Universidad Zaragoza

Trabajo Fin de Grado

Grado en Ingeniería Informática

Automatización de pruebas de aceptación de usuario para aplicaciones móviles desarrolladas en Android

Autora

Ginger Janet Valencia Vásconez

Directores

Javier Nogueras Iso

Miguel Ángel Latre Abadía

Universidad de Zaragoza / Escuela de Ingeniería y Arquitectura

2017

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

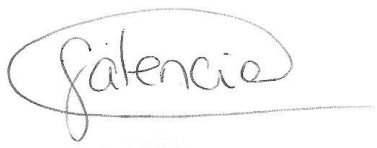
D./Dª. GINGER JANET VALENCIA VÁSCONEZ,

con nº de DNI 26589479F en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Automatización de pruebas de aceptación de usuario para aplicaciones móviles desarrolladas en Android,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, a 3 de febrero de 2017



Fdo: Ginger Janet Valencia Vásconez

Resumen

Este Trabajo Fin de Grado consiste en el estudio y desarrollo de un prototipo de componente para automatizar pruebas de aceptación de usuario en aplicaciones desarrolladas en Android a partir de diagramas representativos de la aplicación a probar, tales como diagramas de actividad a nivel de análisis o diagramas de navegación de la aplicación.

Actualmente es posible realizar pruebas de usuario completas utilizando distintas herramientas, las cuales necesitan una preparación previa por parte de los programadores/usuarios de pruebas. Esta preparación suele consistir en el desarrollo de descripciones del comportamiento de la aplicación, realización de diversos diagramas y programación de código extra, lo cual resulta en una inversión elevada de tiempo y recursos.

Este TFG tiene como objetivo estudiar la manera de unificar y automatizar algunos de los diversos métodos existentes para facilitar así la realización de pruebas. Para ello, se han seleccionado distintas herramientas y se han enlazado con ayuda de un módulo desarrollado para tal fin, obteniendo como resultado la simplificación del proceso de preparación previo a la realización de las pruebas. Como entrada es necesario tan solo un diagrama y como salida se obtiene una implementación parcial del código necesario para ejecutar dichas pruebas.

Para la implementación se ha utilizado: Java 8 SE como lenguaje de programación, Android Studio como entorno de programación; Gradle como herramienta de construcción y gestión de proyectos en Android Studio; GraphWalker y YEd Graph Editor para el diseño automático de los casos de prueba; Cucumber y Gherkin como entornos de automatización de la ejecución de pruebas; y Espresso, e implícitamente JUnit, como tecnología para automatizar las pruebas que actúan sobre el interfaz gráfico de usuario de las aplicaciones móviles.

Agradecimientos

A a mi madre Janet, por darme la oportunidad de poder comenzar una carrera universitaria, por el apoyo ofrecido durante todos estos años y por ser para mí un ejemplo de superación personal, demostrándome que las recompensas se obtienen con trabajo y esfuerzo.

A Antonio, por apoyarme, motivarme y aguantarme cada día. Por estar siempre ahí, tanto en los buenos como en los malos momentos. Por confiar siempre en mí.

A mi hermana Génesis, a mis amigos y compañeros, por animarme y apoyarme continuamente.

Y, por supuesto, a los directores del proyecto, Javier Nogueras y Miguel Ángel Latre, por permitirme realizar este proyecto, guiándome en todo momento con paciencia y dedicación.

Índice

1. Introducción

1.1. Contexto y motivación

Las pruebas son una de las fases del ciclo de vida del *software* más importante ya que están orientadas a garantizar que un sistema cumple con los requisitos establecidos.

El proceso de pruebas de *software* [11] es un proceso mediante el cual se aplican una serie de métodos (ocasionalmente utilizando herramientas) que permiten obtener un conjunto de medidas para verificar y validar el funcionamiento requerido del software. Las pruebas pueden ser ejecutadas por cualquiera de las partes implicadas en el desarrollo de *software*, ya sea la parte proveedora (por ejemplo, la empresa que lo desarrolla), la cual se asegura de que el producto entregado cumple con todos los requerimientos necesarios, o la parte aceptante (cliente o usuario para el cual se realiza el desarrollo de *software*), la cual se asegura de que el producto recibido es lo que se había solicitado. Podemos distinguir distintos niveles de prueba, asociados a las responsabilidades de cada una de las partes. Así, se definen por la parte proveedora las pruebas de desarrollo, que se aseguran del cumplimiento de especificaciones técnicas y las pruebas de sistema, que comprueban el cumplimiento de requisitos y de diseño técnico. Por la parte aceptante tenemos las pruebas de aceptación, las cuales verifican que el producto cumple con las expectativas.

Aunque se ha avanzado en la generación automática de casos de prueba y la ejecución de los mismos, este grado de automatismo es menor en pruebas de aceptación de usuario, ya que las plataformas y herramientas existentes para la automatización de pruebas normalmente están pensadas para ser utilizadas por desarrolladores con avanzados conocimientos en programación e ingeniería del *software*. Por esta razón, este TFG se presenta no solo como una recopilación del trabajo realizado sino también como el prototipo de un posible componente que facilite el trabajo de programadores o encargados de realizar pruebas de aceptación de usuario que no posean amplios conocimientos informáticos.

En particular, en este Trabajo Fin de Grado ha realizado el estudio y aplicación de las tecnologías existentes en automatización de pruebas de aceptación de usuario para aplicaciones móviles en Android [1] a partir de diagramas de estados de la aplicación objeto de pruebas. Dichos diagramas

de estados se utilizan para modelar la navegación de la aplicación, los cuales muestran gráficamente el flujo y comportamiento de la aplicación dependiendo de las acciones realizadas por el usuario.

1.2. Objetivos

El objetivo principal de este TFG es crear un componente que automatice, en la mayor medida posible, no solo la realización de pruebas de aceptación de usuario de aplicaciones móviles desarrolladas en Android sino también el diseño de estas pruebas.

El primer planteamiento del proyecto, antes de estudiar más en detalle herramientas o aplicaciones existentes, se basaba en el uso de la tecnología Cucumber [7] para la definición de escenarios de prueba a partir de descripciones en texto plano (lenguaje Gherkin [8]) de las funcionalidades principales de la aplicación objeto de pruebas.

Tras el estudio de diversas herramientas y tecnologías existentes, se han seleccionado tres de ellas para complementar el uso de la herramienta planteada anteriormente, pudiendo abordar de esa manera un mayor número de tecnologías existentes y obtener así una variedad de elementos que podrán ser utilizados individualmente como datos de entrada de las tecnologías y herramientas de pruebas estudiadas. El estudio y análisis de dichas herramientas se encuentra recogido en el 0.

Otro objetivo de este TFG es reducir el tiempo, los conocimientos técnicos avanzados y los recursos empleados por parte de los usuarios para el proceso de pruebas, facilitando la ejecución de estas lo máximo posible. El componente desarrollado, de nombre **Graph2Test Component** (Graph2Test en adelante) permite obtener datos de entrada específicos para varias herramientas de pruebas además de generar el resultado automáticamente, evitando así el desarrollo total de los casos de pruebas por parte del usuario que ejecuta las pruebas.

1.3. Tecnología utilizada

Se ha utilizado como lenguaje de programación base la tecnología Java en su versión 8 [2], con Android Studio [3] como entorno de programación, ya que la finalidad de Graph2Test es realizar pruebas automáticas para aplicaciones desarrolladas en Android y, de esta manera, poder utilizarlo como un módulo dentro de una aplicación Android.

Se ha utilizado Gradle [4] como herramienta *software* para la construcción y gestión interna del proyecto, ya que dicha herramienta viene integrada en Android Studio y resulta fácil de utilizar una vez se han comprendido los detalles técnicos básicos. Esta herramienta se encarga de la automatización y administración del proceso de compilación, además de permitir definir configuraciones de compilación personalizadas y flexibles. Utiliza ficheros de tipo `gradle` para la configuración, de los cuales, dentro de un proyecto, hay uno a nivel general y uno por cada módulo dentro del proyecto para permitir realizar configuraciones particulares entre distintos proyectos.

Para la creación de los diagramas de estados que representan la navegación de las aplicaciones objeto de pruebas se ha utilizado YEd Graph Editor [6], cuyos diagramas sirven de entrada para GraphWalker [5], herramienta utilizada para realizar un recorrido de los estados que aparecen en el diagrama del diagrama y obtener así secuencias de ejecución de la aplicación objeto de pruebas.

Graph2Test utiliza también la tecnología Cucumber, la cual permite obtener un esqueleto de pruebas (código) a partir de una definición de los requisitos de la aplicación en lenguaje mínimamente controlado (lenguaje Gherkin).

Se ha utilizado Espresso [9] como framework para la codificación de las pruebas, el cual simula una interacción de usuario con la aplicación Android, además de ser fácilmente configurable en la aplicación haciendo uso de Gradle. También se ha hecho uso de JUnit [10] para codificar las pruebas, el cual provee de métodos complementarios a los ofrecidos por Espresso. La metodología utilizada para el desarrollo de Graph2Test y su arquitectura se encuentra recogida el capítulo 3.1.

1.4. Estructura de la memoria

En primer lugar, se detalla en el capítulo 2 en qué consiste el proceso de pruebas de *software*, las distintas técnicas que hay para enfocar las pruebas, los problemas asociados a estas y, en concreto, el diseño de casos de pruebas de aceptación.

En el capítulo 3 se presenta la arquitectura de Graph2Test Component, indicando las tareas que realiza, las herramientas utilizadas y la manera en las aplica para el diseño y automatización de pruebas de aceptación de usuario.

En el capítulo 4 se muestran las aplicaciones Android utilizadas como casos de estudio y los resultados obtenidos para cada una de ellas.

Finalmente, en el capítulo 5 se resumen las conclusiones y opinión personal sobre la realización del trabajo, además de indicar el trabajo futuro sobre este proyecto.

Se dispone también de unos anexos al final de la memoria, en donde queda reflejado el análisis de las tecnologías estudiadas, una explicación de la sintaxis básica de Graph2Test y los resultados completos de los casos de prueba descritos en el capítulo 4.

2. Diseño de casos de prueba de aceptación

El proceso de pruebas de *software* es una actividad que forma parte del control de calidad, consistente en utilizar herramientas y técnicas para proporcionar información objetiva e independiente sobre la calidad del producto desarrollado. Dependiendo del enfoque de las pruebas podemos distinguir entre técnicas de caja blanca y caja negra. Las técnicas de caja negra son aquellas basadas en la especificación del objeto de pruebas y mediante ellas es posible verificar que el comportamiento del objeto satisface un conjunto de criterios o requisitos seleccionados. Las técnicas de caja blanca se basan en la estructura del objeto de pruebas (normalmente, su código fuente), y examinan los distintos flujos de ejecución del programa.

En las fases de análisis y diseño de la aplicación uno de los elementos principales que se acuerdan entre el desarrollador y el cliente son los prototipos de la interfaz y la manera en la que se navegará por ella. Por ese motivo, los mapas de navegación son un elemento clave para asegurar que todos los requisitos establecidos con el cliente se han cumplido al final del desarrollo.

Para representar dichos mapas de navegación en este TFG, se ha optado por el uso de diagramas de estado y utilizando la técnica transición de estados (*state transition testing* [25]), con un nivel de cobertura de estados o de transiciones. El objetivo de estas pruebas es utilizar un algoritmo de recorrido de grafos para generar un conjunto de casos de prueba que cubran todas las transiciones o todos estados del diagrama, de acuerdo con el nivel de cobertura elegido.

Este TFG se centra en las pruebas de aceptación de usuario, las cuales son un tipo de ensayos cuya finalidad es verificar si el producto desarrollado cumple con todos los requisitos establecidos por el cliente. Se clasifican dentro de la metodología de pruebas funcionales, con enfoque de pruebas de caja negra. El trabajo se centra en la investigación de la automatización de las pruebas de aceptación de usuario a partir de la generación de mapas de navegación (representados mediante grafos) del software a probar, pasando por la definición de los requisitos en texto plano mínimamente controlado y obteniendo una generación parcial del código de pruebas.

3. Infraestructura para el diseño y automatización de pruebas

3.1. Arquitectura

En la Figura 1 se muestra el flujo de trabajo propuesto para la automatización del diseño y ejecución de las pruebas de aceptación utilizando Graph2Test, un componente desarrollado para realizar pruebas automáticas en Android. Para utilizar Graph2Test es necesario disponer de un diagrama de estados de la aplicación que se desea probar. Este diagrama de estados se crea utilizando la aplicación YEd Graph Editor siguiendo la estructura predefinida que se encuentra descrita en el Anexo A.2 y, utilizando dicho diagrama y la herramienta GraphWalker, se recorre el diagrama de estados y se obtienen los escenarios de Cucumber, tomando como referencia los textos de cada arista y cada nodo del diagrama.En el diagrama de flujo están marcados en amarillo los pasos a seguir por el usuario que ejecuta las pruebas y los pasos en azul indican la secuencia de ejecución de Graph2Test.

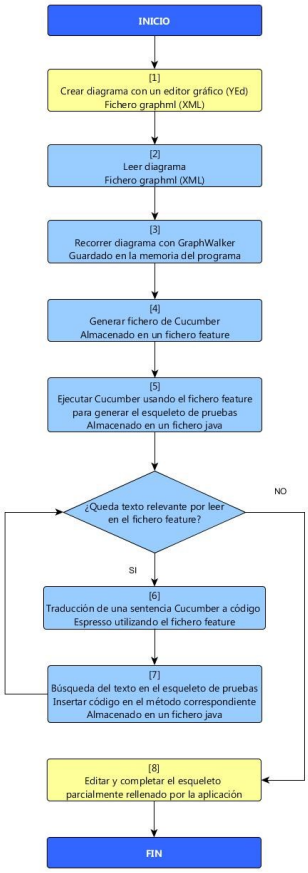


Figura 1. Diagrama de flujo de trabajo para el diseño y automatización de pruebas

Una vez obtenidos los escenarios, se ejecutan los mismos lanzando la herramienta Cucumber dando como resultado el esqueleto de las pruebas (*steps*). Para hacer una implementación parcial a partir del texto de los escenarios se utiliza la herramienta Espresso. Se rellena el esqueleto obtenido con Cucumber automáticamente con los datos disponibles obtenidos del diagrama creado con YEd Graph Editor y realizando una traducción de texto plano a instrucciones de Espresso (o el código Java oportuno) y dicho esqueleto debe ser modificado por el usuario para completarlo. Es posible también empezar el análisis desde los escenarios, partiendo desde el nodo 5 que aparece en la Figura 1, y utilizando un fichero `feature` escrito en lenguaje Gherkin, el cual permite describir el comportamiento del software utilizando para ello texto plano. Se puede proceder sólo al rellenado del esqueleto partiendo desde el nodo 6 pero, como se explica en el Anexo B.4, es necesario disponer de un esqueleto de código almacenado en un fichero `java` y los escenarios asociados a este en un archivo `feature`. Más adelante, en el capítulo 3.3 se dará una breve introducción al lenguaje Gherkin y se detallará el proceso de traducción de los textos en cada elemento del diagrama a escenarios de Cucumber. Los detalles referentes a la traducción de los escenarios de Cucumber a esqueleto de pruebas y código Espresso aparecen recogidos en el capítulo 3.4.

A continuación, se mostrará la arquitectura de Graph2Test, seguido de una explicación más detallada de los pasos seguidos para la obtención automática del código de pruebas.

En la Figura 2 se muestra el diagrama de despliegue donde se puede ver una vista física de la utilización de la infraestructura. Por un lado, se tiene el Graph2Test, el cual utiliza las herramientas Cucumber, GraphWalker y Espresso y las unifica en el artefacto automatizacionPruebas. Graph2Test necesita un diagrama de estados como punto de partida, el cual puede ser creado por el usuario haciendo uso del programa YEd Graph Editor.

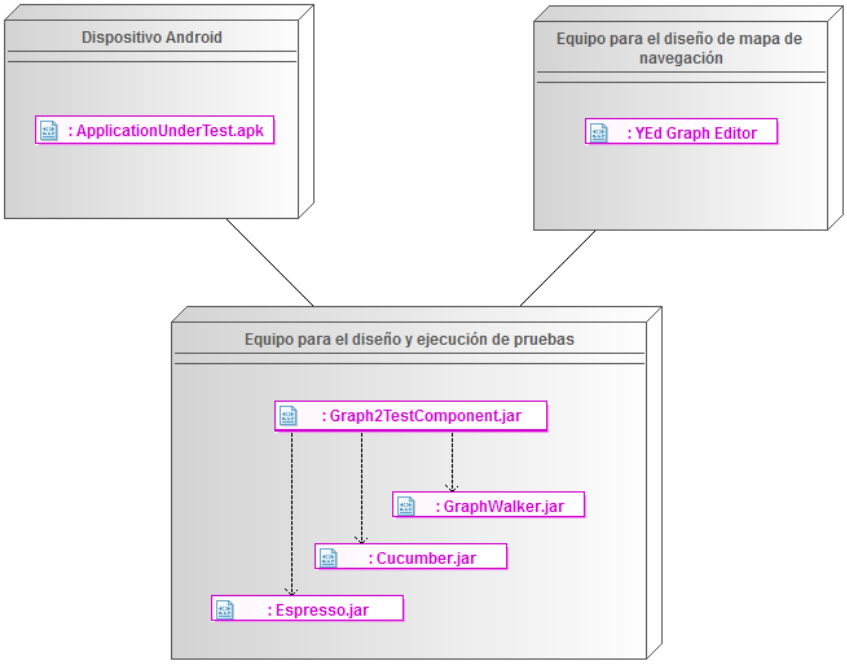


Figura 2. Diagrama de despliegue

En la Figura 3 se muestra la división en subsistemas del *software* que se ha desarrollado. El paquete `commons` contiene las clases principales de Graph2Test. En este paquete se encuentra la clase `DataManager`, encargada de hacer de enlace entre las distintas funcionalidades de Graph2Test y la clase `FileManager` se encarga de la gestión de los ficheros de lectura y escritura. El paquete `graphWalker` almacena las clases necesarias para la gestión del diagrama, tanto para lectura del fichero y ejecución de la herramienta como para el almacenamiento de los elementos del diagrama en memoria. En el paquete `cucumber` tenemos las clases para la gestión de las palabras clave del lenguaje Gherkin. En el paquete `espresso` se encuentran las clases necesarias para la gestión de las palabras clave de Espresso. Finalmente, el paquete `translator` contiene la información necesaria para realizar el emparejamiento entre palabras en texto plano (Gherkin) y palabras clave de Espresso (código). Se explicarán en profundidad cada una de estas partes en los apartados que se muestran a continuación.

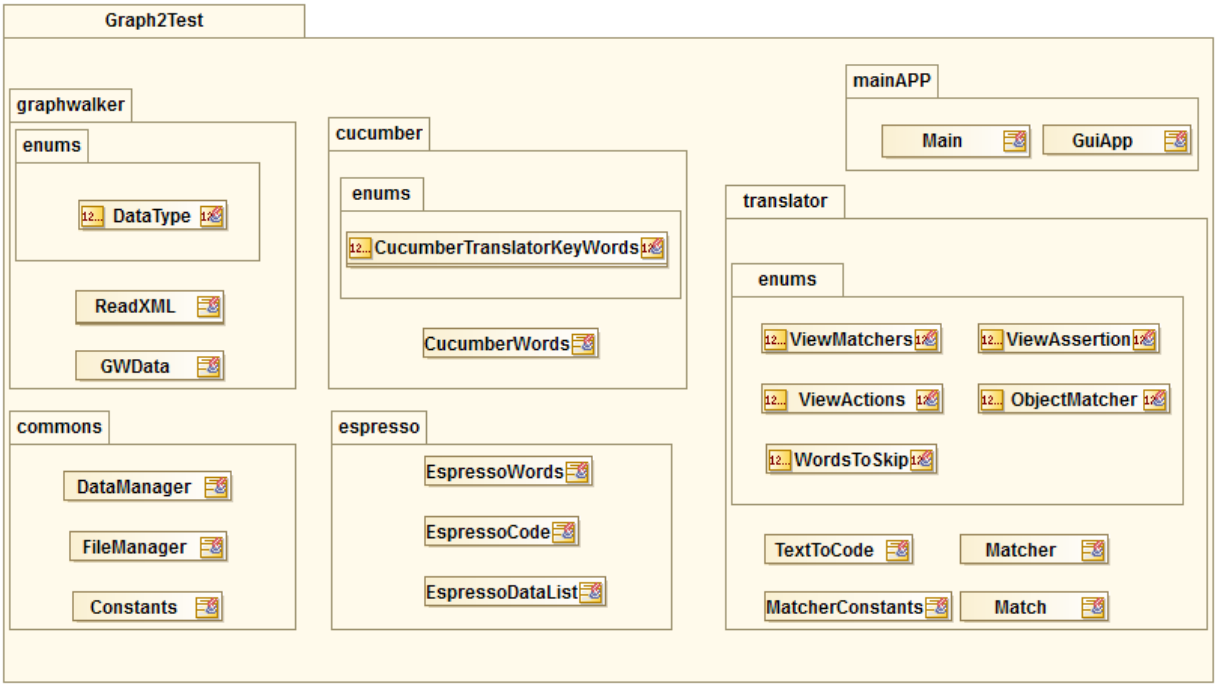


Figura 3. Diagrama de paquetes de Graph2Test

3.2. Recorrido del diagrama con GraphWalker

El primer paso a realizar por parte del usuario es crear un diagrama de estados del *software* objeto de pruebas utilizando el programa YEd Graph Editor. Es recomendable usar dicho programa de diseño de diagramas ya que es el que se utiliza en los ejemplos ofrecidos en la página oficial de GraphWalker.

Para modelar los diagramas con YEd y usarlos posteriormente con GraphWalker es necesario seguir una sintaxis básica (que se encuentra explicada en el Anexo B), que se resume que el modelo debe ser un grafo dirigido, el texto de los nombres de los elementos (aristas/nodos) debe ser una misma palabra (o varias separadas por "_") y se pueden realizar referencias desde un modelo a otro, pudiendo simplificar así la visualización y el diseño de un diagrama de dimensiones considerables.

El resultado es un archivo de extensión `graphml` (Graph Markup Languaje), el cual es básicamente un fichero `XML` que almacena los datos referentes a las distintas aristas/nodos y la relación entre ellos. A continuación, se pondrá como ejemplo el caso de una aplicación que permita pulsar un botón con el texto 0 o 1 y que lo muestre en pantalla. En la Figura 4 se muestra el diagrama realizado con YEd Graph Editor y en la Figura 5 se muestra el

mismo diagrama visualizado como texto XML, simplificado para mostrar sólo los campos relevantes en este TFG.

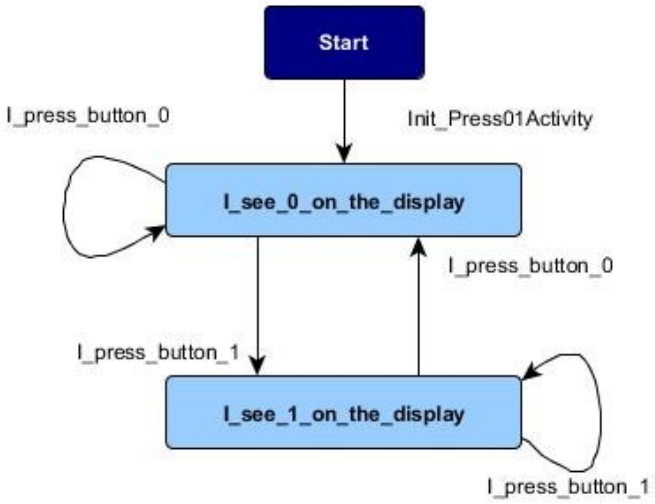


Figura 4. Diagrama de estados visualizado con YEd Graph Editor (fichero graphml)

```
1: <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2: <graphml [...] >
3:     [...]
4:     <graph edgedefault="directed" id="G">
5:         <node id="n0">
6:             <y:NodeLabel>Start</y:NodeLabel> [...]
7:         </node>
8:         <node id="n1">
9:             <y:NodeLabel>I_see_0_on_the_display</y:NodeLabel> [...]
10:        </node>
11:        <node id="n2">
12:            <y:NodeLabel>I_see_1_on_the_display</y:NodeLabel> [...]
13:        </node>
14:        < edge id="e0" source="n0" target="n1">
15:            <y:NodeLabel>Init_Press01Activity</y:NodeLabel> [...]
16:        </edge>
17:        < edge id="e1" source="n1" target="n2">
18:            <y:EdgeLabel>I_press_button_1</y:EdgeLabel> [...]
19:        </edge>
```

```
20:         <edge id="e2" source="n2" target="n1">
21:             <y:EdgeLabel>I_press_button_0</y:EdgeLabel> [...]
22:         </edge>
23:         <edge id="e3" source="n2" target="n2">
24:             <y:EdgeLabel> I_press_button_1</y:EdgeLabel> [...]
25:         </edge>
26:         <edge id="e4" source="n1" target="n1">
27:             <y:EdgeLabel>I_press_button_0</y:EdgeLabel> [...]
28:         </edge>
29:     </graph>
30: </graphml>
```

Figura 5. Diagrama de estados simplificado y visualizado con un bloc de notas (fichero graphml)

Utilizando dicho diagrama, desde Graph2Test se realiza una invocación a GraphWalker, al cual hay que indicarle la ruta del diagrama, el elemento inicial, el tipo de recorrido a realizar y la cobertura deseada. En el uso de GraphWalker como parte de Graph2Test, el recorrido se realiza hasta alcanzar una cobertura de todos las las aristas del grafo.

Si el diagrama ha sido creado correctamente, la ejecución de GraphWalker terminará dando como resultado una lista de las aristas y nodos en el orden en que han sido visitados. En caso contrario, GraphWalker entrará en un bucle infinito. En el Anexo A.2 se explica más en profundidad las opciones ofrecidas por YEd Graph Editor y GraphWalker para la creación, edición y recorrido de grafos.

Utilizando un diagrama como el que aparece en la Figura 4, se lee dicho fichero en formato XML utilizando la librería SAX de Java [12][13] y se realiza la llamada a GraphWalker indicándole el nombre del elemento inicial (Init_Press01Activity en el ejemplo) y solicitando una cobertura del 100% de las aristas. Se obtiene como resultado de la lectura del fichero XML la lista de aristas y nodos que aparecen en la Tabla 1 y del recorrido con GraphWalker la secuencia de aristas y nodos visitados que aparece en la 17Figura 6.

CLAVE	TIPO	FRASE

I_see_0_on_the_display	NODE	I see 0 on the display
I_see_1_on_the_display	NODE	I see 1 on the display
I_press_button_0	EDGE	I press button 0
I_press_button_1	EDGE	I press button 1
Init_Press01Activity	EDGE	Init Press01Activity
Start	NODE	Start

Tabla 1. Lista de aristas/nodos almacenada en memoria

```
1:   {"currentElementName":"Init_Press01Activity"}
2:   {"currentElementName":"I_see_0_on_the_display"}
3:   {"currentElementName":"I_press_button_0"}
4:   {"currentElementName":"I_see_0_on_the_display"}
5:   {"currentElementName":"I_press_button_1"}
6:   {"currentElementName":"I_see_1_on_the_display"}
7:   {"currentElementName":"I_press_button_1"}
8:   {"currentElementName":"I_see_1_on_the_display"}
9:   {"currentElementName":"I_press_button_0"}
10:  {"currentElementName":"I_see_0_on_the_display"} ">
```

Figura 6. Secuencia de aristas/nodos visitados con GraphWalker

A continuación, en la Figura 7 se muestra el diagrama con las clases principales de Graph2Test. La clase `DataManager` se encarga de la gestión de los métodos principales de Graph2Test, tales como generar los casos de prueba a partir de los diagramas de estado creados con GraphWalker, crear los escenarios de Cucumber o realizar la traducción de escenarios a código Espresso.

La primera tarea la realiza la clase `ReadXML`, encargada de leer el fichero XML correspondiente al diagrama de estados para posteriormente almacenar en memoria una representación del grafo mediante objetos de la clase `GWData`, la cual indica el nombre y tipo (arista o nodo) del elemento almacenado. Este paso previo al recorrido del grafo se realiza ya que, como

se puede observar en la Figura 7, GraphWalker no indica el tipo de elemento que se está visitando y ese dato es importante en el momento de realizar la creación de escenarios.

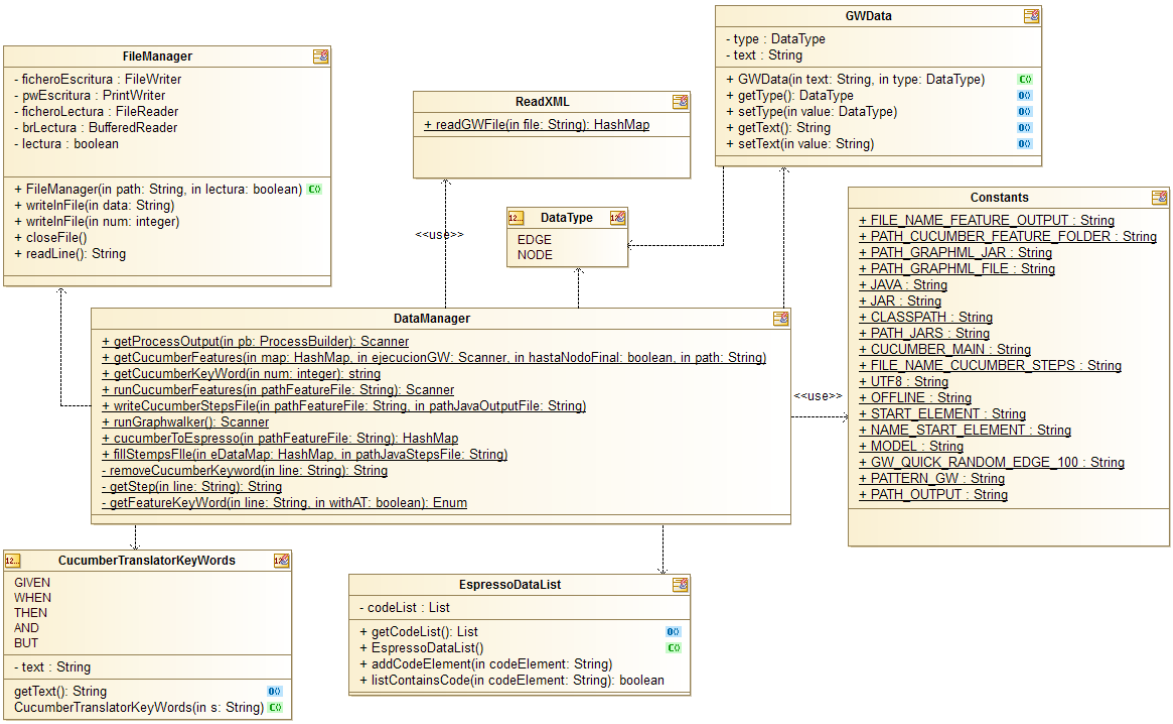


Figura 7. Diagrama de las clases principales

3.3. Traducción a escenarios de Cucumber

El resultado de la correcta ejecución de GraphWalker es una lista de aristas y nodos visitados del diagrama inicial. Utilizando dicha lista, se la recorre elemento a elemento para obtener los textos de los escenarios de Cucumber, estableciendo un paralelismo entre las aristas y nodos con las sentencias `Given-When-Then` de Cucumber.

El lenguaje Gherkin es el utilizado por Cucumber para realizar descripciones en texto del comportamiento de la aplicación. Cada requisito (caso de uso) del *software* se traduce en una característica (denominada *feature* en el lenguaje Gherkin) en la que se describe una situación determinable (escenarios, *scenarios* en lenguaje Gherkin). Cada escenario está compuesto por sentencias, las cuales describen precondiciones (sentencias *Given*), acciones realizadas por el usuario (sentencias *When*) y resultados comprobables (sentencias *Then*). En el anexo A.3 se encuentra una explicación más detallada de este lenguaje.

Los nombres de las aristas o nodos deben contener el texto que se desea que aparezca en cada paso de los escenarios de Cucumber. La generación se realiza tomando cada par de nodos unidos con una aristas (nodo-arista-nodo) y transformándolos en sentencias tipo `Given-When-Then`, tomando como texto de la sentencia `Given` el texto de la sentencia `Then` previa en el caso que sea necesario. Los escenarios se guardarán en un fichero de tipo extensión `feature` cuya ruta de carpeta será indicada por el usuario. El archivo tomará como nombre del fichero `feature` el mismo que el nombre del diagrama.

En la Figura 7 aparece el diagrama con las clases y métodos utilizados en la traducción a escenarios. En la Figura 8 se puede observar cómo la operación `getCucumberFeatures` de la clase `DataManager` permite generar los casos de prueba escritos bajo la forma de escenarios y *features* conformes con el lenguaje Gherkin, utilizando para ello los métodos y palabras clave almacenados en las clases del paquete `cucumber`.

```
1:   Feature: Add your feature here.
2:
3:
4:   Scenario: Write your Scenario here.
5:   Given Start
6:   When Init Press01Activity
7:   Then I see 0 on the display
8:
9:   Scenario: Write your Scenario here.
10:  Given I see 0 on the display
11:  When I press button 0
12:  Then I see 0 on the display
13:
14:  Scenario: Write your Scenario here.
15:  Given I see 0 on the display
16:  When I press button 1
17:  Then I see 1 on the display
18:
19:  Scenario: Write your Scenario here.
20:  Given I see 1 on the display
21:  When I press button 1
22:  Then I see 1 on the display
```

```
23:
24:  Scenario: Write your Scenario here.
25:  Given I see 1 on the display
26:  When I press button 0
27:  Then I see 0 on the display":"
```

Figura 8. Fichero feature obtenido a partir de la lectura y recorrido del diagrama

3.4. Traducción a esqueleto de pruebas y código Espresso

Una vez obtenidos los escenarios de Cucumber y utilizando el texto de cada sentencia, se procede a realizar una traducción de sentencia Cucumber a código Espresso y con eso rellenar el esqueleto obtenido al ejecutar Cucumber utilizando los escenarios generados previamente.

El esqueleto es un fichero de extensión `java` que contiene las cabeceras de los métodos que se ejecutarán al realizar las acciones Espresso. Posteriormente, el usuario debe modificar y completar el código generado. En la Figura 9 se muestra el esqueleto obtenido para el diagrama utilizado como ejemplo hasta ahora.

```
1:   You can implement missing steps with the snippets below:
2:
3:   @When("^Init PressActivity$")
4:   public void init_Press_Activity(int arg1) throws Throwable {
5:       // Write code here that turns the phrase above into concrete actions
6:       throw new PendingException();
7:   }
8:
9:   @Then("^I see (\\d+) on the display$")
10:  public void i_see_on_the_display(int arg1) throws Throwable {
11:      // Write code here that turns the phrase above into concrete actions
12:      throw new PendingException();
13:  }
14:
15:  @Given("^I see (\\d+) on the display$")
16:  public void i_see_on_the_display(int arg1) throws Throwable {
17:      // Write code here that turns the phrase above into concrete actions
18:      throw new PendingException();
19:  }
20:
21:  @When("^I press button (\\d+)$")
```

```
22:  public void i_press_button(int arg1) throws Throwable {
23:      // Write code here that turns the phrase above into concrete actions
24:      throw new PendingException();
25:  }
```

Figura 9. Esqueleto de pruebas obtenido mediante Cucumber

En la Figura 10 podemos observar la traducción realizada por Graph2Test para los casos de la Figura 8, utilizando el método `cucumberToEspresso`. Este código se almacenará en memoria hasta que se encuentren las correspondencias en el esqueleto visto en la Figura 9, procediendo entonces al rellenado del esqueleto utilizando el código generado.

```
1: onView(withId(__DATA__)).check(matches(withText(0)));
2: onView(withId(__DATA__)).check(matches(withText(1)));
3: onView(withId(__DATA__)).check(matches(withText(0)));
4: onView(withId(__DATA__)).check(matches(withText(1)));
5: onView(withText("0")).perform(click());
6: onView(withText("1")).perform(click());
```

Figura 10. Traducción realizada de escenarios Cucumber a código Espresso

Para establecer la correspondencia entre Cucumber y Espresso, se lee el fichero `feature`, tomando solo como datos relevantes las sentencias de tipo `Given-When-Then` (además de las de tipo `And` y `But`), se almacena el tipo de sentencia en memoria y se realiza una traducción siguiendo la sintaxis indicada en el Anexo B. Una vez obtenida dicha traducción, se procede a buscar en el esqueleto la correspondencia del tipo y el texto, introduciendo el texto traducido dentro del método correspondiente. En la Figura 11 se muestra el esqueleto de la Figura 9 parcialmente rellenado con el código de la Figura 10.

```
1:   You can implement missing steps with the snippets below:
2:
3:   @When("^Init PressActivity$")
4:   public void init_Press_Activity(int arg1) throws Throwable {
5:       // Write code here that turns the phrase above into concrete actions
6:       throw new PendingException();
```

```
7:   }
8:
9:   @Then("^I see (\\d+) on the display$")
10:  public void i_see_on_the_display(int arg1) throws Throwable {
11:      // Write code here that turns the phrase above into concrete actions
12:      onView(withId(__DATA__)).check(matches(withText(0)));
13:      onView(withId(__DATA__)).check(matches(withText(1)));
14:  }
15:
16:  @Given("^I see (\\d+) on the display$")
17:  public void i_see_on_the_display(int arg1) throws Throwable {
18:      // Write code here that turns the phrase above into concrete actions
19:      onView(withId(__DATA__)).check(matches(withText(0)));
20:      onView(withId(__DATA__)).check(matches(withText(1)));
21:  }
22:
23:  @When("^I press button (\\d+)$")
24:  public void i_press_button(int arg1) throws Throwable {
25:      // Write code here that turns the phrase above into concrete actions
26:      onView(withText("0")).perform(click());
27:      onView(withText("1")).perform(click());
28:  }
```

Figura 11. Esqueleto parcialmente rellenado

En la Figura 12 se pueden observar las clases que automatizan la generación de código Espresso. Para ello, se hace uso de la clase `TextToCode`, la cual hace la gestión de la traducción de texto plano (procedente de los nombres de los nodos y las aristas del diagrama de navegación) a código Espresso utilizando la clase `Matcher`. Esta clase se encarga de realizar la correspondencia entre cada palabra en el texto con el reconocimiento de las palabras clave almacenadas en las distintas clases que aparecen en la Figura 13. Ambas clases se utilizan en el método `cucumberToEspresso`, mencionado anteriormente. En el anexo A.4 se explica el proceso de traducción y las clases asociadas a la clase `Matcher`.

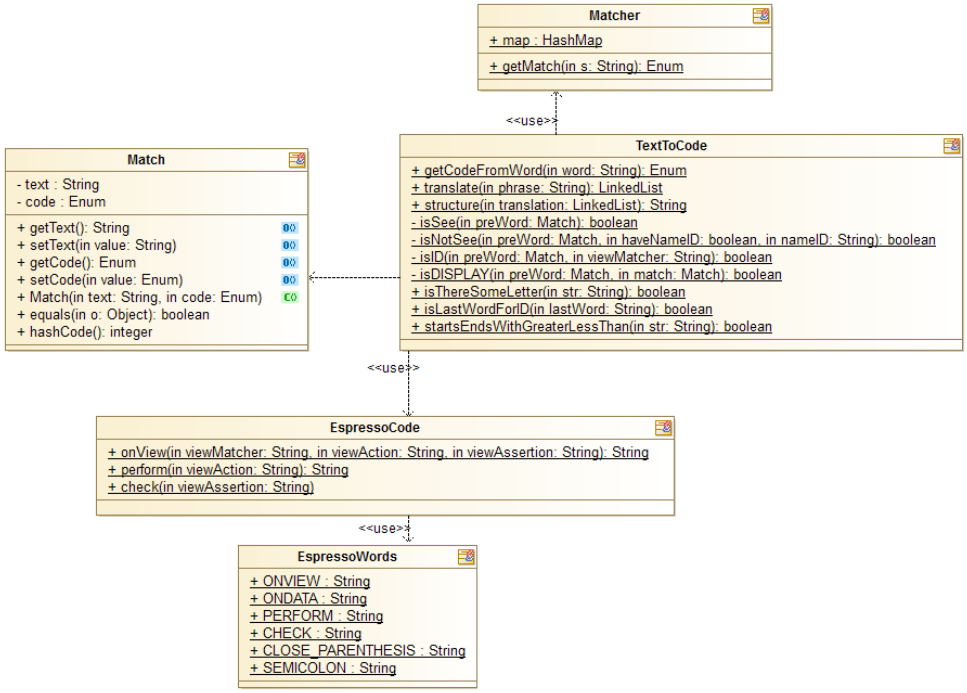


Figura 12. Diagrama de clases de paso de textos Cucumber a código Espresso

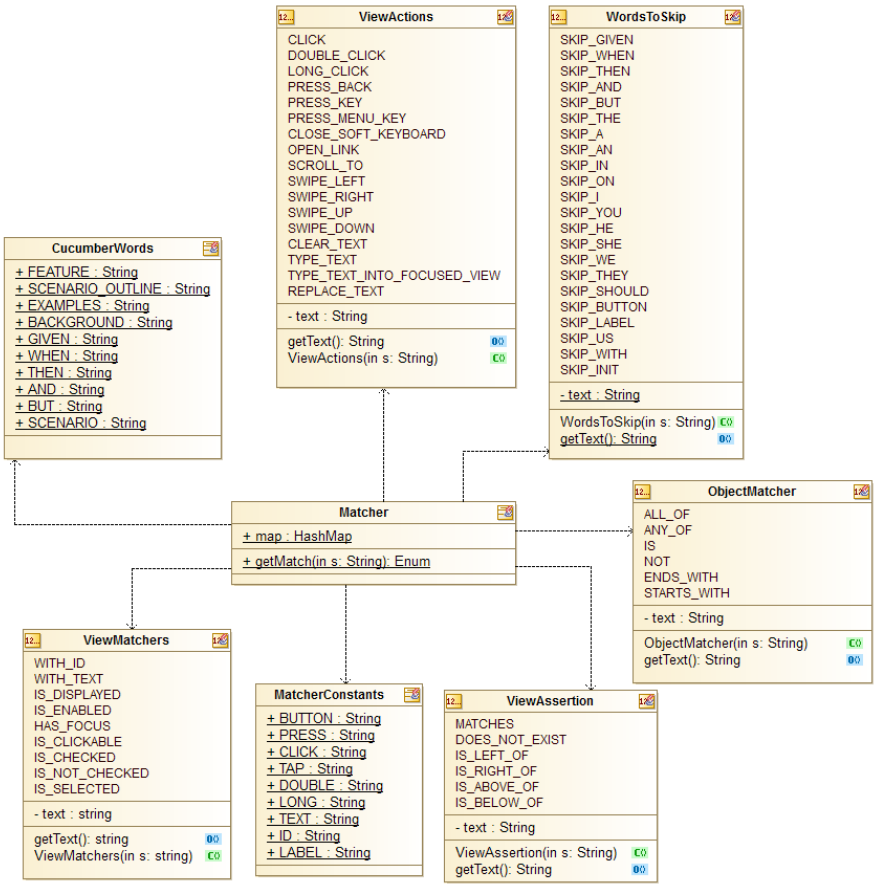


Figura 13. Diagrama de clases del matcher utilizado en el paso de texto a código

En la Tabla 2 se pueden ver ejemplos de emparejamiento. En el apartado de Espresso en el Anexo B.3 se encuentra detallada la lista de palabras clave utilizada en el proceso de traducción. Básicamente, todo lo que no es reconocido como palabra clave se traduce en una constante o variable (en el momento del reconocimiento equivale a null).

When I press num ↓ SKIP_WHEN SKIP_I CLICK (NULL)
Given I see number 5 on the display with ID displD ↓ SKIP_GIVEN SKIP_I MATCHES SKIP_NUMBER (NULL) SKIP_ON SKIP_THE WITH_ID SKIP_WITH WITH_ID (NULL)

Tabla 2. Ejemplos de traducción. Reconocimiento de palabras clave, nombres de datos y variables

4. Casos de estudio

En este capítulo se analizan un par de aplicaciones que han sido utilizadas como objeto de estudio para para comprobar la viabilidad de la infraestructura de pruebas desarrollada para este TFG. La primera es una pequeña calculadora que realiza funciones de cálculo básicas y muestra los resultados por pantalla. La segunda se trata de una aplicación que muestra en tiempo real información sobre las farmacias existentes en la ciudad de Zaragoza y su estado de apertura, cierre o guardia.

4.1. Aplicación Calculadora

La aplicación Cukeulator[1] es un proyecto de ejemplo tomado de la web de Github de Cucumber. Es una calculadora que realiza funciones de cálculo básicas, tales como sumas, restas, multiplicaciones, divisiones y raíces cuadradas. Los resultados se obtienen con decimales y también es posible seleccionar como entrada el valor del número π.

Figura 14. Aplicación Cukeulator ejecutada sobre un emulador Android

[1] https://github.com/cucumber/cucumber-jvm

A continuación, en la Tabla 3 se presenta el catálogo de requisitos funcionales de la aplicación Calculadora, los cuales han sido fundamentales para la creación del diagrama diseñado posteriormente para utilizarlo como dato de entrada de Graph2Test.

REQUISITOS FUNCIONALES	
Código	**Descripción**
RF-1	Realizar suma
RF-2	Realizar resta
RF-3	Realizar multiplicación
RF-4	Realizar división
RF-5	Calcular raíces cuadradas
RF-6	Calcular porcentajes
RF-7	Mostrar por pantalla datos insertados
RF-8	Mostrar por pantalla resultados
RF-9	Limpiar datos introducidos

Tabla 3. Tabla de requisitos de la aplicación Calculadora

El diagrama que se presenta en la Figura 15 ha sido creado utilizando YEd Graph Editor, un programa de edición de grafos recomendado en la página web oficial de GraphWalker.

Como se puede comprobar en el diagrama, solo se ha tenido en cuenta una parte de las funcionalidades de la aplicación Calculadora. Esto se debe a que esta aplicación principalmente ha sido utilizada solo para comprobar el correcto funcionamiento básico de Graph2Test, observando el comportamiento de la ejecución de las distintas partes de este y la integración de cada una de las partes desarrolladas con las herramientas empleadas (GraphWalker, Cucumber, Espresso).

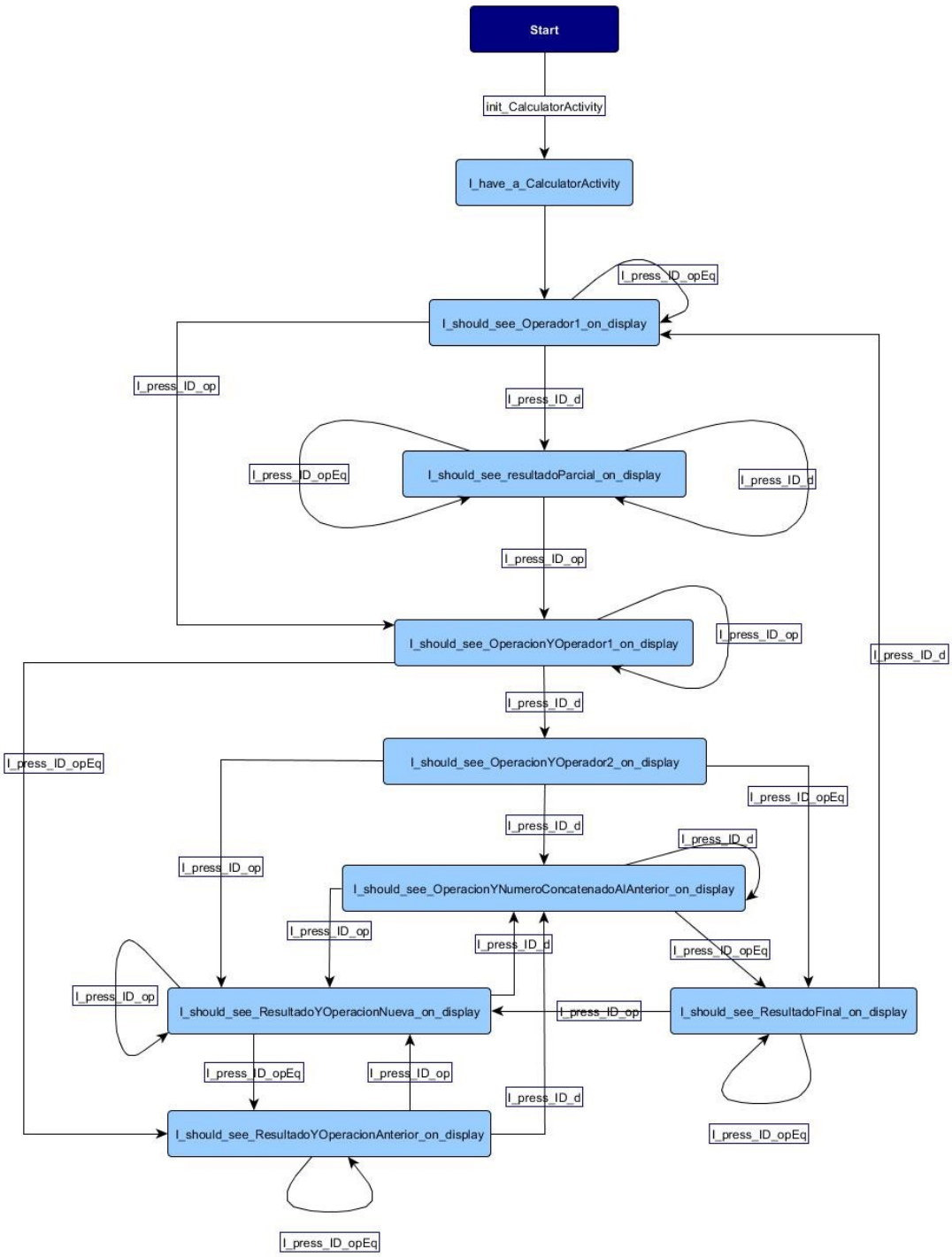


Figura 15. Diagrama de estados de la aplicación Calculadora

Para esta aplicación se han obtenido los resultados que aparecen reflejados en el Anexo C. Se han obtenido un total de 51 escenarios, de los cuales se tomará como ejemplo el que aparece en la Figura 16 para comentar los resultados obtenidos. A partir de este momento, el usuario puede realizar las

modificaciones necesarias para terminar de definir la prueba. Por ejemplo, se puede editar el fichero *feature* cambiando los escenarios normales por `Scenario Outline` cuando corresponda, sobre todo en aquellos que contienen variables (datos del tipo `<dato>`), y agregando los ejemplos (`Examples`) oportunos. En la Figura 17 se puede ver un ejemplo de modificación del escenario que aparece en la Figura 16.

```
1: Scenario: Write your scenario here.
2: Given I should see <Operador1> on display
3: When I press ID op
4: Then I should see <OperacionYOperador1> on display
```

Figura 16. Escenario obtenido para la aplicación Calculadora generado por Graph2Test

```
1: Scenario Outline: Write your scenario here.
2: Given I should see <Operador1> on display
3: When I press ID op
4: Then I should see <OperacionYOperador1> on display
5:
6: Examples:
7:     | Operador1| OperacionYOperador1|
8:     | 9        |"- 9"               |
9:     | 2        | "/ 2"              |
```

Figura 17. Ejemplo de completado de un escenario de la aplicación Calculadora

Uno de los convenios utilizados en Graph2Test es que si se desea que la variable sea tomada como identificador (por ejemplo, de un botón) es necesario indicarlo mediante el uso de la palabra `ID` delante del texto que queremos tomar como identificador. En el ejemplo visto anteriormente en la Figura 16, la variable `op` tiene delante la palabra clave `ID`, por tanto, en el momento de la generación de código obtendremos un resultado similar al de la Figura 18.

```
1:  onView(withId(R.id.op)).perform(click());
```

Figura 18. Resultado de la traducción a código Espresso de la frase "I press ID op"

Si lo que se desea es poder utilizarlo como variable, podemos cambiar en el diagrama inicial la frase "I press ID op" por "I press op", obteniendo como

resultado en el escenario la frase que se observa en la Figura 19 y pudiendo modificar dicho escenario manualmente para que quede como aparece en la Figura 20. En ese caso, la traducción a código Espresso diferiría de la observada anteriormente. Este cambio se puede ver reflejado en la Figura 21.

```
1:   When I press <op>
```

Figura 19. Resultado en el escenario de la conversión al cambiar la frase "I press ID op" por "I press op" en el diagrama inicial

```
1: Scenario Outline: Write your scenario here.
2: Given I should see <Operador1> on display
3: When I press <op>
4: Then I should see <OperacionYOperador1> on display
5:
6: Examples:
7:     | Operador1| op | OperacionYOperador1|
8:     | 9        | -  | "- 9"              |
9:     | 2        | /  | "/ 2"              |
```

Figura 20. Nueva versión de escenario completado manualmente para la aplicación Calculadora

```
1:   onView(withText(op)).perform(click());
```

Figura 21. Resultado de la generación de código al cambiar la frase "I press ID op" por "I press op" en el diagrama inicial

El usuario puede realizar las modificaciones oportunas en el esqueleto de pruebas, tanto para el generado por Cucumber como para el obtenido después del rellenado parcial con código Espresso, para que al lanzar las pruebas se obtenga la navevación de la aplicación por todos los estados modelados. Para el escenario de la Figura 16, los métodos obtenidos son los que aparecen en la Figura 22.

```
1: @Given("^I should see <Operador(\\d+)> on display$")
2: public void i_should_see_Operador_on_display(int arg1) throws Throwable {
3:     onView(withId(__DATA__)).check(matches(withText(Operador1)));
4:     // Write code here that turns the phrase above into concrete actions
5:     throw new PendingException();
6: }
7:
8: @When("^I press ID op$")
```

```
9: public void i_press_ID_op() throws Throwable {
10:     onView(withId(R.id.op)).perform(click());
11:     // Write code here that turns the phrase above into concrete actions
12:     throw new PendingException();
13: }
14:
15: @Then("^I should see <OperacionYOperador(\\d+)> on display$")
16: public void i_should_see_OperacionYOperador_on_display(int arg1)
          throws Throwable {
17:     onView(withId(__DATA__)).check(matches(withText(OperacionYOperador1)));
18:     // Write code here that turns the phrase above into concrete actions
19:     throw new PendingException();
20: }
```

Figura 22. Esqueleto parcialmente rellenado obtenido para el escenario de la Figura 16

En general, los casos obtenidos para esta aplicación resultan muy acertados, puesto que se ha conseguido generar y rellenar un esqueleto de pruebas siendo necesaria una mínima edición posterior por parte del usuario para implementar por completo los escenarios.

4.2. Aplicación Farmacias

Farmacias Ahora! ZGZ[2] es una aplicación Android que permite conocer cuáles son las farmacias que se encuentran abiertas en tiempo real en la ciudad de Zaragoza, incluyendo las farmacias de guardia e información relevante sobre todas las farmacias de la ciudad. La aplicación ha sido desarrollada por la empresa GeoSLab y utiliza datos oficiales proporcionados el Ayuntamiento de Zaragoza y el Colegio Oficial de Farmacéuticos de Zaragoza.

[2] http://www.geoslab.com/es/product/farmacias-ahora-zgz

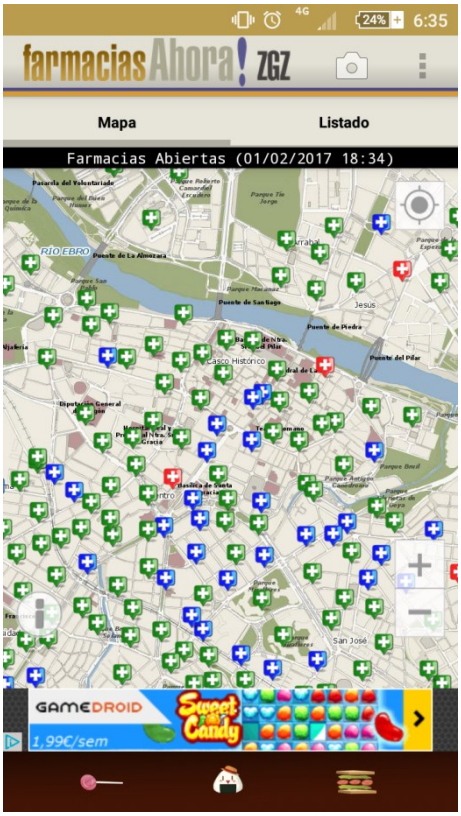


Figura 23. Aplicación Farmacias Ahora! ZGZ ejecutada sobre un dispositivo físico Android

A continuación se presenta el catálogo de requisitos funcionales de la aplicación Farmacias que aparecen reflejados en la Tabla 4, los cuales han sido fundamentales para la creación del diagrama diseñado posteriormente para utilizarlo como dato de entrada de Graph2Test.

REQUISITOS FUNCIONALES	
Código	**Descripción**
RF-1	Mostrar todas las farmacias en el mapa de Zaragoza
RF-2	Mostrar todas las farmacias en el listado
RF-3	Mostrar farmacias abiertas en el mapa
RF-4	Mostrar farmacias abiertas en el listado
RF-5	Mostrar leyenda al pulsar el botón de información
RF-6	Mostrar teléfonos de interés a través del menú
RF-7	Permitir realizar una llamada a los teléfonos de interés
RF-8	Mostrar fecha y hora de la última actualización de datos realizada
RF-9	Permitir escoger el rango de fecha y hora a través del menú
RF-10	Actualizar datos al detectar conexión a internet
RF-11	Actualizar datos al pulsar el botón de refresco a través del menú

RF-12	Actualizar localización (posición del usuario) al activar el GPS
RF-13	Centrar mapa en el punto de localización al pulsar su botón
RF-14	Acercar imagen del mapa al pulsar el botón de aumentar zoom
RF-15	Alejar imagen del mapa al pulsar el botón de disminuir zoom
RF-16	Activar cámara al pulsar botón de cámara
RF-17	Mostrar todas las farmacias a menos de 1500m en la dirección que apunta la cámara
RF-18	Mostrar farmacias abiertas a menos de 1500m en la dirección que apunta la cámara
RF-19	Mostrar nombre y dirección de la farmacia al pulsarla en el mapa
RF-20	Mostrar datos relevantes de la farmacia al solicitar más información a través del mapa
RF-21	Mostrar datos relevantes de la farmacia al solicitar más información a través del listado
RF-22	Mostrar ruta en coche para llegar desde la posición actual a la farmacia seleccionada
RF-23	Mostrar ruta a pie para llegar desde la posición actual a la farmacia seleccionada
RF-24	Mostrar indicaciones de ruta al desplazar el botón de ruta
RF-25	Permitir realizar una llamada a la farmacia seleccionada

Tabla 4. Tabla de requisitos de la aplicación Farmacias

Al igual que para la aplicación calculadora, el diagrama que se presenta a continuación en la Figura 24 ha sido creado utilizando la herramienta de diseño de diagramas YEd Graph Editor. En la Figura 25 se muestra la parte superior del diagrama, ya que dicho diagrama tiene dos partes iguales que difieren en el valor de una variable pero es necesario modelarlo por completo para poder realizar las pruebas correctamente.

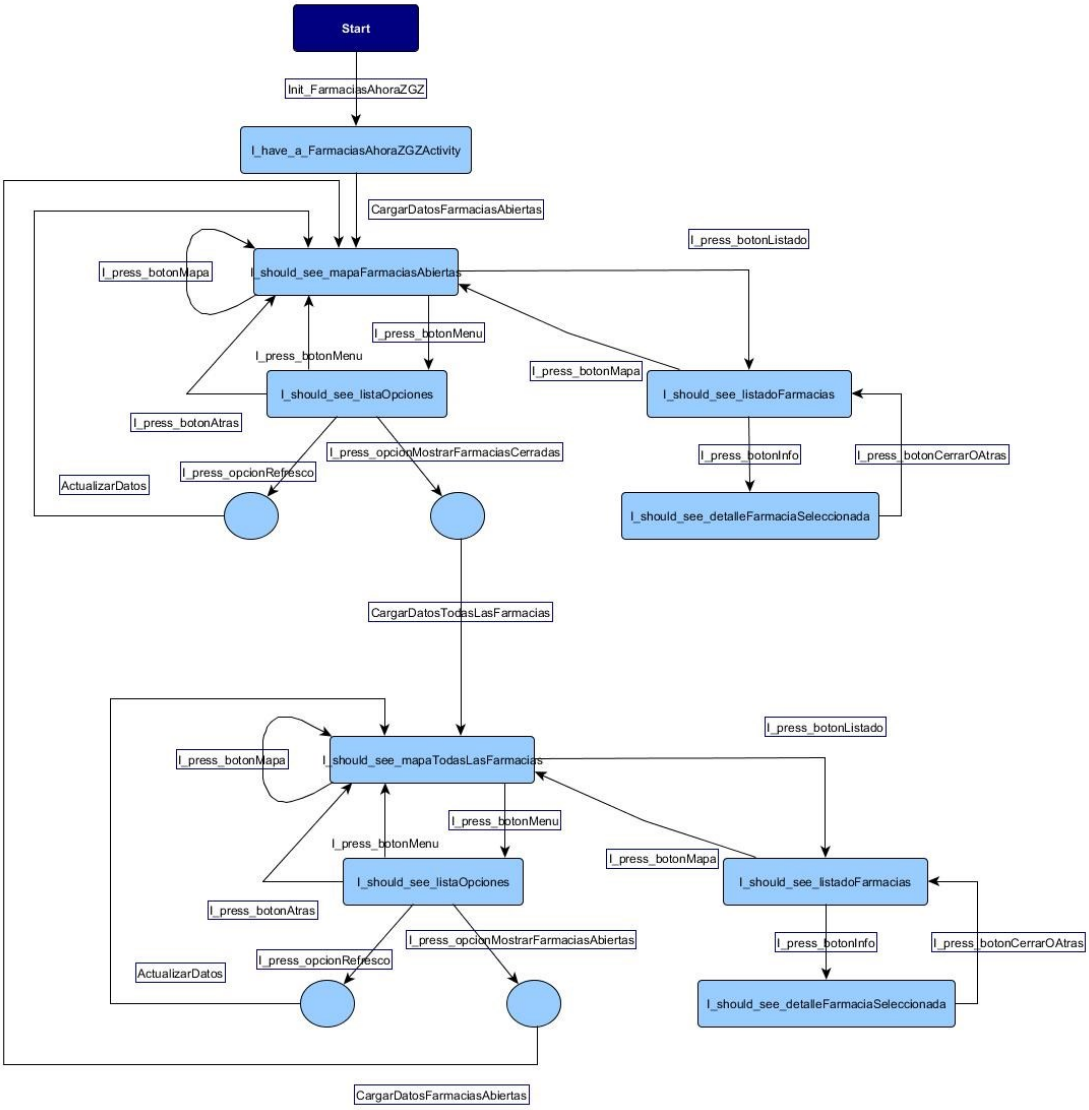


Figura 24. Diagrama de estados completo de la aplicación Farmacias

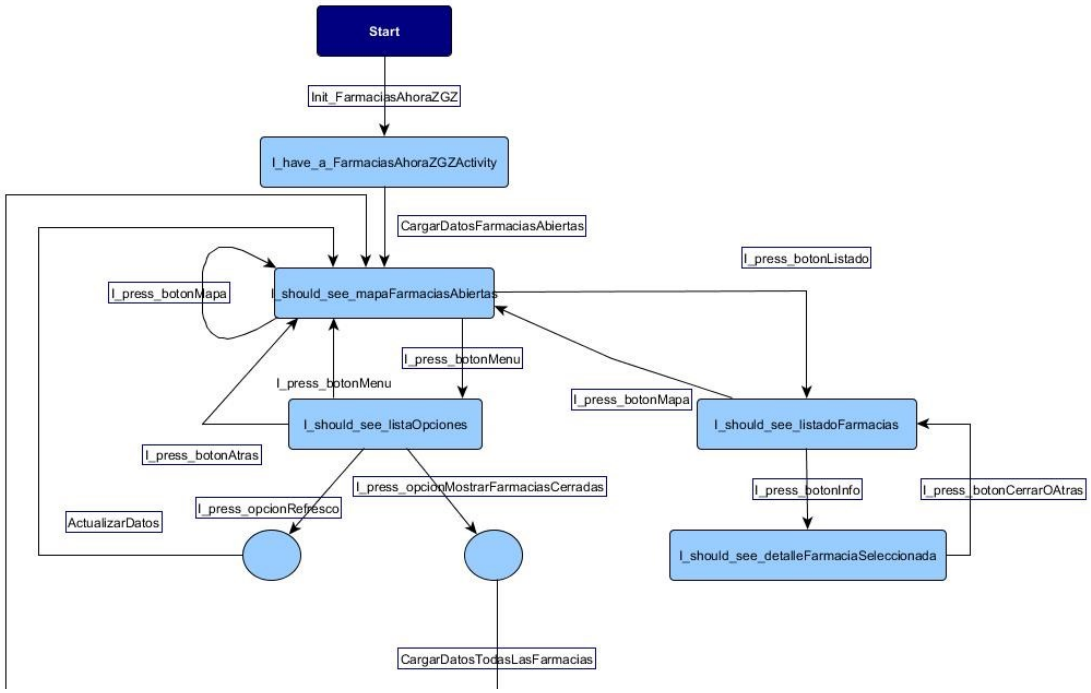


Figura 25. Diagrama de estados parcial de la aplicación Farmacias (parte superior)

Como se puede comprobar, solo se ha tenido en cuenta una parte de los requisitos de la aplicación farmacias, ya que contiene un amplio rango de funciones que no son posibles de comprobar con un emulador, como por ejemplo, búsqueda de farmacias en tiempo real utilizando la cámara, o funciones cuyas pruebas se encuentran limitadas por las posibilidades ofrecidas por Espresso, como puede ser pruebas referentes al refresco de los datos mostrados en el mapa.

Puede observarse que el diagrama diseñado duplica los estados teniendo en cuenta solo si se desea mostrar las farmacias abiertas disponibles o ver todas las farmacias existentes en la ciudad. Esta duplicación se debe a que no ha sido posible obtener una correcta ejecución de GraphWalker utilizando una codificación de palabras que hagan al estado dependiente del valor de una variable que sea modificada de manera externa (es posible utilizar variables en los nodos del diagrama y lanzar GraphWalker sin problemas, pero la variable debe ser inicializada y modificada dentro del propio diagrama).

Aparecen también unos nodos vacíos, los cuales son utilizados para indicar que se desea hacer una acción por parte del usuario y acto seguido ejecutar una acción por parte del programa para después ejecutar otra acción (ejecutar

dos acciones seguidas). Esto se debe que se toma como precondición que cada arista/nodo tenga sólo una acción a realizar. Si se desea poner más de una, será necesario unirlas mediante la palabra clave AND y poner cada una en una línea dentro del fichero.

Para esta aplicación se han obtenido los resultados que aparecen reflejados en el Anexo C. Se han obtenido un total de 37 escenarios, de los cuales se tomará como ejemplo el que aparece en la Figura 26 para comentar los resultados obtenidos. Al igual que para la aplicación Calculadora, es necesario realizar modificaciones por parte del usuario tanto para el fichero `feature` como cambiar los escenarios normales por `Scenario Outline` cuando corresponda sobre todo en aquellos que contienen variables (datos del tipo `<dato>`) y agregar los ejemplos oportunos.

Para este caso hay una mayor dificultad de automatización dada la complejidad de los elementos utilizados en la aplicación. Con la funcionalidad desarrollada para Graph2Test y los datos que se pueden obtener a partir del diagrama el resultado de la ejecución de Graph2Test es más ajustado para la aplicación Calculadora. Para el escenario que aparece en la Figura 26, el código resultante es el que se puede ver en la Figura 27.

```
1: Scenario: Write your scenario here.
2: Given I should see <listaOpciones>
3: When I press <botonMenu>
4: Then I should see <mapaTodasLasFarmacias>
```

Figura 26. Escenario obtenido para la aplicación Farmacias generado por Graph2Test

```
1: @Given("^I should see <listaOpciones>$")
2: public void i_should_see_listaOpciones() throws Throwable {
3:     onView(withText(listaOpciones)).check(matches(withText(listaOpciones)));
4:     // Write code here that turns the phrase above into concrete actions
5:     throw new PendingException();
6: }
7:
8: @When("^I press <botonMenu>$")
9: public void i_press_botonMenu() throws Throwable {
10:      onView(withText(botonMenu)).perform(click());
11:     // Write code here that turns the phrase above into concrete actions
12:     throw new PendingException();
```

```
13: }
14:
15: @Then("^I should see <mapaTodasLasFarmacias>$")
16: public void i_should_see_mapaTodasLasFarmacias() throws Throwable {
17:     onView(withText(mapaTodasLasFarmacias))
                .check(matches(withText(mapaTodasLasFarmacias)));
18:     // Write code here that turns the phrase above into concrete actions
19:     throw new PendingException();
20: }
```

Figura 27. Ejemplo de completado de un escenario de la aplicación Farmacias

A modo de ejemplo, en este caso resulta más útil hacer uso de los `IDs`, ya que los elementos con los que se interactúa en esta aplicación no suelen disponer de un texto visible de referencia. Realizando esas modificaciones, se obtienen resultados como los que aparecen en la Figura 28 para los escenarios y Figura 29 para el código Espresso.

```
1: Scenario: Write your scenario here.
2: Given I should see ID listaOpciones
3: When I press ID botonMenu
4: Then I should see ID mapaTodasLasFarmacias
```

Figura 28. Escenario resultante de modificar los textos en el diagrama para obtener una correspondiencia con los identificadores

```
1: @Given("^I should see ID listaOpciones$")
2: public void i_should_see_ID_listaOpciones() throws Throwable {
3:       onView(withId(R.id.listaOpciones))
                  .check(matches(withId(R.id.listaOpciones)));
4:     // Write code here that turns the phrase above into concrete actions
5:     throw new PendingException();
6: }
7:
8: @When("^I press ID botonMenu$")
9: public void i_press_ID_botonMenu() throws Throwable {
10:      onView(withId(R.id.botonMenu)).perform(click());
11:     // Write code here that turns the phrase above into concrete actions
12:     throw new PendingException();
13: }
```

```
14:
15: @Then("^I should see ID mapaTodasLasFarmacias$")
16: public void i_should_see_ID_mapaTodasLasFarmacias() throws Throwable {
17:       onView(withId(R.id.mapaTodasLasFarmacias))
                    .check(matches(withId(R.id.mapaTodasLasFarmacias)));
18:     // Write code here that turns the phrase above into concrete actions
19:     throw new PendingException();
20: }
```

Figura 29. Esqueleto parcialmente rellenado obtenido para el escenario de la Figura 26

Los casos obtenidos para esta aplicación resultan también bastante acertados dada la complejidad de la aplicación de Farmacias, ya que hay casos difíciles de abordar, como es el caso de ver el mapa de todas las farmacias que aparece en el ejemplo anterior. En general, los resultados para las funcionalidades que se encuentran dentro del alcance de Graph2Test han sido satisfactorios.

5. Conclusiones y trabajo futuro

5.1. Planificación

A continuación, en la Figura 30 se muestra el diagrama de Gantt correspondiente a la planificación del proyecto. Se puede observar que hay periodos de tiempo en los que coinciden el estudio previo con el análisis y el análisis con el diseño de Graph2Test. Esto es debido a la compaginación del estudio de las distintas herramientas existentes, las cuales debían poseer características que se pudiesen adaptar a las herramientas previamente seleccionadas (Cucumber y Espresso) llegando a casos en los que se ha optado por abandonar el uso de una herramienta seleccionada por haber encontrado otra que ofrece mejores características para el desarrollo del trabajo (por ejemplo, el uso de GraphWalker sobre Modelio o Enterprise Architect). Además, había que valorar todas las adaptaciones que había que realizar mediante Graph2Test y el esfuerzo adicional que debía hacer el usuario para poder utilizarlo correctamente.

También hay que destacar el tiempo invertido en el desarrollo y pruebas *software*, los cuales se han extendido ya que la compilación y ejecución del código en Android Studio con el dispositivo hardware utilizado resultaba excesivamente costoso en tiempo.

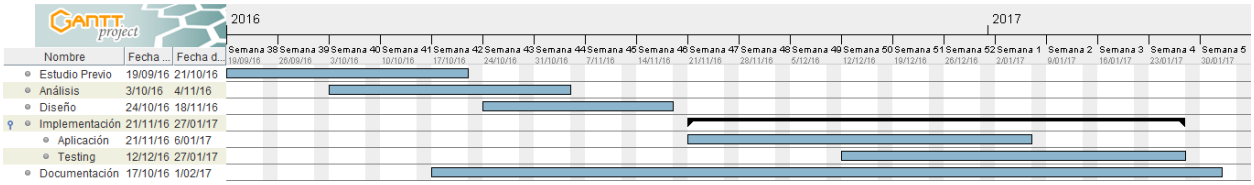


Figura 30. Diagrama de Gantt del proyecto

En total se han invertido 430h repartidas en 19 semanas. El desglose de las horas queda de la siguiente manera:

- Estudio previo: 60h
- Análisis: 65h
- Diseño: 55h
- Implementación: 180h
  - Aplicación: 105h
  - Testing: 75h
- Documentación: 70h

5.2. Conclusiones

La realización de este trabajo ha supuesto un beneficio para mi formación, ya que me ha permitido aprender nuevas tecnologías, tales como son Cucumber, Gradle y Espresso, y comprender que, a pesar de ser diferentes, su finalidad es la misma; hacer más asequible y rápida la programación de aplicaciones y su proceso de pruebas.

El trabajo realizado consiste en la creación de un de componente que integre algunas de las herramientas existentes para desarrollo de pruebas de aceptación de usuario, dando como resultado una mayor facilidad de uso, reducción del tiempo empleado en el proceso de pruebas y generación parcial de elementos de pruebas (código y textos mínimamente controlados). A pesar de los problemas encontrados relacionados con la unificación transparente de diversas herramientas y el uso de los nuevos entornos de programación en ordenadores no tan modernos, se han obtenido buenos resultado que, como primera aproximación y ajustándose a los objetivos y alcance de un Trabajo Fin de Grado, se ha conseguido desarrollar un prototipo con funcionalidades básicas que se pueden probar en una aplicación Android. La solución aportada permite realizar modificaciones en un futuro para poder implementar nuevas funcionalidades de carácter más avanzado.

Graph2Test posee un carácter innovador, ya que a fecha de presentación de este trabajo no existe un componente que ofrezca la integración de distintas herramientas de pruebas y obtenga, a partir de un elemento básico del desarrollo de *software*, una implementación semi-automática del código de pruebas. Herramientas de gestión y diseño de diagramas y componentes de aplicaciones y herramientas de pruebas que utilizan distintas metodologías son un ejemplo del contenido variado que ha sido abordado en este TFG, además de los temas estudiados en el Grado en Ingeniería Informática, tales como algoritmia, estructuras de datos, distintos lenguajes de programación y aplicación de la ingeniería del *software* para el diseño de diagramas y componentes.

5.3. Trabajo futuro

Graph2Test realiza una buena automatización y gestión para obtener una base de código de pruebas a partir de un mapa de navegación modelado como diagrama de estados. Sin embargo, a lo largo de la creación de dicho componente se han encontrado nuevas funcionalidades que mejorarían la

calidad del código generado, además de algunos problemas que no han aparecido pero se han identificado y son posibles de solucionar.

En un principio, se contempló la posibilidad de desarrollar este componente como un plug-in [14] para Android Studio. Esa idea se descartó dado que la creación de Graph2Test como un archivo ejecutable resultaba más atractiva en tiempo y se adaptaba mejor a la planificación realizada. Sin embargo, es una idea que resultaría muy útil a la hora de realizar la generación de casos de prueba en el desarrollo del *software* ya que de esa manera se tendría todo unificado en un mismo entorno de trabajo.

Referente a los diagramas, se podrían crear de tal manera que no resulte en partes redundantes (repetición de aristas o nodos que representan la misma funcionalidad cambiando las variables de entrada) ni aristas o nodos vacíos como ocurría en los ejemplos mostrados.

A la hora de generar los escenarios de Cucumber, se realiza un paso de los textos en el grafo a sentencias en lenguaje Gherkin de una manera muy básica. Se podría implementar el reconocimiento de unos "estados finales" de la aplicación objeto de pruebas para así poder generar directamente unos escenarios más completos. En el ejemplo visto de la aplicación Calculadora, esto sería muy útil para no realizar verificaciones intermedias y comprobar solo el "final" (como sería el realizar varias operaciones aritméticas seguidas y comprobar el resultado al pulsar el botón "="). Una primera aproximación podría ser incluir una palabra clave dentro del diagrama de estados inicial (por ejemplo, la cadena "0FINAL0", ya que el carácter barra baja ("_") se usa como separador de palabras) y realizar el recorrido del grafo traduciendo el prmer nodo por un `GIVEN` en el fichero de escenarios, tomando como sentencia `WHEN` la primera arista encontrada e ignorando todos los nodos intermedios y separando los textos de las aristas mediante palabras clave `AND` hasta llegar a un nodo marcado con la palabra clave de finalización, la cual se traduciría por un `THEN`.

También se puede configurar Graph2Test para que reciba una lista de elementos de la aplicación con sus respectivos nombres Por ejemplo, indicar el ID de un botón y su nombre en el diagrama de estados, para así no tener que escribir el nombre del botón en el diagrama y quede mejor estéticamente, además de realizarse una correspondencia exacta del identificador.

La posibilidad de generar escenarios múltiples utilizando esta herramienta es una funcionalidad que puede resultar muy interesante de implementar. Para conseguirla, se podría utilizar un fichero de entrada en el que aparezcan los nombres de las variables y los valores de ejemplo. Unificándola con la funcionalidad descrita anteriormente, se podría utilizar el mismo fichero para ambas finalidades y obtener unos escenarios que necesiten una edición mínima o nula por parte del usuario.

Actualmente, si se desea empezar desde los escenarios y no desde el diagrama, solo es posible pasarle como entrada a Graph2Test un fichero `feature`. En la práctica normalmente se dispone de varios ficheros `feature` que componen los casos de uso de la aplicación. Se podría realizar una gestión de varios ficheros `feature` utilizando un mismo diagrama en una misma sesión, o incluso que la salida generada se realice en distintos ficheros `feature`.

Otro aspecto a mejorar es la gestión de entradas y salidas del componente en tiempo de ejecución. A modo de ejemplo, para los datos de entrada se podría evitar tener que introducir la ruta fichero `feature` origen si lo que se desea es partir del esqueleto para su rellenado. Para los datos de salida se podría preguntar al usuario si desea modificar el fichero `feature` antes de pasar a la generación del esqueleto, o si quiere modificar el esqueleto generado antes de proceder con su rellenado con código Espresso. También se debería dejar elegir al usuario el nombre que tendrá cada uno de los ficheros generados (por ahora solo es posible pasarle una ruta de salida y el nombre de cada elemento de salida es el mismo que el del elemento de entrada modificando la extensión por la pertinente).

Respecto a la generación de código a partir de texto plano se pueden hacer varias mejoras. Una podría ser utilizar un diccionario más grande y eficiente que cubriese una mayor parte de los posibles textos que pudiese introducir el usuario y, de esta manera, no condicionar tanto el uso de Graph2Test con una sintaxis tan estricta como la que posee actualmente. Se podría también ampliar el abanico de posibilidades de generación de código, por ejemplo, referentes a pulsaciones en elementos comunes a las aplicaciones Android (como pulsar el botón atrás o el botón menú) y tratamiento de elementos y acciones que requieran una mayor complejidad (por ejemplo, las pruebas en mapas o en aplicaciones abiertas desde la aplicación que se está probando).

Glosario

A* – Algoritmo de búsqueda en grafo que siempre encontrará un camino de un nodo A a un nodo B, si dicho camino existe.

BDD – Behaviour Driven Development. Desarrollo guiado por el comportamiento. Es un proceso de desarrollo de software basado en especificaciones del comportamiento de la aplicación a desarrollar. [33]

DTM – *Dialogues Test Method*. Método de pruebas de diálogo. Es un enfoque de pruebas de metodología ágil que permite el desarrollo del código *software* a partir de las pruebas *software*, es decir, el proceso de pruebas se inicia antes de que se haya escrito una línea de código. [17]

GUI – *Graphical User Interface.* Interfaz Gráfica de Usuario. Tipo de interfaz de usuario que permite interactuar con los dispositivos electrónicos a través de indicadores visuales en lugar de texto.

Herramienta CASE – *Computer Aided Software Engineering*. Ingeniería de Software Asistida por Computadora. Son diversas aplicaciones o programas informáticos destinados a aumentar la productividad en el desarrollo de software reduciendo el costo de las mismas en términos de tiempo y de dinero. [29]

Log – Grabación secuencial en un registro, ya sea en un archivo o en una base de datos, de todos los acontecimientos (eventos o acciones) que afectan a un proceso particular (aplicación, actividad de una red informática…). De esta forma constituye una evidencia del comportamiento del sistema. [18]

MBT – *Model Based Testing*. Desarrollo de pruebas basadas en modelos. Es un proceso de desarrollo de pruebas a partir de la descripción de modelos UML. [19]

NDT – *Navigational Development Techniques*. Técnicas de desarrollo de navegación. Es una metodología de desarrollo de sistemas *software* basado en la definición de los requisitos de la aplicación (comportamiento).

Plug-in – Complemento. Aplicación que se relaciona con otra para agregarle una función nueva y generalmente muy específica. Esta aplicación adicional es ejecutada por la aplicación principal e interactúan por medio de la interfaz de programación de aplicaciones. [24]

UML – *Unified Modeling Language.* Lenguaje unificado de modelado. Es el lenguaje de modelado gráfico de sistemas *software*, el cual ofrece un estándar para describir un modelo del sistema. [20]

XML – *eXtensible Markup Languaje*. Lenguaje de Marcas Extensible. Es un meta-lenguaje que permite definir un conjunto de normas para la codificación de documentos en un formato que es tanto legible por humanos como por la máquina. [21]

Bibliografía

[1] Android. Página web oficial.
https://www.android.com/intl/es_es/

[2] Oracle Java 8 SE. Página web oficial.
http://docs.oracle.com/javase/8/docs/

[3] Android Studio. Página web oficial.
https://developer.android.com/studio/index.html

[4] Gradle. Página web oficial.
https://gradle.org/getting-started-gradle/

[5] GraphWalker. Página web oficial
http://graphwalker.github.io/

[6] YEd Graph Editor. Página web oficial
http://www.yworks.com/products/yed

[7] Cucumber. Página web oficial.
https://cucumber.io/docs/reference/jvm

[8] Gherkin. Definición del lenguaje en el Github de Cucumber.
https://github.com/cucumber/cucumber/wiki/Gherkin

[9] Espresso. Artículo en el Github de Android.
https://google.github.io/android-testing-support-library/docs/espresso/index.html

[10] JUnit. Artículo en Github.
https://github.com/junit-team/junit4/wiki/Getting-started

[11] Proceso de pruebas software. Definición tomada de los apuntes de la asignatura Verificación y Validación de la Universidad de Zaragoza.

[12] SAX. Librería de Java para lectura de ficheros XML.
https://docs.oracle.com/javase/tutorial/jaxp/sax/parsing.html

[13] SAX. Artículo de la web de Mkyong. Pasos para leer un fichero XML utilizando SAX.
https://www.mkyong.com/java/how-to-read-xml-file-in-java-sax-parser/

[14] Plug-in. Artículo de la web oficial de IntelliJ. Pasos para crear un plug-in.
http://www.jetbrains.org/intellij/sdk/docs/basics/getting_started.html

[15] Definición de Log. Wikipedia.
https://es.wikipedia.org/wiki/Log_(inform%C3%A1tica)

[16] Apache Log4j2. Página web oficial de Apache.
https://logging.apache.org/log4j/2.x/

[17] Definición de DMT. Artículo de la web dialoguestechnology.nl
http://www.dialoguestechnology.nl/2011/11/agile-testing-3/

[18] Definición de Log. Wikipedia.

https://es.wikipedia.org/wiki/Log_(inform%C3%A1tica)
[19] Definición de MBT. Wikipedia.
https://en.wikipedia.org/wiki/Model-based_testing
[20] Definición de UML. Wikipedia.
https://es.wikipedia.org/wiki/Lenguaje_unificado_de_modelado
[21] Definición de XML. Wikipedia en inglés.
https://en.wikipedia.org/wiki/XML
[22] NDT Suite. JJ Gutiérrez, MJ Escalona, M Mejías, J Torres, 2006. An approach to generate test cases from use cases. Proceedings of the 6th international conference on Web engineering, 113-114
[23] NDT Suite. MJ Escalona, M Mejías, J Torres, 2004. 13th International conference on information systems development: methods and tools, theory and practice, Vilna, Lithuania, pp 149-59
[24] Definición de Plugin. Wikipedia.
https://es.wikipedia.org/wiki/Complemento_(inform%C3%A1tica)
[25] ISO/IEC/IEEE 29119-4, 2015. International Standard for Software and systems engineering--Software testing--Part 4: Test techniques
[26] Espresso. Tabla con los métodos disponibles y la manera de usarlos.
https://google.github.io/android-testing-support-library/docs/espresso/cheatsheet/index.html
[27] Android 2.2 Froyo. Página web official de desadolladores Android.
https://developer.android.com/about/versions/android-2.2-highlights.html
[28] Android 4.3 Jelly Bean. Página web official de desadolladores Android
https://developer.android.com/about/versions/jelly-bean.html
[29] Definición de Herramienta Case. Wikipedia
https://es.wikipedia.org/wiki/Herramienta_CASE
[30] Enterprise Architect. Página web oficial de Sparx Systems.
http://www.sparxsystems.com/products/ea/
[31] Modelio. Página web oficial.
https://www.modelio.org/documentation-menu/quick-start-guide.html
[32] DTM Tool. Página web oficial.
http://dtmtool.com/
[33] Definición de BDD. Wikipedia en inglés.
https://en.wikipedia.org/wiki/Behavior-driven_development
[34] Defnición de UNIX. Wikipedia.
https://es.wikipedia.org/wiki/Unix
[35] Adobe Air. Página web oficial de Adobe.
http://www.adobe.com/es/products/air.html

[36] UIAutomator. Página web oficial de Android
https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html
[37] Definición de GUI. Wikipedia en inglés.
https://en.wikipedia.org/wiki/Graphical_user_interface

Anexo A. Tecnologías para el diseño y automatización de pruebas

A.1. Introducción

Previamente a la selección de las herramientas integradas dentro de la infraestructura propuesta en este TFG para automatizar el diseño y la ejecución de pruebas de aceptación, se realizó un estudio previo de las herramientas que se podían utilizar.

En primer lugar, se analizaron distintas alternativas que facilitasen la generación automática de casos de prueba. En particular, se analizó la adaptación de herramientas CASE como Enterprise Architect [30] o Modelio [31] y la utilización de las herramientas DTM, NDT-Suite y GraphWalker.

Una primera opción fue la posible generación *ad hoc* de casos de prueba a partir de diagramas de actividad modelados con Enterprise Architect que representasen los mapas de navegación de las aplicaciones móviles. Enterprise Architect es una herramienta basada en UML para realizar diseño y construcción de sistemas software y modelado visual de procesos de negocio. No se optó por usar esta herramienta porque hay otras más sencillas de utilizar con las que se obtienen los mismos resultados, además de disponer tan solo de una versión de prueba de 30 días con funcionalidades limitadas; es decir, es necesario disponer de una licencia para su uso completo.

También se analizó la posibilidad de utilizar Modelio para para la generación *ad hoc* de casos de prueba a partir de diagramas de actividad modelados con Modelio. Modelio es similar a Enterprise Architect, con la diferencia de que su licencia es de código abierto y, por tanto, no es necesario adquirir una licencia de pago para poder utilizarlo. Esta herramienta permite el modelado de los distintos diagramas que forman parte del diseño de sistemas software. No se optó por usar esta herramienta como parte a integrar en Graph2Test porque hay otras más sencillas de utilizar con las que se obtienen los mismos resultados, siendo éstos incluso mejor adaptados a las otras herramientas utilizadas. Sin embargo, sí que ha sido utilizada para la gestión y diseño de los diagramas del componente de automatización desarrollado.

Viendo que la adaptación de herramientas CASE presentaba ciertas dificultades y en algunos casos se necesitaba la utilización de herramientas comerciales,

se analizó la posibilidad de utilizar herramientas que realizasen directamente el diseño de casos de prueba basados en modelos (MBT).

Por ejemplo, la herramienta DTM [32] utiliza diagramas de flujo como entrada, los cuales se obtienen realizando el diseño dentro de la misma aplicación, y, a partir de ellos, genera como salida secuencias de casos de prueba basados en cobertura media (ejecución de secuencias de caminos hasta llegar a un nodo final de la manera más rápida) o cobertura total (ejecución de secuencias de caminos pasando por la mayoría de nodos posibles antes de llegar a un nodo final). Además, se pueden realizar referencias a otros modelos y, de esta manera, tener diagramas de flujo más cortos que puedan estar relacionados entre sí. Las salidas pueden ser exportadas en formato PDF o Excel, indicando en ambos casos el camino generado, los pasos seguidos en cada momento y la salida esperada de cada nodo decisión. Las razones principales por las que se ha descartado esta herramienta han sido las siguientes: su licencia no es gratuita, no resulta sencilla su instalación en sistemas UNIX [34] y es necesario tener instalado Adobe Air [35] para poder utilizar esta herramienta

NDT-Suite [23] también es un conjunto de herramientas que permite la aplicación de la tecnología NDT en un determinado proyecto *software* y, en particular, la generación de casos de prueba a partir de casos de uso especificados mediante diagramas de actividad [22]. Como desventaja se ha observado que no resulta sencilla la adaptación para el flujo de trabajo presentado para este TFG y la ejecución de automática de las pruebas sobre aplicaciones en entornos móviles.

Finalmente, se estudió la herramienta GraphWalker como alternativa para la generación de casos de prueba. GraphWalker es una herramienta MBT que utiliza como entrada diagramas de estado finitos o grafos dirigidos, los cuales se realizan utilizando la el diseñador de diagramas YEd Graph Editor. Este editor de grafos genera una salida cuya estructura interna está en formato XML. A partir de dicho grafo, la herramienta GraphWalker genera un esqueleto de código, los cuales son los nombres de las aristas/vértices y que se utilizan también como los nombres de los métodos, además de rellenarse posteriormente con las acciones que se deben realizar en cada momento (la salida es similar a la generada al utilizar la herramienta Cucumber). La principal desventaja que se ha observado con esta herramienta es que solo se puede obtener un camino cada vez que es ejecutada, aunque dicho camino, si así se le indica, realizará una cobertura del 100% de los nodos o aristas.

Pese a ello, se ha seleccionado por la facilidad de integración con la herramienta Cucumber y la simplicidad a la hora de realizar los diagramas de estados con la herramienta YEd Graph Editor.

Por otro lado, una vez que se tuvo claro cómo generar los casos de prueba, se analizaron también distintas alternativas para entornos para ejecución automatización de pruebas. Como se ha comentado anteriormente, la herramienta GraphWalker puede utilizarse también como entorno de ejecución. Sin embargo, en este TFG se pretendía que el usuario final o parte aceptante tuviese una descripción lo más sencilla posible de los escenarios de las prueba, permitiendo, si así lo desease, cambiar algunos parámetros de los escenarios de estas pruebas. Por ello, se optó por analizar las posibilidades ofrecidas por la tecnología Cucumber.

Cucumber es una herramienta que utiliza descripciones funcionales en texto plano (prácticamente en lenguaje natural) como base para realizar pruebas automatizadas. Para utilizar esta herramienta, hay que describir el comportamiento esperado utilizando el lenguaje Gherkin, el cual permite describir el comportamiento del software sin entrar en detalles de implementación. De hecho, esta tecnología también se usa como base para una nueva metodología de desarrollo denominada Desarrollo Dirigido por el Comportamiento (BDD) donde las pruebas que especifican el comportamiento son previas al comienzo del desarrollo. En cualquier caso, mediante las descripciones se puede obtener el esqueleto de los métodos que permiten la definición de un paso (*step*) de las pruebas. Utilizando dicho esqueleto, se le indica el comportamiento de cada método, permitiendo posteriormente ejecutar dichos métodos de testeo siguiendo los pasos previamente descritos en texto plano. Posteriormente, se puede utilizar cualquier herramienta de pruebas para rellenar el esqueleto, tales como JUnit, Espresso, Selenium, Sikuli, etc., las cuales proveen métodos para interactuar con aplicaciones Android y web, entre otras.

Para este TFG se ha elegido Cucumber por su facilidad de uso y la rapidez de la ejecución de las pruebas, además de poder relacionar directamente los casos de uso de una aplicación o transiciones entre estados con las descripciones en lenguaje Gherkin. Sin embargo, tal como hemos comentado Cucumber solo genera el esqueleto de las pruebas y era necesario añadir el código de las pruebas que realmente actuase sobre el interfaz gráfico de usuario (GUI) de las aplicaciones desarrolladas para Android. Para poder

automatizar la ejecución de pruebas sobre el GUI de las aplicaciones Android hay dos alternativas principales en la actualidad: Espresso o UIAutomator [36].

Espresso es un *framework* mediante el cual se pueden crear pruebas de interfaz de usuario para simular interacciones entre un usuario y una aplicación Android. Esta herramienta facilita la gestión de la ejecución de las pruebas ya que provee de una sincronización automática de las acciones de las pruebas con la interfaz de usuario de la aplicación a probar (por ejemplo, comprueba automáticamente que la actividad principal de la aplicación Android este inicializada), además de contar con una API pequeña y fácil de utilizar, la cual sigue un patrón determinado a la hora de construir la codificación de cada acción. Las acciones cuentan con una búsqueda de la vista en la jerarquía de vista actual, realizar acciones sobre la vista seleccionada y comprobar el estado de dicha vista.

UIAutomator es una herramienta similar a Espresso, con la principal diferencia de que es mucho más potente ya que permite el acceso a múltiples aplicaciones durante el mismo test. Por ejemplo, permite modificar los ajustes durante la prueba de una aplicación de terceros, hecho de gran interés ya que algunas de las herramientas que se utilizan como caso de estudio en este TFG incluyen widgets externos para facilitar el uso de mapas de Google.

Ambas alternativas, Espresso y UIAutomator, parecían ajustarse a este TFG. Sin embargo, finalmente se decidió utilizar Espresso. Aunque Espresso no permite realizar pruebas automáticas sobre aplicaciones externas como el acceso a mapas de Google, se eligió esta herramienta por tener licencia de código abierto, su fácil aprendizaje y la integración con Android de manera transparente y simplificada, además de una compatibilidad con las versiones anteriores de Android bastante elevadas; soporta Android 2.2 Froyo (API 8) [27] en adelante. La principal desventaja que se observó de UIAutomator fue que resulta mucho más compleja de utilizar, más pesada al ser más potente y que la compatibilidad con versiones anteriores de Android no es muy amplia; soporta Android 4.3 Jelly Bean (API 18) [28] en adelante.

A continuación, se realizará una breve introducción a las funcionalidades ofrecidas por YEd Graph Editor y GraphWalker, Cucumber y Gherkin y Espresso, además de las funcionales que han sido seleccionadas para su uso en el desarrollo de Graph2Test.

A.2. YEd Graph Editor y GraphWalker

GraphWalker es una herramienta MBT que permite la automatización de pruebas utilizando como base grafos dirigidos, los cuales, en el ámbito de este TFG, serán representaciones de diagramas de navegación de la aplicación que se desea probar.

Esta herramienta recorre el grafo y genera un esqueleto de pruebas, el cual debe ser rellenado siguiendo la funcionalidad de la aplicación objeto de pruebas. Se puede utilizar cualquier herramienta de pruebas para rellenar el esqueleto, tales como Espresso, Selenium, Sikuli, etc. Se puede también generar caminos utilizando distintas técnicas, indicándole el grado de cobertura que se desea, el nodo objetivo o el algoritmo a utilizar (camino aleatorio, algoritmo A*, etc.). Al igual que con la herramienta DTM, se pueden realizar referencias a otros modelos que se dispongan en ficheros aparte. Una vez se tiene generado, relleno el esqueleto obtenido a partir del grafo e indicado las técnicas a utilizar, se obtiene un camino de prueba que lanzará en cada paso las acciones necesarias para poder verificar la correcta ejecución del programa.

YEd Graph Editor es un editor de grafos que soporta la creación de varios tipos de diagramas, pero será necesario crear un grafo dirigido similar al que se observa en la Figura 31 para poder utilizarlo correctamente con GraphWalker.

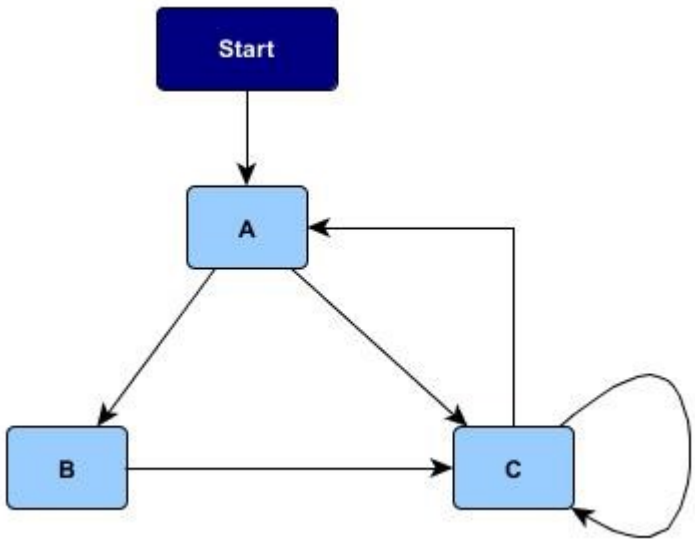


Figura 31. Ejemplo de grafo

Los grafos tienen dos elementos básicos: los nodos y las aristas. Referentes a estos elementos, hay que tener en cuenta lo siguiente:

- Los nodos representan un estado esperado que queremos evaluar, las cuales generalmente serán aserciones. Están representados por una "caja" (cualquier figura geométrica que pueda representarla, ya sea un cuadrado, un rectángulo, una circunferencia o cualquier elemento similar). GraphWalker no tiene en cuenta el color o la forma que tienen los vértices.
- Las aristas representan transiciones de un vértice a otro, las cuales hacen referencia a las acciones que se deben ejecutar para alcanzar el siguiente estado. Hay que tener en cuenta que GraphWalker solo admite aristas de una dirección (flechas unidireccionales) y que, al igual que con los vértices, es irrelevante el color o el grosor que tenga la arista.
- Los nombres de los nodos o aristas deben coincidir con la primera palabra de la primera línea de la etiqueta de cada elemento.

Se puede utilizar un nodo inicial `Start`, el cual no es obligatorio pero, en caso de aparecer debe ser único en el modelo. Además, del nodo inicial `Start` solo puede haber una arista que salga de él y estos nodos especiales no se incluirán en ningún recorrido generado con GraphWalker.

Utilizando los grafos creados con YEd Graph Editor, GraphWalker permite realizar un recorrido completo (visitando todas las aristas y nodos) del mismo para comprobar que el grafo ha sido creado correctamente. En caso de que haya algún tipo de fallo en el diseño, el recorrido del grafo no terminará. En el caso de Graph2Test, finalizará esa ejecución indicando el error en un log [18] pero la interfaz gráfica seguirá activa para poder lanzar una nueva ejecución. El uso del log aparece explicado en el Anexo B.4.

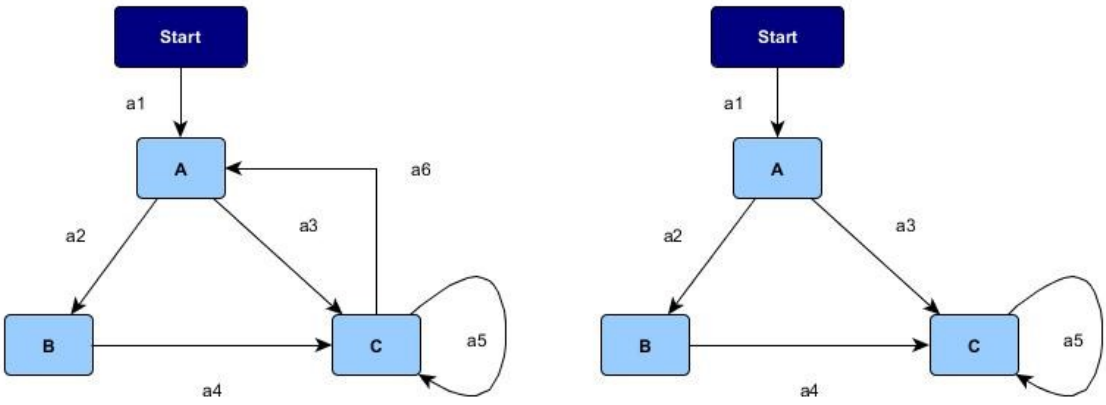


Figura 32. Grafo correcto (izquierda) vs incorrecto (derecha)

Como se puede comprobar en la Figura 32, si utilizamos el diagrama que no tiene un buen diseño (derecha en la figura) no es posible recorrer las aristas a2 ni a4 si inicialmente tomamos el camino ofrecido por la arista a3, ya que el nodo A resulta imposible de acceder desde el nodo C y, por tanto, tampoco es posible el acceso al nodo B.

Para realizar el recorrido del grafo, GraphWalker permite indicar el algoritmo y la condición de parada que queremos utilizar.

Algoritmos:

- random. Permite recorrer el grafo de manera aleatoria, eligiendo cada vez una arista no visitada.
- quick_random. Al igual que el algoritmo random permite recorrer el grafo de manera aleatoria, pero haciendo uso del algoritmo de Dijkstra para obtener el camino más corto hacia la arista elegida y no visitada.
- a_star. Permite recorrer el grafo utilizando el algoritmo A*.

Condiciones de parada:

- edge_coverage. El recorrido no se detendrá hasta que la cobertura de las aristas sea equivalente a la indicada (1-100). Cara arista recorrida cuenta sólo una vez, aunque se pase por ella en varias ocasiones.
- vertex_coverage. El recorrido no se detendrá hasta que la cobertura de los nodos sea equivalente a la indicada (1-100). Cara nodo recorrido cuenta sólo una vez, aunque se pase por él en varias ocasiones.

- **reached_vertex.** El recorrido no se detendrá hasta que se haya llegado al nodo que tiene el nombre indicado.
- **reached_edge.** El recorrido no se detendrá hasta que se haya llegado a la arista que tiene el nombre indicado.
- **time_duration.** El recorrido no se detendrá hasta que haya pasado el tiempo indicado en segundos.
- **length.** Utilizado para recorrer el grafo hasta que se haya alcanzado el total de pares arista-nodo indicado. Por ejemplo, si el número es 100, se recorrerán 200 elementos (100 pares de aristas y nodos)
- **never.** Se utiliza para no detener nunca el recorrido.

Para Graph2Test se ha seleccionado el algoritmo quick_random con una cobertura del 100% de las aristas (edge_coverage), ya que el objetivo final del uso de GraphWalker es poder recorrer el diagrama completo de la forma más eficiente posible y así obtener los textos de cada elemento en los ficheros de Cucumber.

A.3. Cucumber y Gherkin

Para poder utilizar Cucumber es necesario escribir especificaciones del comportamiento de la aplicación utilizando el lenguaje Gherkin. El idioma utilizado será el inglés, pero a modo de primer ejemplo se pondrán los textos no relevantes para la ejecución de la prueba en castellano. Es decir, en el ejemplo siguiente se mostrarán las palabras clave en inglés.

Cada característica (caso de uso o requisito) de la aplicación se corresponde con un *feature* en Cucumber, los cuales tienen la siguiente estructura:

```
1: Feature: Texto conciso y descriptivo del requisito/caso de uso
2:   Información adicional que haga la característica más fácil de comprender
3:
4:   Scenario: Descripción de una situación determinable
5:     Given (dada) una precondición
6:       And (y) otra precondición
7:     When (cuando) el usuario realiza una acción
8:       And (y) otra acción
9:       And (y) otra acción más
10:     Then (entonces) se alcanza un resultado comprobable
11:       But (pero) no sucede otra cosa (también comprobable)
12:
```

```
13:   Scenario: Otra situación pero dentro del contexto del requisito
14:       ...
```

Figura 33. Estructura de un fichero feature

Como se puede observar en el ejemplo anterior, las palabras clave básicas son `Feature`, `Scenario`, `Given`, `When`, `Then`, `And` y `But`. Por cada fichero feature, debe haber uno y solo un texto Feature, al menos un Escenario que contenga la descripción de la situación determinable y compuesto de al menos uno de éstos elementos: `Given`, `When` o `Then`. Es decir, podemos tener un Escenario del tipo `Given […] Then […]`, pero por motivos de legibilidad es recomendable utilizar por cada escenario un `Given`, un `When` y un `Then`. Así, se obtienen ejemplos como los siguientes: en la Figura 34 se puede ver un ejemplo básico de un fichero `feature` que cuenta únicamente con un escenario y en la Figura 35 se observa un ejemplo más compuesto de escenario.

```
1: Feature: Show a number on the display
2:
3:   Scenario: Enter a digit
4:     Given I have a CalculatorActivity
5:     When I press 1
6:     Then I should see 1 on the display
```

Figura 34. Definición simple de un Escenario

```
1: Feature: Add two numbers
2:   Calculate the sum of two numbers which consist of one digit
3:
4:   Scenario: Enter a digit, an operator and another digit
5:     Given I have a CalculatorActivity
6:     When I press 1
7:       And I press +
8:       And I press 2
9:       And I press =
10:    Then I should see 3 on the display
```

Figura 35. Definición de un escenario con condiciones múltiples

Como se puede observar en la Figura 36, es posible además utilizar variables dentro de cada escenario mediante los `Scenario Outline`, los cuales describen una situación comprobable para varios ejemplos distintos:

```
1: Feature: Calculate a result
2:   Perform an arithmetic operation on two numbers using a mathematical
3:    operator
4:
5:   Scenario Outline: Enter a digit, an operator and another digit
6:     Given I have a CalculatorActivity
7:     When I press <num1>
8:       And I press <op>
9:       And I press <num2>
10:      And I press =
11:    Then I should see <result> on the display
12:
13:   Examples:
14:     | num1 | num2 | op | result   |
15:     | 9    | 8    | +  | 17.0     |
16:     | 7    | 6    | -  | 1.0      |
17:     | 5    | 4    | x  | 20.0     |
18:     | 3    | 2    | /  | 1.5      |
19:     | 1    | 0    | /  | Infinity |display
```

Figura 36. Definición de un escenario múltiple

En este caso, se ha reutilizado el mismo escenario para comprobar la funcionalidad básica completa de una calculadora.

Graph2Test toma los nombres de las aristas/nodos de los grafos y los transforma en las sentencias Given-When-Then de Cucumber. Sin embargo, al no disponer de datos suficientes para crear los escenarios múltiples sólo se realiza la automatización a escenarios indicando los nombres de las variables. En la Figura 37 se puede ver un ejemplo de un fichero `feature` creado utilizando Graph2Test.

```
1: Feature: Enter your feature here
2:
3: Scenario: Enter your Scenario here
```

```
4: Given I have a CalculatorActivity
5: When I press <num1>
6: And I press <op>
7: And I press <num2>
8: And I press <eq>
9: Then I should see <result> on the display
```

Figura 37. Contenido de un fichero feature creado utilizando Graph2Test

Es posible modificar manualmente el fichero `feature` obtenido y relanzar el programa desde el punto de conversión de escenarios a esqueleto de código. Esto aparece reflejado en el anexo B.4. Si modificamos el fichero `feature` de manera que su contenido sea el que aparece en la Figura 36, el resultado de ejecutar Cucumber sería el que aparece reflejado en la Figura 36.

```
1: You can implement missing steps with the snippets below:
2:
3: @Given("^I have a CalculatorActivity$")
4: public void i_have_a_CalculatorActivity() throws Throwable {
5:     // Write code here that turns the phrase above into concrete actions
6:     throw new PendingException();
7: }
8:
9: @When("^I press (\\d+)$")
10: public void i_press(int arg1) throws Throwable {
11:     // Write code here that turns the phrase above into concrete actions
12:     throw new PendingException();
13: }
14:
15: @When("^I press \\+$")
16: public void i_press() throws Throwable {
17:     // Write code here that turns the phrase above into concrete actions
18:     throw new PendingException();
19: }
20:
21: @When("^I press =$")
22: public void i_press() throws Throwable {
23:     // Write code here that turns the phrase above into concrete actions
```

```
24:     throw new PendingException();
25: }
26:
27: @Then("^I should see (\\d+)\\.(\\d+) on the display$")
28:  public  void  i_should_see_on_the_display(int  arg1,  int  arg2)  throws
Throwable {
29:     // Write code here that turns the phrase above into concrete actions
30:     throw new PendingException();
31: }
32:
33: @When("^I press ?$")
34: public void i_press() throws Throwable {
35:     // Write code here that turns the phrase above into concrete actions
36:     throw new PendingException();
37: }
38:
39: @When("^I press x$")
40: public void i_press_x() throws Throwable {
41:     // Write code here that turns the phrase above into concrete actions
42:     throw new PendingException();
43: }
44:
45: @When("^I press /$")
46: public void i_press() throws Throwable {
47:     // Write code here that turns the phrase above into concrete actions
48:     throw new PendingException();
49: }
50:
51: @Then("^I should see Infinity on the display$")
52: public void i_should_see_Infinity_on_the_display() throws Throwable {
53:     // Write code here that turns the phrase above into concrete actions
54:     throw new PendingException();
55: }
```

Figura 38. Esqueleto obtenido mediante Cucumber utilizando el fichero feature de la Figura 36

A.4. Espresso

Espresso es un *framework* mediante el cual se pueden crear pruebas de interfaz de usuario para simular interacciones entre un usuario y una aplicación Android. Esta herramienta sincroniza automáticamente las acciones de prueba con la interfaz de usuario de la aplicación a probar. También se asegura que la aplicación está inicializada antes de que se ejecuten las pruebas, además de esperar a que todas las actividades que se ejecutan en segundo plano y que han sido llamadas por la aplicación hayan terminado.

Espresso tiene básicamente tres tipos de componentes:

- ViewMatchers. Permiten encontrar la vista (componente gráfico de la interfaz de usuario) en la jerarquía de la vista actual.
- ViewActions. Permiten realizar acciones en las vistas.
- ViewAssertions. Permiten confirmar el estado de una vista.

Los cuales siguen el siguiente patrón para crear el código de pruebas:

```
1:   onView(ViewMatcher)
2:        .perform(ViewAction)
3:              .check(ViewAssertion);
```

Figura 39. Estructura de código Espresso

En la página web de Espresso hay una tabla de referencia [26] con los posibles elementos a utilizar. Como se ha mencionado anteriormente, en la Figura 13 se mostraba el diagrama de clases del emparejador (clase `Matcher`) utilizado. En ella, aparecen las clases asociadas al emparejador, las cuales coinciden en nombre con los componentes básicos de Cucumber. Se puede comprobar que se han incluído varios de ellos en Graph2Test, seleccionado algunos de ellos para su uso actual y permitiendo así una expansión de la funcionalidad de Graph2Test. A continuación, en la Tabla 5, se mostrarán los utilizados actualmente en Graph2Test. Se han utilizado enumeraciones para realizar la correspondencia con el código, evitando así posibles errores de escritura.

ViewMatchers		ViewActions		ViewAssertions	
WITH_ID	withId(__DATA__)	CLICK	click()	MATCHES	matches(__DATA__)
WITH_TEXT	withText(__DATA__)	DOUBLE_CLICK	doubleClick()		
		LONG_CLICK	longClick()		

Tabla 5. Listado de elementos Espresso incluídos en Graph2Test

Por ejemplo, si queremos pulsar un botón en pantalla con el texto 5, el código correspondiente sería el que aparece en la Figura 40.

```
1:   onView(withText("5")).perform(click());
```

Figura 40. Código Espresso para pulsar un botón

Anexo B. Sintaxis básica. Uso de Graph2Test

Para poder utilizar correctamente Graph2Test es necesario seguir una mínima sintaxis. A continuación, se mostrará la sintaxis tanto para crear diagramas de entrada de GraphWalker con YEd Graph Editor como para que se pueda realizar una traducción de Cucumber a Espresso lo más acertada posible.

B.1. YEd Graph Editor y GraphWalker

Para gestionar los nombres de las aristas y nodos de los diagramas y transformarlos a sentencias Gherkin/Cucumber se ha tomado el carácter "_" como elemento separador de cada palabra y así poder escribir frases dentro del diagrama. Por ejemplo, si queremos poner la frase "Then I should see number 5 on the display" debemos escribir "Then_I_should_see_number_5_on_the_display" como nombre del elemento, en la primera línea de la arista o nodo.

Hay que tener en cuentan que al utilizar el guion bajo como separador de palabras no se podrá utilizar el mismo como parte del nombre de una variable, ya que si se utilizase se tomaría como un conjunto de palabras independientes y no se realizaría una correcta traducción. Por ejemplo, en lugar de llamar a una variable "button_number_5" se le puede poner como nombre "buttonNumber5" y así no interferiría con la sintaxis del traductor. Esto será explicado en profundidad en el apartado de Espresso, en el Anexo B.3 .

También es necesario comprobar que no se incluyen caracteres de tipo "<" ni ">". Este tipo de caracteres es utilizado por GraphWalker como símbolos de mayor y menor, puesto que es posible el uso de variables dentro de los grafos para priorizar el recorrido por un nodo en concreto antes que por otro. Graph2Test realizará el reconocimiento de variables e incluirá dichos caracteres, si es necesario, en el momento de la generación del fichero `feature`. Esto aparece recogido más adelante, en el apartado de Cucumber y Gherkin en el Anexo B.2.

Haciendo referencia a dichas variables, es posible generar una acción de salida dada una acción de entrada (una transición entre estados como se modelan normalmente en los diagramas de estados), pero únicamente es útil para hacer modificaciones de valores en dichas variables; es decir, no es posible nombrar una arista de manera "acciónEntrada / acciónSalida" si la

acción de salida no modifica el valor de una variable definida previamente dentro del grafo.

B.2. Cucumber y Gherkin

Para el uso de éstos programas no hay utilizar una sintaxis muy especial más allá de la propia solicitada por el lenguaje Gherkin. El lenguaje utilizado para el reconocimiento de palabras es el inglés, por tanto, hay que tenerlo en cuenta en el momento de nombrar los nodos/aristas de GraphWalker y al modificar los ficheros `feature` de Cucumber.

Como requisitos para el uso de Gherkin con Graph2Test hay que destacar que no se pueden utilizar símbolos de tipo "<" ni ">" dentro del diagrama de GraphWalker, a pesar de ser utilizados por Cucumber para nombrar variables en los escenarios múltiples. Para diferenciar entre un identificador, un número y una variable se hará uso de las palabras clave `ID` y `DISPLAY`, las cuales indican que la siguiente palabra hace referencia al identificador de un elemento.

También es necesario indicar que sólo puede aparecer una acción por línea, ya que, en caso contrario, el comportamiento del programa en el reconocimiento de las palabras sería impredecible. En la parte superior de la Tabla 6 vemos que aparecen dos acciones en la misma línea unidas mediante la palabra `AND`. En ese caso, Graph2Test no sabría que se trata de dos acciones distintas y tan solo escribiría el código asociado a una de ellas. En la parte inferior de la tabla, podemos ver que se ha generado correctamente el código al separar ambas acciones y ponerlas cada una en una línea distinta.

When I press button with ID button1 and I press button with ID button2 ↓ `onView(withId(button2)).perform(click());`
When I press button with ID button1 And I press button with ID button2 ↓ `onView(withId(button2)).perform(click());` `onView(withId(button2)).perform(click());`

Tabla 6. Ejemplos de comportamiento de Graph2Test al traducir el texto de los escenarios a código Espresso.

B.3. Espresso

Las acciones más básicas a realizar por el usuario se pueden resumir en pulsar un elemento, comprobar con resultados visuales la ejecución de dichas acciones y deslizar el dedo sobre la pantalla para ver elementos que no se encuentran en el campo de visión actual.

En este TFG se tendrán en cuenta las dos primeras acciones mencionadas anteriormente para realizar una traducción parcial de textos en lenguaje Gherkin (Cucumber) a acciones Espresso (código Java). Por ello, se ha realizado una tabla con las equivalencias que aparecen en la Tabla 7.

Palabra Cucumber		Código Espresso	Palabra Cucumber		Código Espresso
M	ID	withId(__DATA__)	S	SEE	matches(__DATA__)
M	TEXT	withText(__DATA__)	M	SHOULD	SKIP_SHOULD
A	PRESS	click()		ON	SKIP_ON
A	CLICK	click()		THE	SKIP_THE
A	TAP	click()		A	SKIP_A
A	DOUBLE	doubleClick()		AN	SKIP_AN
A	LONG	longClick()		IN	SKIP_IN
	GIVEN	SKIP_GIVEN		I	SKIP_I
	WHEN	SKIP_WHEN		YOU	SKIP_YOU
	THEN	SKIP_THEN		HE	SKIP_HE
	AND	SKIP_AND		SHE	SKIP_SHE
	BUTTON	SKIP_BUTTON		WE	SKIP_WE
	LABEL	SKIP_LABEL		US	SKIP_US
M	DISPLAY	withId(__DATA__)		THEY	SKIP_THEY
	WITH	SKIP_WITH			

Tabla 7. Traducción entre palabras de las sentencias Cucumber a código Espresso.

Siendo M-ViewMatcher, A-ViewAction, S-ViewAssertion (cambiando `__DATA__` por el código que corresponda, si se tiene) y los SKIP palabras que son ignoradas. Se tienen en cuenta palabras no relevantes para de esta manera tomarlas como ID/nombre/dato a sustituir los textos tipo `__DATA__`.

En determinadas ocasiones, el texto `__DATA__` se modificará automáticamente por el elemento correspondiente que haya sido leído. Al realizar la lectura de una línea de texto de los escenarios de Cucumber se realiza un paso a letras mayúsculas de toda la línea actual, pudiendo así realizarse una comparación

directa con las palabras almacenadas, las cuales se encuentran en mayúsculas.

Por defecto se realizará una búsqueda por texto (`withText`), pero se puede configurar la búsqueda por identificadores indicándolo mediante el uso de la palabra `ID` y siendo la siguiente palabra el identificador correspondiente. En la Tabla 8 podemos ver ejemplos de traducciones de texto en lenguaje Gherkin a código Espresso.

When I press num ↓ `onView(withText(num)).perform(click());`
When I press 5 ↓ `onView(withText(5)).perform(click());`
When I press number 5 ↓ `onView(withText("5")).perform(click());`
When I press button with ID buttonNumber5 ↓ `onView(withId(R.id.buttonNumber5)).perform(click());`
Then I should see number 5 on the display dispTxt ↓ `onView(withId(R.id.dispTxt)).check(matches(withText("5")));`
Then I should see num on the display with ID dispTxt ↓ `onView(withId(R.id.dispTxt)).check(matches(withText(num)));`

Tabla 8. Traducciones de sentencias Cucumber a código Espresso.

B.4. Interfaz gráfica de Graph2Test

Se ha diseñado una pequeña interfaz gráfica, que aparece reflejada en la Figura 41, para facilitar el uso de Graph2Test. Con ella, se realiza la gestión de los datos de entrada. Es posible seleccionar desde qué punto se desea comenzar; desde el diagrama de estados, desde los escenarios de Cucumber o desde el esqueleto. Un elemento en común a todas las opciones es la ruta de la carpeta de salida, donde se almacenarán los ficheros generados.

Figura 41. Interfaz gráfica de Graph2Test Component

Por defecto, estará seleccionada la opción de partir desde el diagrama de estados. Si se elige esta opción es necesario introducir la ruta del diagrama y su elemento inicial, el cual es un fichero `graphml`, además de la ruta de la carpeta de salida (sin terminar en "/"). Si se dispone de un fichero `feature`, se puede utilizar dicho fichero como dato de partida. Para ello, hay que seleccionar las opcines "Partir desde otros ficheros" y "Partir desde escenarios", además de indicar la ruta donde se encuentra el fichero `feature`. Seleccionando las opciones Partir desde otros ficheros" y "Partir desde esqueleto". Esta opción permite comenzar directamente desde el rellenado parcial del esqueleto con código Espresso pero, además del fichero `java` que contiene el esqueleto y la ruta de la carpeta de salida, es necesario introducir la ruta del fichero `feature` asociado a dicho esqueleto.

Una vez presionado el botón "Lanzar", el programa ejecuta toda su lógica con la información introducida, indicando mediante un mensaje final que se han realizado las operaciones. También se permite limpiar todas las rutas de entrada mediante el botón "Limpiar datos".

Para la gestión de errores se ha utilizado la librería log4j2 [16], la cual permite la generación de un log que registre las trazas generadas por Graph2Test, almacenándolas en un fichero. La ruta del fichero será la misma que la carpeta de salida y el nombre del fichero `Graph2Test.log`

Se ha empleado la librería nativa de Java Swing para su implementación, utilizando para ello botones, cajas de texto, etiquetas y selectores de archivos y directorios. En la Figura 42 se puede ver el diagrama de clases referente a la interfaz gráfica.

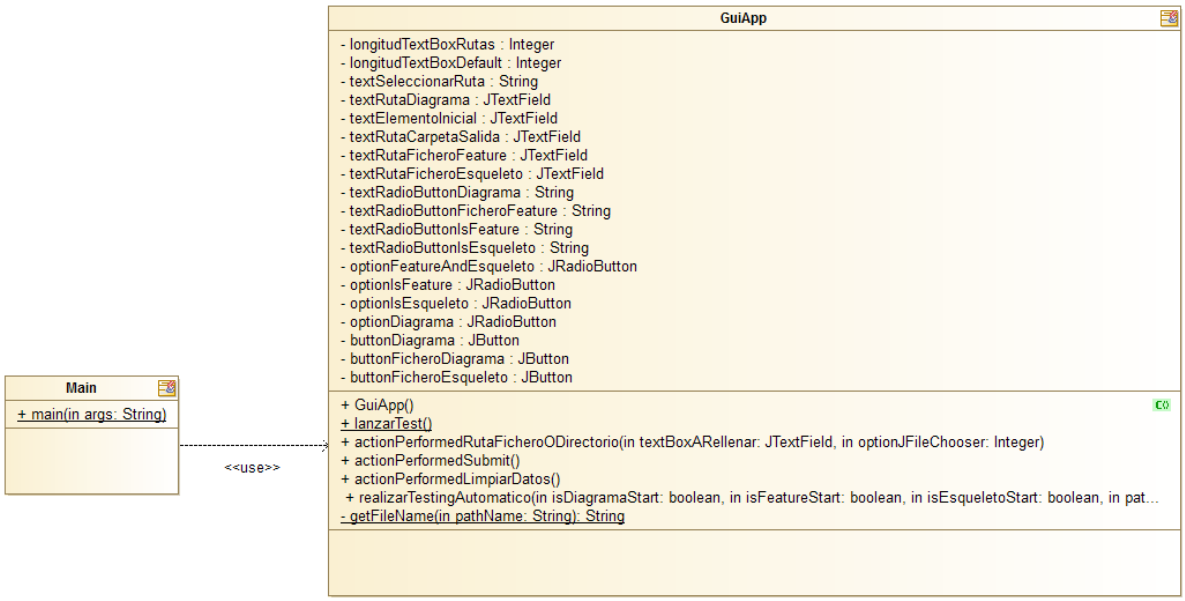


Figura 42. Diagrama de clases de la interfaz gráfica de Graph2Test

Anexo C. Resultados de la ejecución de los casos de prueba

A continuación, se muestran los resultados completos obtenidos de la ejecución de Graph2Test para los ejemplos de las aplicaciones Calculadora y Farmacia. Estos resultados son los obtenidos sin realizar modificaciones a los escenarios o al esqueleto y ejecutados partiendo del diagrama de estados.

C.1. Aplicación Calculadora

En la Figura 43 se pueden observar los escenarios para la aplicación Calculadora, resultado utilizar GraphWalker para el recorrido del diagrama inicial que aparece en la Figura 15. A partir de los escenarios, se ha obtenido mediante Cucumber el esqueleto y rellenado con código Espresso. Se puede ver la implementación parcial obtenida en el Figura 44. Para este caso, se han obtenido en total 51 escenarios de Cucumber. Al implementar estos escenarios, se hacen pruebas automatizadas en la aplicación Calculadora que ejercitan todas las transiciones de estados de estados en la misma.

```
1: Feature: Add your feature here.
2:
3:
4: Scenario: Write your scenario here.
5: Given Start
6: When init CalculatorActivity
7: Then I have a <CalculatorActivity>
8:
9: Scenario: Write your scenario here.
10: Given I should see <Operador1> on display
11: When I press ID op
12: Then I should see <OperacionYOperador1> on display
13:
14: Scenario: Write your scenario here.
15: Given I should see <OperacionYOperador1> on display
16: When I press ID d
17: Then I should see <OperacionYOperador2> on display
```

```
18:
19: Scenario: Write your scenario here.
20: Given I should see <OperacionYOperador2> on display
21: When I press ID opEq
22: Then I should see <ResultadoFinal> on display
23:
24: Scenario: Write your scenario here.
25: Given I should see <ResultadoFinal> on display
26: When I press ID d
27: Then I should see <Operador1> on display
28:
29: Scenario: Write your scenario here.
30: Given I should see <Operador1> on display
31: When I press ID op
32: Then I should see <OperacionYOperador1> on display
33:
34: Scenario: Write your scenario here.
35: Given I should see <OperacionYOperador1> on display
36: When I press ID d
37: Then I should see <OperacionYOperador2> on display
38:
39: Scenario: Write your scenario here.
40: Given I should see <OperacionYOperador2> on display
41: When I press ID op
42: Then I should see <ResultadoYOperacionNueva> on display
43:
44: Scenario: Write your scenario here.
45: Given I should see <ResultadoYOperacionNueva> on display
46: When I press ID d
47: Then I should see <OperacionYNumeroConcatenadoAlAnterior> on display
48:
49: Scenario: Write your scenario here.
50: Given I should see <OperacionYNumeroConcatenadoAlAnterior> on display
51: When I press ID opEq
52: Then I should see <ResultadoFinal> on display
53:
54: Scenario: Write your scenario here.
55: Given I should see <ResultadoFinal> on display
56: When I press ID op
57: Then I should see <ResultadoYOperacionNueva> on display
58:
```

```
Scenario: Write your scenario here.
Given I should see <ResultadoYOperacionNueva> on display
When I press ID d
Then I should see <OperacionYNumeroConcatenadoAlAnterior> on display

Scenario: Write your scenario here.
Given I should see <OperacionYNumeroConcatenadoAlAnterior> on display
When I press ID opEq
Then I should see <ResultadoFinal> on display

Scenario: Write your scenario here.
Given I should see <ResultadoFinal> on display
When I press ID d
Then I should see <Operador1> on display

Scenario: Write your scenario here.
Given I should see <Operador1> on display
When I press ID d
Then I should see <resultadoParcial> on display

Scenario: Write your scenario here.
Given I should see <resultadoParcial> on display
When I press ID opEq
Then I should see <resultadoParcial> on display

Scenario: Write your scenario here.
Given I should see <resultadoParcial> on display
When I press ID op
Then I should see <OperacionYOperador1> on display

Scenario: Write your scenario here.
Given I should see <OperacionYOperador1> on display
When I press ID d
Then I should see <OperacionYOperador2> on display

Scenario: Write your scenario here.
Given I should see <OperacionYOperador2> on display
When I press ID d
Then I should see <OperacionYNumeroConcatenadoAlAnterior> on display

Scenario: Write your scenario here.
```

```
Given I should see <OperacionYNumeroConcatenadoAlAnterior> on display
When I press ID op
Then I should see <ResultadoYOperacionNueva> on display

Scenario: Write your scenario here.
Given I should see <ResultadoYOperacionNueva> on display
When I press ID opEq
Then I should see <ResultadoYOperacionAnterior> on display

Scenario: Write your scenario here.
Given I should see <ResultadoYOperacionAnterior> on display
When I press ID d
Then I should see <OperacionYNumeroConcatenadoAlAnterior> on display

Scenario: Write your scenario here.
Given I should see <OperacionYNumeroConcatenadoAlAnterior> on display
When I press ID opEq
Then I should see <ResultadoFinal> on display

Scenario: Write your scenario here.
Given I should see <ResultadoFinal> on display
When I press ID d
Then I should see <Operador1> on display

Scenario: Write your scenario here.
Given I should see <Operador1> on display
When I press ID op
Then I should see <OperacionYOperador1> on display

Scenario: Write your scenario here.
Given I should see <OperacionYOperador1> on display
When I press ID opEq
Then I should see <ResultadoYOperacionAnterior> on display

Scenario: Write your scenario here.
Given I should see <ResultadoYOperacionAnterior> on display
When I press ID d
Then I should see <OperacionYNumeroConcatenadoAlAnterior> on display

Scenario: Write your scenario here.
Given I should see <OperacionYNumeroConcatenadoAlAnterior> on display
```

```
When I press ID d
Then I should see <OperacionYNumeroConcatenadoAlAnterior> on display

Scenario: Write your scenario here.
Given I should see <OperacionYNumeroConcatenadoAlAnterior> on display
When I press ID opEq
Then I should see <ResultadoFinal> on display

Scenario: Write your scenario here.
Given I should see <ResultadoFinal> on display
When I press ID d
Then I should see <Operador1> on display

Scenario: Write your scenario here.
Given I should see <Operador1> on display
When I press ID opEq
Then I should see <Operador1> on display

Scenario: Write your scenario here.
Given I should see <Operador1> on display
When I press ID op
Then I should see <OperacionYOperador1> on display

Scenario: Write your scenario here.
Given I should see <OperacionYOperador1> on display
When I press ID opEq
Then I should see <ResultadoYOperacionAnterior> on display

Scenario: Write your scenario here.
Given I should see <ResultadoYOperacionAnterior> on display
When I press ID opEq
Then I should see <ResultadoYOperacionAnterior> on display

Scenario: Write your scenario here.
Given I should see <ResultadoYOperacionAnterior> on display
When I press ID d
Then I should see <OperacionYNumeroConcatenadoAlAnterior> on display

Scenario: Write your scenario here.
Given I should see <OperacionYNumeroConcatenadoAlAnterior> on display
When I press ID opEq
```

```
Then I should see <ResultadoFinal> on display

Scenario: Write your scenario here.
Given I should see <ResultadoFinal> on display
When I press ID d
Then I should see <Operador1> on display

Scenario: Write your scenario here.
Given I should see <Operador1> on display
When I press ID op
Then I should see <OperacionYOperador1> on display

Scenario: Write your scenario here.
Given I should see <OperacionYOperador1> on display
When I press ID op
Then I should see <OperacionYOperador1> on display

Scenario: Write your scenario here.
Given I should see <OperacionYOperador1> on display
When I press ID d
Then I should see <OperacionYOperador2> on display

Scenario: Write your scenario here.
Given I should see <OperacionYOperador2> on display
When I press ID opEq
Then I should see <ResultadoFinal> on display

Scenario: Write your scenario here.
Given I should see <ResultadoFinal> on display
When I press ID opEq
Then I should see <ResultadoFinal> on display

Scenario: Write your scenario here.
Given I should see <ResultadoFinal> on display
When I press ID op
Then I should see <ResultadoYOperacionNueva> on display

Scenario: Write your scenario here.
Given I should see <ResultadoYOperacionNueva> on display
When I press ID opEq
Then I should see <ResultadoYOperacionAnterior> on display
```

```
223:
224: Scenario: Write your scenario here.
225: Given I should see <ResultadoYOperacionAnterior> on display
226: When I press ID op
227: Then I should see <ResultadoYOperacionNueva> on display
228:
229: Scenario: Write your scenario here.
230: Given I should see <ResultadoYOperacionNueva> on display
231: When I press ID op
232: Then I should see <ResultadoYOperacionNueva> on display
233:
234: Scenario: Write your scenario here.
235: Given I should see <ResultadoYOperacionNueva> on display
236: When I press ID d
237: Then I should see <OperacionYNumeroConcatenadoAlAnterior> on display
238:
239: Scenario: Write your scenario here.
240: Given I should see <OperacionYNumeroConcatenadoAlAnterior> on display
241: When I press ID opEq
242: Then I should see <ResultadoFinal> on display
243:
244: Scenario: Write your scenario here.
245: Given I should see <ResultadoFinal> on display
246: When I press ID d
247: Then I should see <Operador1> on display
248:
249: Scenario: Write your scenario here.
250: Given I should see <Operador1> on display
251: When I press ID d
252: Then I should see <resultadoParcial> on display
253:
254: Scenario: Write your scenario here.
255: Given I should see <resultadoParcial> on display
256: When I press ID d
257: Then I should see <resultadoParcial> on display
```

Figura 43. Resultado de la creación del fichero feature para la aplicación Calculadora, después de recorrer el diagrama de estados

```
1: You can implement missing steps with the snippets below:
2:
3: @Given("^Start$")
4: public void start() throws Throwable {
5:     // Write code here that turns the phrase above into concrete actions
6:     throw new PendingException();
7: }
8:
9: @When("^init CalculatorActivity$")
10: public void init_CalculatorActivity() throws Throwable {
11:     // Write code here that turns the phrase above into concrete actions
12:     throw new PendingException();
13: }
14:
15: @Then("^I have a <CalculatorActivity>$")
16: public void i_have_a_CalculatorActivity() throws Throwable {
17:      onView(withText(CalculatorActivity));
18:     // Write code here that turns the phrase above into concrete actions
19:     throw new PendingException();
20: }
21:
22: @Given("^I have a <CalculatorActivity>$")
23: public void i_have_a_CalculatorActivity() throws Throwable {
24:      onView(withText(CalculatorActivity));
25:     // Write code here that turns the phrase above into concrete actions
26:     throw new PendingException();
27: }
28:
29: @Given("^I should see <Operador(\\d+)> on display$")
30: public void i_should_see_Operador_on_display(int arg1) throws Throwable {
31:      onView(withId(__DATA__)).check(matches(withText(Operador1)));
32:     // Write code here that turns the phrase above into concrete actions
33:     throw new PendingException();
34: }
35:
36: @When("^I press ID op$")
37: public void i_press_ID_op() throws Throwable {
38:      onView(withId(R.id.op)).perform(click());
39:     // Write code here that turns the phrase above into concrete actions
40:     throw new PendingException();
41: }
```

```
42:
43: @Then("^I should see <OperacionYOperador(\\d+)> on display$")
44: public void i_should_see_OperacionYOperador_on_display(int arg1) throws Throwable
{
45: onView(withId(__DATA__)).check(matches(withText(OperacionYOperador1)));
46: onView(withId(__DATA__)).check(matches(withText(OperacionYOperador2)));
47:     // Write code here that turns the phrase above into concrete actions
48:     throw new PendingException();
49: }
50:
51: @Given("^I should see <OperacionYOperador(\\d+)> on display$")
52: public void i_should_see_OperacionYOperador_on_display(int arg1) throws Throwable
{
53:      onView(withId(__DATA__)).check(matches(withText(OperacionYOperador1)));
54:      onView(withId(__DATA__)).check(matches(withText(OperacionYOperador2)));
55:     // Write code here that turns the phrase above into concrete actions
56:     throw new PendingException();
57: }
58:
59: @When("^I press ID d$")
60: public void i_press_ID_d() throws Throwable {
61:      onView(withId(R.id.d)).perform(click());
62:     // Write code here that turns the phrase above into concrete actions
63:     throw new PendingException();
64: }
65:
66: @When("^I press ID opEq$")
67: public void i_press_ID_opEq() throws Throwable {
68:      onView(withId(R.id.opEq)).perform(click());
69:     // Write code here that turns the phrase above into concrete actions
70:     throw new PendingException();
71: }
72:
73: @Then("^I should see <ResultadoFinal> on display$")
74: public void i_should_see_ResultadoFinal_on_display() throws Throwable {
75:       onView(withId(__DATA__)).check(matches(withText(ResultadoFinal)));
76:     // Write code here that turns the phrase above into concrete actions
77:     throw new PendingException();
78: }
79:
80: @Given("^I should see <ResultadoFinal> on display$")
81: public void i_should_see_ResultadoFinal_on_display() throws Throwable {
```

```
82:       onView(withId(__DATA__)).check(matches(withText(ResultadoFinal)));
83:      // Write code here that turns the phrase above into concrete actions
84:      throw new PendingException();
85: }
86:
87: @Then("^I should see <Operador(\\d+)> on display$")
88: public void i_should_see_Operador_on_display(int arg1) throws Throwable {
89:       onView(withId(__DATA__)).check(matches(withText(Operador1)));
90:      // Write code here that turns the phrase above into concrete actions
91:      throw new PendingException();
92: }
93:
94: @Then("^I should see <ResultadoYOperacionNueva> on display$")
95: public void i_should_see_ResultadoYOperacionNueva_on_display() throws Throwable {
96:       onView(withId(__DATA__)).check(matches(withText(ResultadoYOperacionNueva)));
97:      // Write code here that turns the phrase above into concrete actions
98:      throw new PendingException();
99: }
100:
101: @Given("^I should see <ResultadoYOperacionNueva> on display$")
102: public void i_should_see_ResultadoYOperacionNueva_on_display() throws Throwable{
103:      onView(withId(__DATA__)).check(matches(withText(ResultadoYOperacionNueva)));
104:      // Write code here that turns the phrase above into concrete actions
105:      throw new PendingException();
106: }
107:
108: @Then("^I should see <OperacionYNumeroConcatenadoAlAnterior> on display$")
109: public void i_should_see_OperacionYNumeroConcatenadoAlAnterior_on_display()
         throws Throwable {
110:      onView(withId(__DATA__)).check(matches(
                  withText(OperacionYNumeroConcatenadoAlAnterior)));
111:      // Write code here that turns the phrase above into concrete actions
112:      throw new PendingException();
113: }
114:
115: @Given("^I should see <OperacionYNumeroConcatenadoAlAnterior> on display$")
116: public void i_should_see_OperacionYNumeroConcatenadoAlAnterior_on_display()
         throws Throwable {
117:      onView(withId(__DATA__)).check(matches(
                  withText(OperacionYNumeroConcatenadoAlAnterior)));
118:      // Write code here that turns the phrase above into concrete actions
```

```
119:     throw new PendingException();
120: }
121:
122: @Then("^I should see <resultadoParcial> on display$")
123: public void i_should_see_resultadoParcial_on_display() throws Throwable {
124:      onView(withId(__DATA__)).check(matches(withText(resultadoParcial)));
125:     // Write code here that turns the phrase above into concrete actions
126:     throw new PendingException();
127: }
128:
129: @Given("^I should see <resultadoParcial> on display$")
130: public void i_should_see_resultadoParcial_on_display() throws Throwable {
131:     onView(withId(__DATA__)).check(matches(withText(resultadoParcial)));
132:     // Write code here that turns the phrase above into concrete actions
133:     throw new PendingException();
134: }
135:
136: @Then("^I should see <ResultadoYOperacionAnterior> on display$")
137: public void i_should_see_ResultadoYOperacionAnterior_on_display()
         throws Throwable {
138:     onView(withId(__DATA__)).check(matches(
                 withText(ResultadoYOperacionAnterior)));
139:     // Write code here that turns the phrase above into concrete actions
140:     throw new PendingException();
141: }
142:
143: @Given("^I should see <ResultadoYOperacionAnterior> on display$")
144: public void i_should_see_ResultadoYOperacionAnterior_on_display()
         throws Throwable {
145:     onView(withId(__DATA__)).check(matches(
                 withText(ResultadoYOperacionAnterior)));
146:     // Write code here that turns the phrase above into concrete actions
147:     throw new PendingException();
148: }
```

Figura 44. Resultado final de la ejecución de Graph2Test para la aplicación Calculadora. Rellenado del esqueleto con código Espresso

C.2. Aplicación Farmacias

Para el diagrma de la aplicación Farmacias que aparece en la Figura 24, podemos ver en la Figura 45 los escenarios obtenidos y en la Figura 46 el código parcialmente rellenado. Para este caso, se han obtenido un total de 37 estados. Al implementar estos escenarios, se hacen pruebas automatizadas en la aplicación Farmacias que ejercitan todas las transiciones de estados de estados en la misma.

```
1: Feature: Add your feature here.
2:
3:
4: Scenario: Write your scenario here.
5: Given Start
6: When Init FarmaciasAhoraZGZ
7: Then I have a <FarmaciasAhoraZGZActivity>
8:
9: Scenario: Write your scenario here.
10: Given I have a <FarmaciasAhoraZGZActivity>
11: When CargarDatosFarmaciasAbiertas
12: Then I should see <mapaFarmaciasAbiertas>
13:
14: Scenario: Write your scenario here.
15: Given I should see <mapaFarmaciasAbiertas>
16: When I press <botonMenu>
17: Then I should see <listaOpciones>
18:
19: Scenario: Write your scenario here.
20: Given I should see <listaOpciones>
21: When I press <opcionMostrarFarmaciasCerradas>
22: Then I should see <mapaTodasLasFarmacias>
23:
24: Scenario: Write your scenario here.
25: Given I should see <mapaTodasLasFarmacias>
26: When I press <botonMenu>
27: Then I should see <listaOpciones>
28:
29: Scenario: Write your scenario here.
30: Given I should see <listaOpciones>
31: When I press <opcionMostrarFarmaciasAbiertas>
32: Then I should see <mapaFarmaciasAbiertas>
```

```
33:
34: Scenario: Write your scenario here.
35: Given I should see <mapaFarmaciasAbiertas>
36: When I press <botonMenu>
37: Then I should see <listaOpciones>
38:
39: Scenario: Write your scenario here.
40: Given I should see <listaOpciones>
41: When I press <opcionMostrarFarmaciasCerradas>
42: Then I should see <mapaTodasLasFarmacias>
43:
44: Scenario: Write your scenario here.
45: Given I should see <mapaTodasLasFarmacias>
46: And I should see <listadoFarmacias>
47: When I press <botonInfo>
48: Then I should see <detalleFarmaciaSeleccionada>
49:
50: Scenario: Write your scenario here.
51: Given I should see <detalleFarmaciaSeleccionada>
52: When I press <botonCerrarOAtras>
53: Then I should see <listadoFarmacias>
54:
55: Scenario: Write your scenario here.
56: Given I should see <listadoFarmacias>
57: When I press <botonMapa>
58: Then I should see <mapaTodasLasFarmacias>
59:
60: Scenario: Write your scenario here.
61: Given I should see <mapaTodasLasFarmacias>
62: When I press <botonMenu>
63: Then I should see <listaOpciones>
64:
65: Scenario: Write your scenario here.
66: Given I should see <listaOpciones>
67: When I press <botonAtras>
68: Then I should see <mapaTodasLasFarmacias>
69:
70: Scenario: Write your scenario here.
71: Given I should see <mapaTodasLasFarmacias>
72: When I press <botonMenu>
73: Then I should see <listaOpciones>
```

```
74:
75: Scenario: Write your scenario here.
76: Given I should see <listaOpciones>
77: When I press <opcionRefresco>
78: Then I should see <mapaTodasLasFarmacias>
79:
80: Scenario: Write your scenario here.
81: Given I should see <mapaTodasLasFarmacias>
82: When I press <botonMenu>
83: Then I should see <listaOpciones>
84:
85: Scenario: Write your scenario here.
86: Given I should see <listaOpciones>
87: When I press <opcionMostrarFarmaciasAbiertas>
88: Then I should see <mapaFarmaciasAbiertas>
89:
90: Scenario: Write your scenario here.
91: Given I should see <mapaFarmaciasAbiertas>
92: And I should see <listadoFarmacias>
93: When I press <botonInfo>
94: Then I should see <detalleFarmaciaSeleccionada>
95:
96: Scenario: Write your scenario here.
97: Given I should see <detalleFarmaciaSeleccionada>
98: When I press <botonCerrarOAtras>
99: Then I should see <listadoFarmacias>
100:
101: Scenario: Write your scenario here.
102: Given I should see <listadoFarmacias>
103: When I press <botonMapa>
104: Then I should see <mapaFarmaciasAbiertas>
105:
106: Scenario: Write your scenario here.
107: Given I should see <mapaFarmaciasAbiertas>
108: When I press <botonMenu>
109: Then I should see <listaOpciones>
110:
111: Scenario: Write your scenario here.
112: Given I should see <listaOpciones>
113: When I press <opcionRefresco>
114: Then I should see <mapaFarmaciasAbiertas>
```

```
115:
116: Scenario: Write your scenario here.
117: Given I should see <mapaFarmaciasAbiertas>
118: When I press <botonMenu>
119: Then I should see <listaOpciones>
120:
121: Scenario: Write your scenario here.
122: Given I should see <listaOpciones>
123: When I press <opcionMostrarFarmaciasCerradas>
124: Then I should see <mapaTodasLasFarmacias>
125:
126: Scenario: Write your scenario here.
127: Given I should see <mapaTodasLasFarmacias>
128: When I press <botonMapa>
129: Then I should see <mapaTodasLasFarmacias>
130:
131: Scenario: Write your scenario here.
132: Given I should see <mapaTodasLasFarmacias>
133: When I press <botonMenu>
134: Then I should see <listaOpciones>
135:
136: Scenario: Write your scenario here.
137: Given I should see <listaOpciones>
138: When I press <opcionMostrarFarmaciasAbiertas>
139: Then I should see <mapaFarmaciasAbiertas>
140:
141: Scenario: Write your scenario here.
142: Given I should see <mapaFarmaciasAbiertas>
143: When I press <botonMapa>
144: Then I should see <mapaFarmaciasAbiertas>
145:
146: Scenario: Write your scenario here.
147: Given I should see <mapaFarmaciasAbiertas>
148: When I press <botonMenu>
149: Then I should see <listaOpciones>
150:
151: Scenario: Write your scenario here.
152: Given I should see <listaOpciones>
153: When I press <botonAtras>
154: Then I should see <mapaFarmaciasAbiertas>
155:
```

```
156: Scenario: Write your scenario here.
157: Given I should see <mapaFarmaciasAbiertas>
158: When I press <botonMenu>
159: Then I should see <listaOpciones>
160:
161: Scenario: Write your scenario here.
162: Given I should see <listaOpciones>
163: When I press <opcionMostrarFarmaciasCerradas>
164: Then I should see <mapaTodasLasFarmacias>
165:
166: Scenario: Write your scenario here.
167: Given I should see <mapaTodasLasFarmacias>
168: When I press <botonMenu>
169: Then I should see <listaOpciones>
170:
171: Scenario: Write your scenario here.
172: Given I should see <listaOpciones>
173: When I press <botonMenu>
174: Then I should see <mapaTodasLasFarmacias>
175:
176: Scenario: Write your scenario here.
177: Given I should see <mapaTodasLasFarmacias>
178: When I press <botonMenu>
179: Then I should see <listaOpciones>
180:
181: Scenario: Write your scenario here.
182: Given I should see <listaOpciones>
183: When I press <opcionMostrarFarmaciasAbiertas>
184: Then I should see <mapaFarmaciasAbiertas>
185:
186: Scenario: Write your scenario here.
187: Given I should see <mapaFarmaciasAbiertas>
188: When I press <botonMenu>
189: Then I should see <listaOpciones>
190:
191: Scenario: Write your scenario here.
192: Given I should see <listaOpciones>
193: When I press <botonMenu>
194: Then I should see <mapaFarmaciasAbiertas>
195:
```

Figura 45. Escenarios generados a partir del diagrama creado para la aplicación Farmacias

```
1: You can implement missing steps with the snippets below:
2:
3: @Given("^Start$")
4: public void start() throws Throwable {
5:     // Write code here that turns the phrase above into concrete actions
6:     throw new PendingException();
7: }
8:
9: @When("^Init FarmaciasAhoraZGZ$")
10: public void init_FarmaciasAhoraZGZ() throws Throwable {
11:     // Write code here that turns the phrase above into concrete actions
12:     throw new PendingException();
13: }
14:
15: @Then("^I have a <FarmaciasAhoraZGZActivity>$")
16: public void i_have_a_FarmaciasAhoraZGZActivity() throws Throwable {
17:     onView(withText(FarmaciasAhoraZGZActivity));
18:     // Write code here that turns the phrase above into concrete actions
19:     throw new PendingException();
20: }
21:
22: @Given("^I have a <FarmaciasAhoraZGZActivity>$")
23: public void i_have_a_FarmaciasAhoraZGZActivity() throws Throwable {
24:     onView(withText(FarmaciasAhoraZGZActivity));
25:     // Write code here that turns the phrase above into concrete actions
26:     throw new PendingException();
27: }
28:
29: @When("^CargarDatosFarmaciasAbiertas$")
30: public void cargardatosfarmaciasabiertas() throws Throwable {
31:     // Write code here that turns the phrase above into concrete actions
32:     throw new PendingException();
33: }
34:
35: @Then("^I should see <mapaFarmaciasAbiertas>$")
36: public void i_should_see_mapaFarmaciasAbiertas() throws Throwable {
37:     onView(withText(mapaFarmaciasAbiertas))
                .check(matches(withText(mapaFarmaciasAbiertas)));
38:     // Write code here that turns the phrase above into concrete actions
```

```
39:     throw new PendingException();
40: }
41:
42: @Given("^I should see <mapaFarmaciasAbiertas>$")
43: public void i_should_see_mapaFarmaciasAbiertas() throws Throwable {
44:      onView(withText(mapaFarmaciasAbiertas))
                .check(matches(withText(mapaFarmaciasAbiertas)));
45:     // Write code here that turns the phrase above into concrete actions
46:     throw new PendingException();
47: }
48:
49: @When("^I press <botonMenu>$")
50: public void i_press_botonMenu() throws Throwable {
51:      onView(withText(botonMenu)).perform(click());
52:     // Write code here that turns the phrase above into concrete actions
53:     throw new PendingException();
54: }
55:
56: @Then("^I should see <listaOpciones>$")
57: public void i_should_see_listaOpciones() throws Throwable {
58:      onView(withText(listaOpciones)).check(matches(withText(listaOpciones)));
59:     // Write code here that turns the phrase above into concrete actions
60:     throw new PendingException();
61: }
62:
63: @Given("^I should see <listaOpciones>$")
64: public void i_should_see_listaOpciones() throws Throwable {
65:      onView(withText(listaOpciones)).check(matches(withText(listaOpciones)));
66:     // Write code here that turns the phrase above into concrete actions
67:     throw new PendingException();
68: }
69:
70: @When("^I press <opcionMostrarFarmaciasCerradas>$")
71: public void i_press_opcionMostrarFarmaciasCerradas() throws Throwable {
72:      onView(withText(opcionMostrarFarmaciasCerradas)).perform(click());
73:     // Write code here that turns the phrase above into concrete actions
74:     throw new PendingException();
75: }
76:
77: @Then("^I should see <mapaTodasLasFarmacias>$")
78: public void i_should_see_mapaTodasLasFarmacias() throws Throwable {
```

```
79:      onView(withText(mapaTodasLasFarmacias))
                 .check(matches(withText(mapaTodasLasFarmacias)));
80:     // Write code here that turns the phrase above into concrete actions
81:     throw new PendingException();
82: }
83:
84: @Given("^I should see <mapaTodasLasFarmacias>$")
85: public void i_should_see_mapaTodasLasFarmacias() throws Throwable {
86:      onView(withText(mapaTodasLasFarmacias))
                 .check(matches(withText(mapaTodasLasFarmacias)));
87:     // Write code here that turns the phrase above into concrete actions
88:     throw new PendingException();
89: }
90:
91: @When("^I press <opcionMostrarFarmaciasAbiertas>$")
92: public void i_press_opcionMostrarFarmaciasAbiertas() throws Throwable {
93:       onView(withText(opcionMostrarFarmaciasAbiertas)).perform(click());
94:     // Write code here that turns the phrase above into concrete actions
95:     throw new PendingException();
96: }
97:
98: @Given("^I should see <listadoFarmacias>$")
99: public void i_should_see_listadoFarmacias() throws Throwable {
100:     onView(withText(listadoFarmacias))
                 .check(matches(withText(listadoFarmacias)));
101:     // Write code here that turns the phrase above into concrete actions
102:     throw new PendingException();
103: }
104:
105: @When("^I press <botonInfo>$")
106: public void i_press_botonInfo() throws Throwable {
107: onView(withText(botonInfo)).perform(click());
108:     // Write code here that turns the phrase above into concrete actions
109:     throw new PendingException();
110: }
111:
112: @Then("^I should see <detalleFarmaciaSeleccionada>$")
113: public void i_should_see_detalleFarmaciaSeleccionada() throws Throwable {
114:     onView(withText(detalleFarmaciaSeleccionada))
                 .check(matches(withText(detalleFarmaciaSeleccionada)));
115:     // Write code here that turns the phrase above into concrete actions
```

```
116:     throw new PendingException();
117: }
118:
119: @Given("^I should see <detalleFarmaciaSeleccionada>$")
120: public void i_should_see_detalleFarmaciaSeleccionada() throws Throwable {
121:     onView(withText(detalleFarmaciaSeleccionada))
                .check(matches(withText(detalleFarmaciaSeleccionada)));
122:     // Write code here that turns the phrase above into concrete actions
123:     throw new PendingException();
124: }
125:
126: @When("^I press <botonCerrarOAtras>$")
127: public void i_press_botonCerrarOAtras() throws Throwable {
128:     onView(withText(botonCerrarOAtras)).perform(click());
129:     // Write code here that turns the phrase above into concrete actions
130:     throw new PendingException();
131: }
132:
133: @Then("^I should see <listadoFarmacias>$")
134: public void i_should_see_listadoFarmacias() throws Throwable {
135:     onView(withText(listadoFarmacias))
                .check(matches(withText(listadoFarmacias)));
136:     // Write code here that turns the phrase above into concrete actions
137:     throw new PendingException();
138: }
139:
140: @When("^I press <botonMapa>$")
141: public void i_press_botonMapa() throws Throwable {
142:     onView(withText(botonMapa)).perform(click());
143:     // Write code here that turns the phrase above into concrete actions
144:     throw new PendingException();
145: }
146:
147: @When("^I press <botonAtras>$")
148: public void i_press_botonAtras() throws Throwable {
149:     onView(withText(botonAtras)).perform(click());
150:     // Write code here that turns the phrase above into concrete actions
151:     throw new PendingException();
152: }
153:
154: @When("^I press <opcionRefresco>$")
```

```
155: public void i_press_opcionRefresco() throws Throwable {
156:     onView(withText(opcionRefresco)).perform(click());
157:     // Write code here that turns the phrase above into concrete actions
158:     throw new PendingException();
159: }
```

Figura 46. Esqueleto de la aplicación Farmacias parcialmente rellenado por Graph2Test