usuario final, y se comprobará si cumple con todos los requisitos pedidos y su correcto funcionamiento. Se pedirá que estos usuarios documenten si la aplicación cumple el funcionamiento esperado o no, y en caso de que no lo cumpla, que se especifiquen los cambios que consideren pertinentes.

Por último, la aceptación del cliente, el product owner, Eros Guerrero, hará la prueba definitiva del producto, para comprobar que tiene todo lo que solicitaba y que cumple con sus especificaciones.

Las **pruebas de integridad** las realizarán Johan y David y se implementarán cada vez que tenga lugar un gran cambio en la aplicación o se acaben un conjunto de clases que interactúan entre ellas, de esta manera se comprobará que la relación que hay entre ellas se lleva a cabo de una manera correcta. Estas pruebas están implementadas con Espresso y con AndroidJUnit4 y se situarán en la carpeta de AndroidTest. Estos test están automatizados a través del propio Android Studio, que te deja ejecutarlas como si fuese un test unitario y te muestra por pantalla si se ha pasado de manera satisfactoria o si ha surgido un error durante la ejecución del mismo.

Una vez ejecutados los test de integridad sería importante documentar si el resultado ha sido correcto, en cuyo caso se debe aportar mayor información. Si por el contrario han surgido fallos y los algún test no ha pasado de manera satisfactoria, se debe explicar qué clases han fallado y/o están dando problemas, para que se pueda solucionar en el futuro.

2.2. Prueba Piloto

En la prueba piloto del plan de pruebas, se han probado las pruebas de unidad y las de integración, ya que a diferencia de las de sistema o aceptabilidad, que se harán de forma manual; son las únicas que estarán automatizadas. Ambas dos están automatizadas mediante diferentes librerías que hemos tenido que importar en Android Studio (como JUnit o Mockito).

Para la automatización de las **pruebas unitarias**, tal como se ha visto anteriormente, se ha usado JUnit. Para probar la funcionalidad de estos test, se usó como ejemplo la función que se muestra en la imagen 2.2.1

```
public boolean comprobarFormato(String nombre, String descripcion){
    if (nombre.equals("") || descripcion.equals("") || nombre.length() > 50 || descripcion.length() > 200) return false;
    else return true;
}
```

Imagen 2.2.1 “Código de función de prueba”

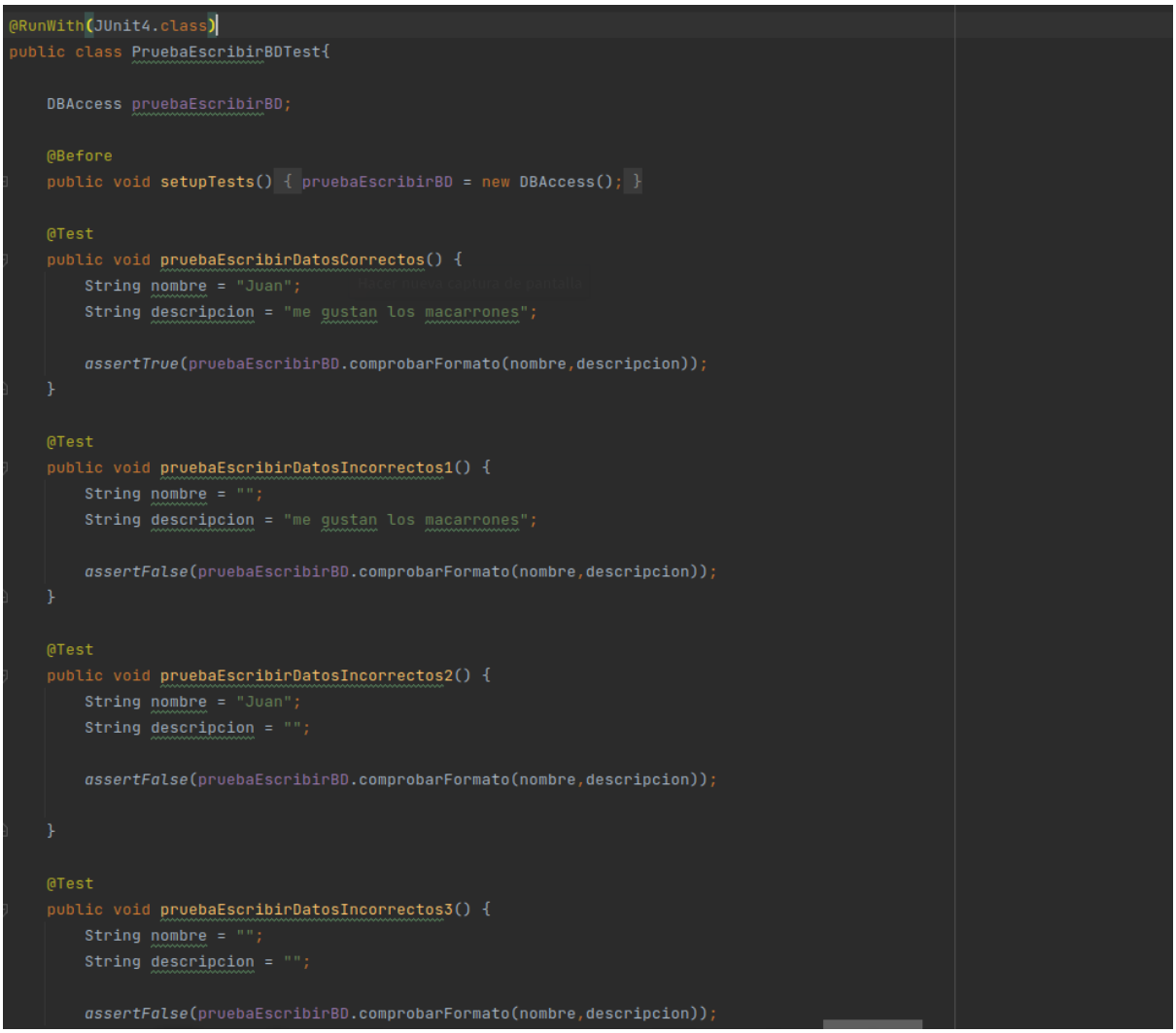
Esta función comprueba el formato de dos campos de texto (nombre y descripción), que son los campos que usamos en el prototipo de funcionalidades, devolviendo *true* en caso de que el formato esté correcto y *false* en caso contrario. Se decidió probar esta función ya que será una función con bastante importancia dentro de la aplicación en un futuro, ya sea para comprobar el formato de texto, imágenes…

Se han creado cuatro test, el primero comprueba que la función devuelve true si se introducen los datos con el formato correcto y las otras tres comprueban

que la función devuelva false al introducir datos incorrectos. El código utilizado para implementar estos test se muestra en la imagen 2.2.2

Imagen 2.2.2 “Código de las pruebas de unidad”

Para poder probar los test implementados y en general, cualquier test unitario



```
@RunWith(JUnit4.class)
public class PruebaEscribirBDTest{

    DBAccess pruebaEscribirBD;

    @Before
    public void setupTests() { pruebaEscribirBD = new DBAccess(); }

    @Test
    public void pruebaEscribirDatosCorrectos() {
        String nombre = "Juan";
        String descripcion = "me gustan los macarrones";

        assertTrue(pruebaEscribirBD.comprobarFormato(nombre,descripcion));
    }

    @Test
    public void pruebaEscribirDatosIncorrectos1() {
        String nombre = "";
        String descripcion = "me gustan los macarrones";

        assertFalse(pruebaEscribirBD.comprobarFormato(nombre,descripcion));
    }

    @Test
    public void pruebaEscribirDatosIncorrectos2() {
        String nombre = "Juan";
        String descripcion = "";

        assertFalse(pruebaEscribirBD.comprobarFormato(nombre,descripcion));

    }

    @Test
    public void pruebaEscribirDatosIncorrectos3() {
        String nombre = "";
        String descripcion = "";

        assertFalse(pruebaEscribirBD.comprobarFormato(nombre,descripcion));
```

que se realice para la aplicación: descargamos la última versión del proyecto desde el repositorio de GitHub y localizamos los test unitarios que se encuentran en la carpeta /app/source/test/java. Una vez dentro de esta carpeta podremos dar click derecho encima de la clase para que se ejecuten los test, tal como se muestra en la imagen 2.2.3.

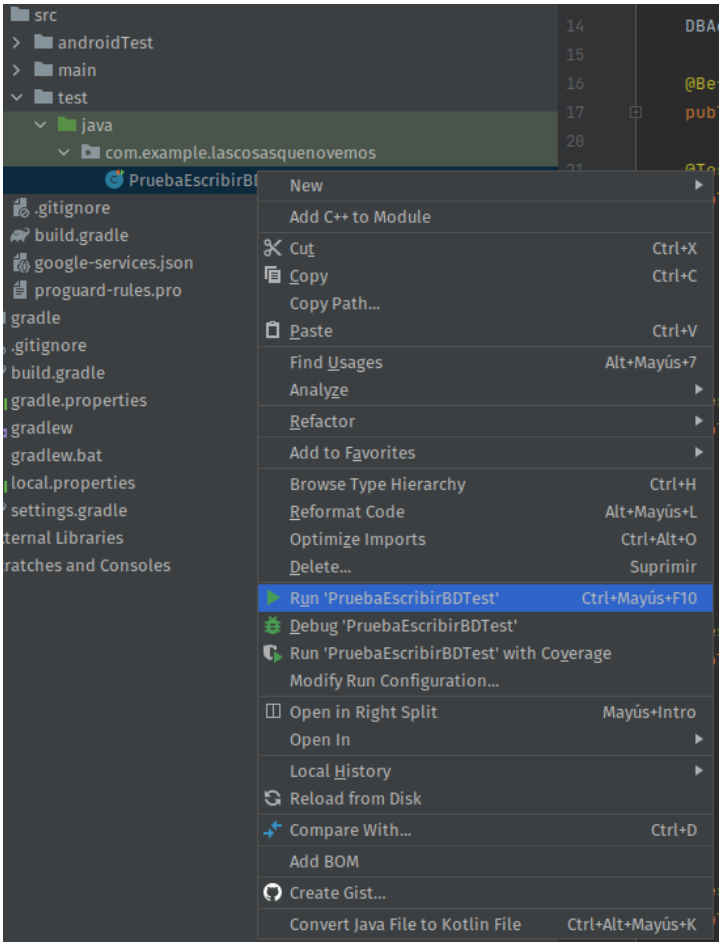


Imagen 2.2.3 “Modo de ejecutar todas las pruebas de unidad”

Una vez que se ejecute la aplicación tendremos que mirar la salida por consola para verificar que no ha habido ningún error independiente a los test (como puede ser un error al compilar el proyecto o que se lance una excepción), además de comprobar que se han pasado todos los test.

En el caso de la prueba que se ha realizado, se siguieron los pasos explicados anteriormente y se obtuvo la salida por consola que se puede ver en la imagen 2.2.4.

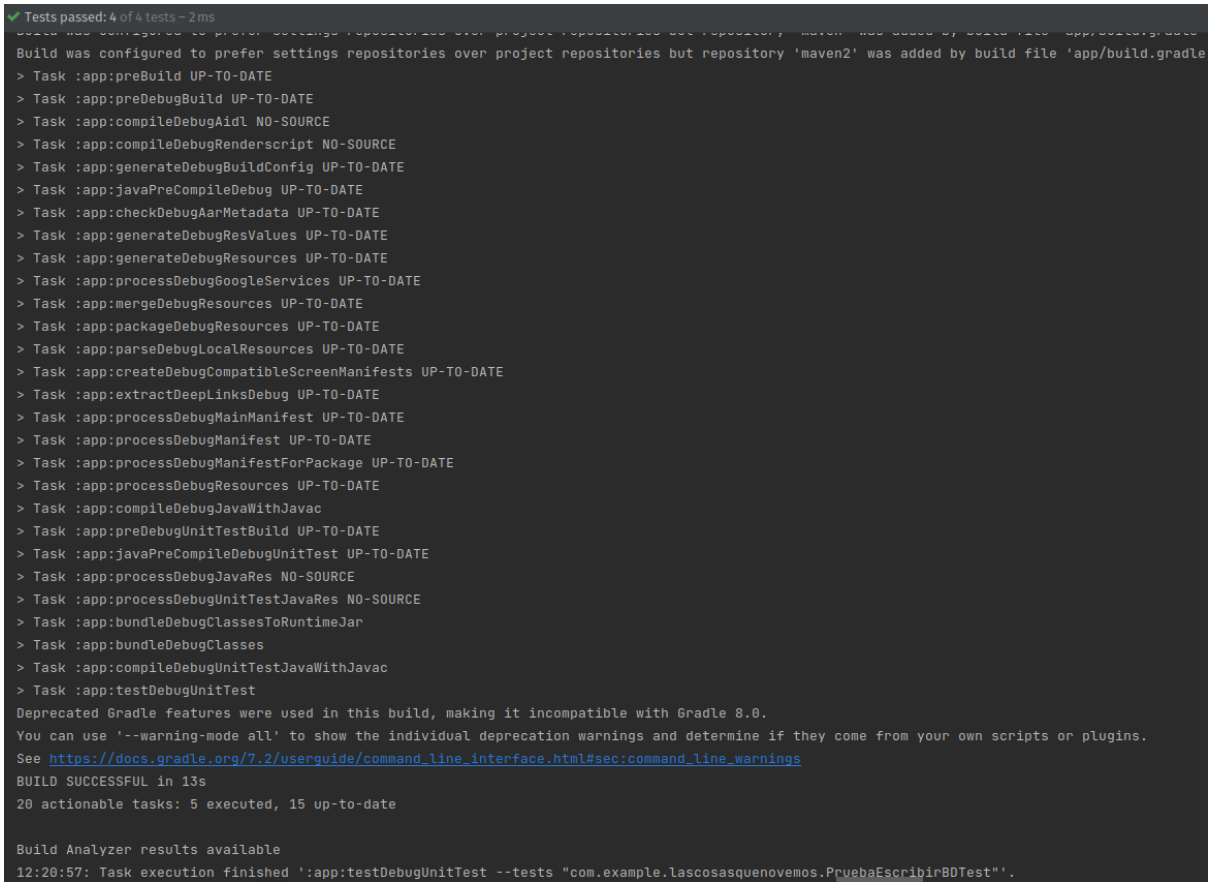
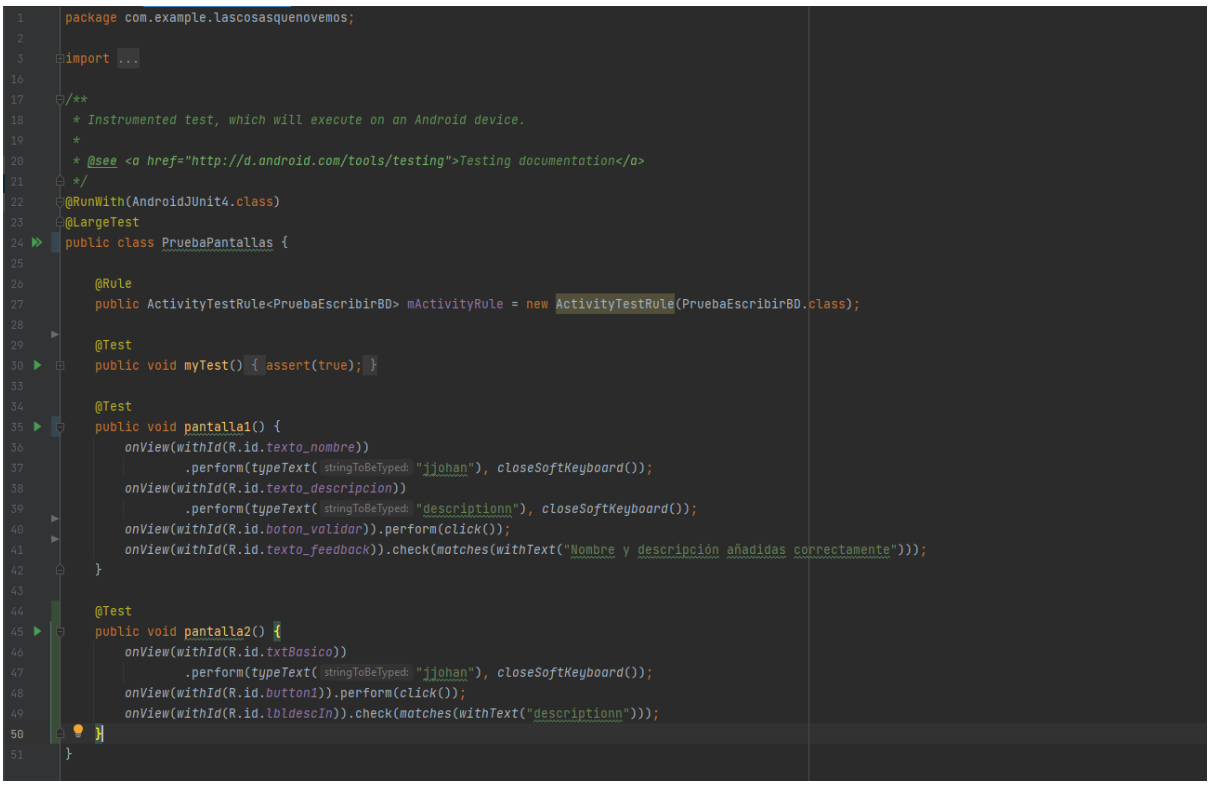


Imagen 2.2.4 “Salida por consola de las pruebas de unidad”

Para la automatización de las **pruebas de integración** y tras comparar diferentes opciones que hemos visto por internet, nos hemos decantado por Espresso y AndroidJUnit4, como hemos indicado anteriormente. En un principio íbamos a usar TestNG (porque tiene funciones para que un test solo se ejecute cuando otro ha pasado), pero la dificultad de encontrar información aplicada a Android Studio ha provocado que de manera consensuada hayamos buscado otras opciones.

El código que hemos implementado (se puede ver en la imagen para la prueba consta de tres tests, el primero es un test de prueba que únicamente hace un assert de true, es decir siempre tiene que dar como test pasado, esto nos sirve para probar que los test están teniendo lugar de una manera correcta. El segundo test que se ejecuta utiliza la primera actividad de la aplicación, en la cual se introduce un nombre y una descripción y luego clickea el botón que hace que se ejecute el método (de PruebaEscribirBD) que guarda lo previamente escrito en la base de datos; seguidamente se comprueba el campo de feedback que habíamos implementado en la activity ha escrito el mensaje “Nombre y descripción añadidas correctamente”, es decir que la acción se ha llevado a cabo de una manera correcta. Por último, el tercer test escribe en la segunda actividad (PruebaLeerBD) el nombre escrito durante el test anterior en un EditText, esto hace que al activar el botón de esa misma

pantalla nos devuelva el texto que hemos escrito antes y que se ha guardado en la BD, finalmente hacemos una comparación para comprobar que efectivamente se cumple lo que hemos dicho anteriormente y que el texto devuelto es el mismo al que hemos. De esta manera hemos comprobado que la clase PruebaEscribirBD ha llevado sus funciones a cabo de manera correcta para que posteriormente PruebaLeerBD pueda ejecutar las suyas.



```
package com.example.lascosasquenovemos;

import ...

/**
 * Instrumented test, which will execute on an Android device.
 *
 * @see <a href="http://d.android.com/tools/testing">Testing documentation</a>
 */
@RunWith(AndroidJUnit4.class)
@LargeTest
public class PruebaPantallas {

    @Rule
    public ActivityTestRule<PruebaEscribirBD> mActivityRule = new ActivityTestRule(PruebaEscribirBD.class);

    @Test
    public void myTest() { assert(true); }

    @Test
    public void pantalla1() {
        onView(withId(R.id.texto_nombre))
                .perform(typeText( stringToBeTyped: "jjohan"), closeSoftKeyboard());
        onView(withId(R.id.texto_descripcion))
                .perform(typeText( stringToBeTyped: "descriptionn"), closeSoftKeyboard());
        onView(withId(R.id.boton_validar)).perform(click());
        onView(withId(R.id.texto_feedback)).check(matches(withText("Nombre y descripción añadidas correctamente")));
    }

    @Test
    public void pantalla2() {
        onView(withId(R.id.txtBasico))
                .perform(typeText( stringToBeTyped: "jjohan"), closeSoftKeyboard());
        onView(withId(R.id.button1)).perform(click());
        onView(withId(R.id.lbldescIn)).check(matches(withText("descriptionn")));
    }
}
```

Imagen 2.2.5 “Código tests de integración”

Para poder probar los test de integración los localizamos en la carpeta de androidTest y se ejecutan igual que los test de unidad, sin embargo es necesario tener un dispositivo conectado en el que la aplicación pueda funcionar mientras se prueban. En la imagen 2.2.6 se muestra que opción se debe seleccionar una vez se hace click derecho sobre el test que se quiere comprobar.

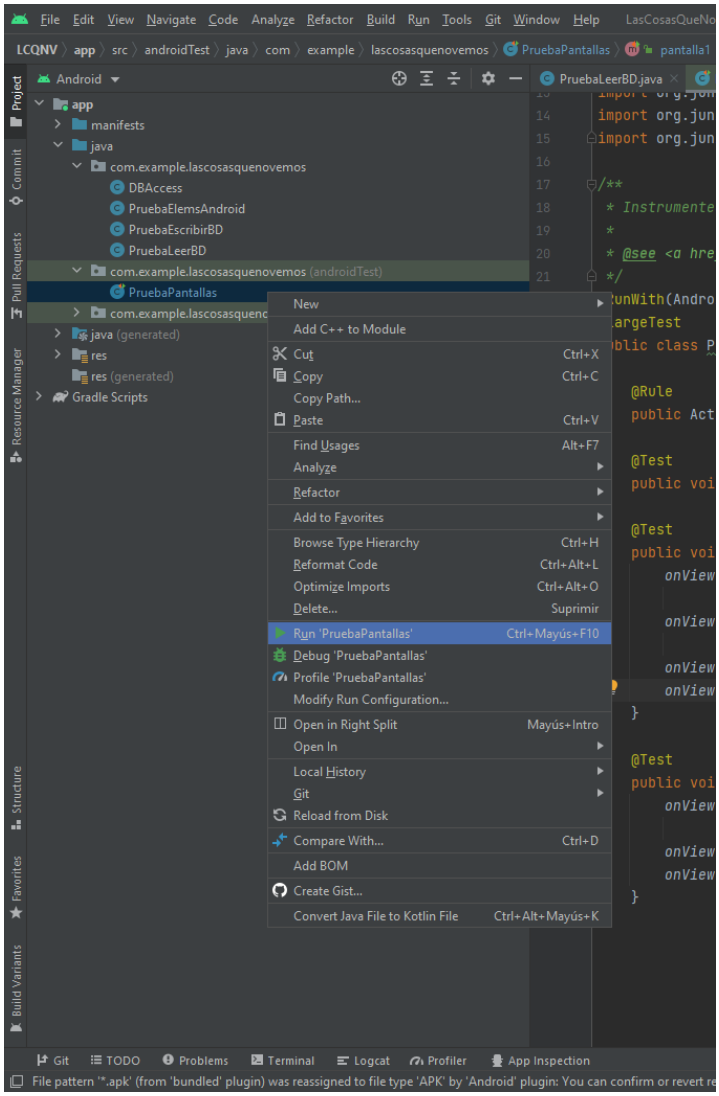


Imagen 2.2.6 “Modo de ejecutar un test de integración”

3. Integración Continua

En el presente punto tres de la memoría se explica todo lo referente a la integración continúa que vamos a usar en nuestro proyecto. Se hace un análisis de diferentes herramientas que permiten esta funcionalidad y por qué finalmente nos hemos decantado por la que hemos elegido (GitHub). También se explica cómo funciona la herramienta seleccionada y qué pasos debemos seguir para poder utilizarla de manera correcta. Y por último se narra cómo ha sido la primera experiencia, la prueba piloto, con esta aplicación y/o herramienta.

3.1. Descripción de herramienta

La primera herramienta que probamos para la integración continua fue Bamboo, es una herramienta que permite crear entornos de construcción y

disponíamos de un tutorial para aprender a utilizarlo. Pese a que inicialmente parecía una buena opción ya que pertenece a Atlassian y permitiría una buena integración con Jira, resultó ser una aplicación muy cara y, aunque cuenta con una prueba gratis de 30 días, la duración de nuestro proyecto es mayor que ese periodo y por este motivo fue descartada.

La segunda herramienta probada fue Jenkins, esta permite la automatización de procesos facilitando la integración contínua la cual nos permite ejecutar proyectos y secuencias de comandos en linux. Varios puntos a favor que encontramos fueron su capacidad de extender su funcionalidad mediante plugins, también permite mirar archivos internos que contienen configuraciones hechas ahorrándonos el volver a crearlas, además al ser una herramienta tan conocida existen muchos foros y documentación sobre Jenkins, esto nos resulta beneficioso a la hora de resolver los distintos problemas que puedan surgir durante la integración continua. Por otro lado encontramos varios inconvenientes como son la dificultad a la hora de usar la interfaz, es muy complicada de instalar y configurar y existe una alta probabilidad de que se rompa debido a algún cambio de configuración.

Posteriormente, probamos tanto TRAVIS CI como CIRCLECI. Como punto positivo, descubrimos que ambas herramientas se vinculan con GitHub lo cual nos vendría bien para el desarrollo de nuestro proyecto, pero rápidamente encontramos varios inconvenientes ya que ambas requerían que aprendieramos un nuevo idioma, eran difíciles de configurar y además TRAVIS CI no tiene versión gratuita.

Por último, la herramienta elegida es Github Actions. La hemos elegido porque consideramos que es más óptimo trabajar directamente con GitHub que es donde se encuentra nuestra repositorio, evitando la instalación y aprendizaje de programas externos. Al trabajar con GitHub Actions el fichero de configuración se encuentra alojado en el repositorio y la configuración nos pareció más intuitiva y fácil que otros programas ya que hemos utilizado anteriormente GitHub y tenemos más experiencia.

3.2. Pipeline de integración continua

Para nuestro proyecto vamos a configurar el pipeline para que pueda realizar distintas pruebas (ya sean test unitarios o de integración, que son los test que están automatizados) cada vez que se realice una acción en la rama main de nuestro repositorio de github.

Para considerar que las pruebas sean correctas tienen que ejecutar sin problemas una serie de pasos descritos en el fichero .yml, un fichero que sirve para configurar los *workflow* (en este caso usamos uno específico para el

GitHub Actions) y que se iniciará cada vez que se haga un pull en la rama de main.

El primer paso para considerarse correcto es comprobar que podemos arrancar una máquina virtual (Ubuntu) para poder simular la aplicación. Una vez que hemos arrancado la máquina, comprobamos si se pueden compilar los distintos ficheros de código subidos y cambiamos el lenguaje para poder ejecutar nuestra aplicación. Por último, se probará que existen los ficheros necesarios para hacer las pruebas automáticas.

El pipeline trabajará con los test que previamente se han creado para las distintas funcionalidades. La idea principal de nuestro pipeline es tenerlo configurado en un inicio y poder ir añadiendo más tests cada vez que se crean. De esta manera, cuando una funcionalidad se considere terminada y haya pasado los test correspondientes, estos se añadirán al pipeline, haciendo así que cada vez que se haga un cambio del proyecto en la rama main, se estará comprobando que estos cambios no se hayan propagado a otras clases.

3.3. Hacer prueba piloto de la integración continua

Para la prueba piloto que se ha realizado, se ha habilitado la herramienta de actions dentro de nuestro repositorio de GitHub para que cada vez que se realice un push o un pull-request se realicen las acciones que especifiquemos en el fichero yml. En la prueba realizada, además de preparar el entorno para compilar el proyecto, se realiza una acción que ejecutará los distintos test mediante el comando *./gradlew test* tal como muestra la imagen 3.3.1.

```yaml
name: Android Build

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: set up JDK 11
      uses: actions/setup-java@v2
      with:
        java-version: '11'
        distribution: 'temurin'
        cache: gradle

    - name: Checkout the code
      uses: actions/checkout@v2

    - name: Grant execute permission for gradlew
      run: chmod +x gradlew

    - name: Run Tests
      run: ./gradlew test

    - name: Upload test report
      uses: actions/upload-artifact@v2
      with:
       name: unit_test_report
       path: app/build/reports/tests/testDebugUnitTest/
```

Imagen 3.3.1 “Código de configuración de Github Actions”

Tras haber realizado el push o pull-request sobre el main, podremos dirigirnos a la pestaña de “actions” dentro del repositorio para ver si ha habido algún error. Si todo ha ido bien podremos ver un tick verde al lado del nombre de la acción realizada, por ejemplo, en la imágen 3.3.2 podemos observar un ejemplo de un update en el fichero android.yml (fichero que se usa para la configuración del GitHub actions), tiene un “tick” verde indicando que se ha realizado sin errores y, además, muestra cada una de las fases que se han realizado (set up, build, run…), esto permite que, en caso de que haya un error, se pueda localizar en qué fase específica ha ocurrido y solucionarlo de manera más rápida.

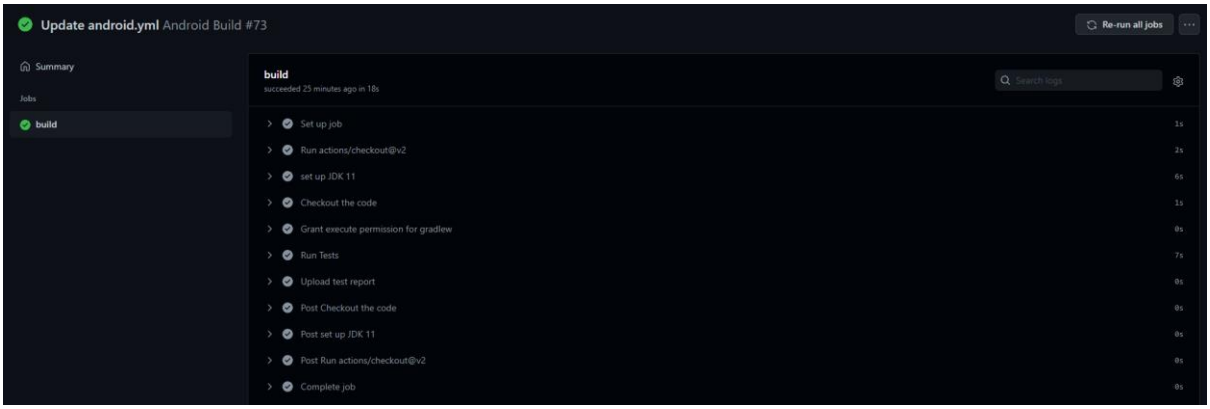


Imagen 3.3.2 “Código de ejecución de Github Actions y pruebas para comprobar su funcionamiento”

4. Definición de Hecho (DOD)

En este proyecto consideramos *Definición de Hecho* o *Definition of Done* (DOD) a los criterios mínimos necesarios que necesita cumplir una Historia de Usuario (UH) para poder ser movida a la columna de DONE.

El primer criterio que consideramos imprescindible para que una UH se pueda dar por hecha es que **cumpla con sus criterios de aceptación**, ya que estos definen las funcionalidades que el Product Owner (PO) considera básicas para la UH en cuestión. Este siguiente criterio está estrechamente relacionado con el último ya que, si los criterios de aceptación definen qué debería hacer una UH en la mente del PO y, de estar perfectamente hechos, no sería necesario nada más para que la visión del PO y la realidad de la UH implementada coincidieran, siempre puede haber algún aspecto que no se tuviera en consideración a la hora de definir los criterios de aceptación, por lo que para considerar una UH como hecha es extremadamente importante que **pase los tests de aceptación** y, de esta forma, cualquier aspecto no contemplado por el PO se pueda transmitir al equipo de desarrollo.

Otro aspecto que consideramos imprescindible para dar una UH por terminada es **hacer *pull-request* a la rama dev y que este sea aceptado**, ya que esto ocurrirá inmediatamente antes de que el PO valide el producto y para que esa *pull-request* sea aceptada, el trabajo del miembro del equipo de desarrollo deberá cumplir los criterios abajo enumerados.

Entrando en consideraciones más técnicas y de menor granularidad, una UH no puede llegar a ser validada por el PO si, para ciertos inputs del usuario, el programa no responde de la forma prevista o, aún peor, *crashea* . Es por esto que consideramos que otro criterio de imprescindible cumplimiento para que una UH se pueda dar por terminada es que **pase los tests automatizados**

(unitarios y de integración) **y los de sistema** ya que de esta forma nos aseguramos de que los elementos del software del proyecto actúan y se comunican entre sí de la forma que esperamos.

Para asegurar la mayor calidad del software que el equipo pueda entregar, de forma que los cambios que se puedan realizar en el mismo de una *release* a otra conlleven el menor grado de complicación posible, **el código que cada miembro del equipo de desarrollo realice deberá estar comentado y será revisado por al menos un miembro más del equipo** cuando el miembro que hay escrito el código considere necesario, pero al menos una vez y antes de subir el código a la rama dev para llegar a considerar la UH como hecha.

5. Product Backlog

En esta práctica nos hemos dedicado a repasar todas las historias que teníamos anteriormente y hemos quitado aquellas que empezaban por “ir a modo”, puesto que se decidió que no eran necesarias. Además el PO y el SM han corregido todo el resto de historias según la indicación de la corrección del PB entregado la práctica anterior, se han mejorado las descripciones y se han cambiado algunos criterios de aceptación para hacerlos más precisos y acordes con el resto.

No hemos agregado nuevas historias porque no hemos considerado que por ahora está como queremos, pero no descartamos que en el futuro modifiquemos algunas y agreguemos nuevas puesto que nuestra aplicación sigue en constante evolución.

6. Actividad Grupal

En las actividades propuestas en clase y en los laboratorios, nos hemos dado cuenta de que una de nuestras debilidades del equipo ahora mismo es la falta de importancia que tienen para nosotros las pruebas software. Por eso mismo la actividad grupal de esta semana consiste en aprender lo importante que es esta parte en el desarrollo de un proyecto de software.

6.1. Descripción de la Actividad

Para la realización de esta práctica, cada miembro del grupo buscó en internet un desastre de una empresa relacionado con el software, los costes de ese desastre y la manera en la que se podría haber evitado. Más adelante se pusieron en común todas estas historias con la intención de concienciarnos sobre la importancia de las pruebas.

6.2. Desarrollo

Para la realización de la práctica, pusimos una fecha de reunión para hacer la actividad en grupo y nos conectamos todos en “Kumo Space”. En un orden preestablecido antes de empezar la grabación, cada uno fue contándonos su historia y sus ideas al respecto en un intervalo de dos a cinco minutos, luego se explicó una manera sencilla con la que se habría evitado el problema y más adelante se escribieron resúmenes de cada una en un documento de google drive. Por último, el Scrum Master hizo un breve resumen sintetizando la idea general.

link del video:

https://drive.google.com/file/d/1m7JQtz-wOKND3n-gzXoqIJs9QEcHIz5P/view?usp=sharing

link del documento:

https://docs.google.com/document/d/1iWPa70WDwgsRSpKp-MZnwnuNADDfAROK70oUNLy7idw/edit?usp=sharing

6.3. Resultados

El equipo ha aprendido la importancia de las pruebas en todos los proyectos, sin importar la relevancia o magnitud de este. Viendo los ejemplos de fallos de software se ha puesto en evidencia que hasta el error más pequeño puede hacer que un programa entero quede completamente destrozado o que una empresa pierda cantidades excesivas de dinero por su culpa.

Desde el punto de vista del Scrum Master, la actividad ha cumplido su objetivo puesto que ahora todo el equipo está mucho más volcado en esas pruebas.