MÁSTER EN INVESTIGACIÓN EN INGENIERÍA DEL SOFTWARE Y SISTEMAS INFORMÁTICOS

Trabajo de Fin de Máster
Itinerario Ingeniería del Software (Código: 31105151)

DSL para pruebas unitarias en Aplicaciones Móviles usando Appium como interprete

Alumno: H. Paúl Pasquel Galárraga
Director: Ismael Abad Cardiel

Curso Académico 2019/2020

Convocatoria Septiembre

MÁSTER EN INVESTIGACIÓN EN INGENIERÍA DEL SOFTWARE Y SISTEMAS INFORMÁTICOS

ITINERARIO: Ingeniería del Software

CÓDIGO ASIGNATURA: 31105151

TÍTULO DEL TRABAJO: DSL para pruebas Unitarias en Aplicaciones Móviles usando APPIUM como interprete

TIPO DE TRABAJO: TIPO B, propuesto por el alumno

ALUMNO: H. Paúl Pasquel Galárraga

DIRECTOR: Ismael Abad Cardiel

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MASTER

Fecha: 05/09/2020

Quién suscribe:

Autor(a): **Hugo Paúl Pasquel Galárraga**

D.N.I/N.I.E/Pasaporte.: **1001898517**

Hace constar que es la autor(a) del trabajo:

DSL para pruebas Unitarias en Aplicaciones Móviles usando APPIUM como interprete

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo. 

Hoja Autorización

IMPRESO TFDM05_AUTOR
AUTORIZACIÓN DE PUBLICACIÓN
CON FINES ACADÉMICOS

Autorización

Autorizo/amos a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del/los Autor/es

Juan del Rosal, 16 28040,
Madrid
Tel: 91 398 89 10
Fax: 91 398 89 09
www.issi.uned.es

RESUMEN

El auge de los teléfonos inteligente, en la última década, ha llevado a que cada vez más las empresas e instituciones deban replantearse sus procesos y flujo de trabajo, para hacer que sus clientes finales sean una parte fundamental de los mismos. Proveer aplicaciones móviles, pasó de ser una tendencia a una necesidad; las nuevas generaciones quieren estar y formar parte de los servicios que las empresas proveen. Un comentario o una calificación negativa en las diferentes tiendas de aplicaciones, no sólo es un indicativo de un malestar, sino puede llegar a presentarse como problemas de competitividad o eficiencia, al proveer un servicio a sus clientes. Es así que proveer una aplicación confiable, segura, y que cumpla eficazmente con las necesidades del cliente final, se vuelve un factor crítico con respecto al negocio.

El presente trabajo, inicia con un análisis y evaluación de la problemática que conlleva la construcción, y sobre todo la realización de pruebas de aplicaciones móviles. Se agrega la base teórica que conlleva la ejecución de pruebas, y se analizan varias herramientas, utilizadas como alternativa para llevarlas a cabo. En función de este análisis y evaluación, el trabajo pretende mostrar una alternativa de solución viable, a la sistematización del proceso de pruebas, a través de la implementación de un DSL de negocio, con el cual un usuario experto pueda escribir los escenarios de prueba, utilizando un lenguaje asequible a su entorno y experiencia. Antes de continuar, se debe mencionar en este punto, que para el ámbito de este trabajo se entiende como aplicación móvil, a dicha aplicación que es parte de un sistema de información, es decir, a través de la cual una empresa provee servicios o productos a sus clientes finales.

Seguidamente, en la sección desarrollo, se propone un nuevo lenguaje DSL llamado aatDSL (Android Application Testing DSL), el que está basado en una estructura similar a la propuesta por Gherkin. En este lenguaje se implementa, a manera de prototipo que evalúe su factibilidad, varias instrucciones utilizables para elaborar casos de prueba. Como motor de ejecución se propone el uso de Appium, a través de la implementación de una librería tipo API que permite abstraer ciertas características de su uso, y que además facilita la correspondencia entre las instrucciones DSL y los eventos a ejecutar a través de Appium.

Finalmente se agrega un apartado con ejemplos que muestra el uso del nuevo DSL en 3 aplicaciones, desde diferente perspectiva (pruebas a un desarrollo, pruebas a una aplicación existente y desarrollo de aplicaciones usando la metodología BDD)

El último capítulo está destinado a los criterios personales, hallazgos, conclusiones y trabajos futuros.

Tabla de Contenidos

Lista de tablas

Lista de figuras

1 Introducción

1.1 Evolución de las aplicaciones

La aparición de los teléfonos inteligentes, hace más de 10 años, ha transformado la forma en la que las empresas interactúa con sus clientes. Las aplicaciones de negocios empresariales que antes eran enfocadas en ser utilizadas por los empleados y asociados, ahora, con la "transformación digital", pasan a ser basadas en el cliente final, convirtiéndolo en el usuario principal de este tipo de aplicaciones. Así ha conllevado al aparecimiento de nuevas formas en las que las empresas brindan servicios y productos, y a la par, ha conllevado a que modifiquen sus procesos de negocio, haciendo que estos incluyan directamente al cliente final. De otro lado la competencia entre empresas se enfoca cada vez más, en la provisión de nuevos servicios, funcionalidades y productos, a través de aplicaciones móviles.

Todo esto vuelve crítico, algunos aspectos relacionados a dichas aplicaciones de software, como: la usabilidad, la seguridad y confiabilidad. Dado que no sólo está en juego la productividad que proporciona la automatización, sino además ítems como la confianza, prestigio y reputación de la empresa. Por otro lado, la construcción y mantenimiento de aplicaciones móviles, requiere ser llevado a cabo con enfoque ágil; dado su alcance, utilización, y ciclo de vida funcional (todas las aplicaciones tienen una vida útil relativamente corta), es prácticamente inviable aplicar metodologías tradicionales.

En este sentido, proveer mecanismos que permitan realizar pruebas sobre aplicativos móviles de forma ágil, o incluso se pueda contar con herramientas que permitan enforcar el desarrollo a la funcionalidad requerida (base de las herramientas orientadas al desarrollo dirigido por comportamiento BDD), se vuelve primordial para el desarrollo o mejoras en los sistemas de información.

1.2 Propuesta de solución

En la siguiente disertación, proponemos como hipótesis, la búsqueda de un DSL que permita agilizar el proceso de pruebas de aplicaciones móviles; en el trabajo se analizan varias alternativas de solución existentes evaluando sus funcionalidades, y enfatizando sus debilidades, las que puedan traducirse en mejoras. De otro lado para construir la solución, y facilitar la implementación, se propone el uso de herramientas y librerías existentes, en este caso Appium y Xtext.

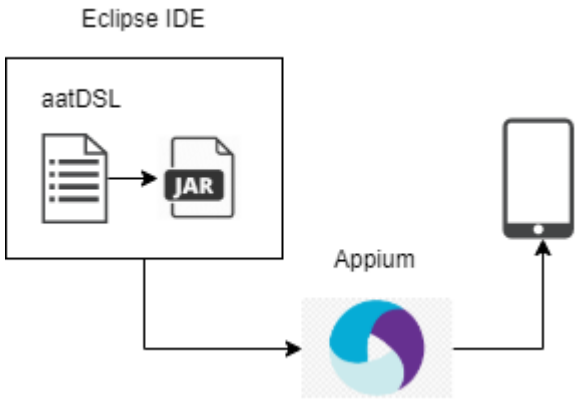


Figura 1. Esquema de la solución propuesta

Como se explica en la figura anterior, la intención es contar con IDE de desarrollo, en el que un usuario experto del negocio, escribe el escenario de prueba, usando el DSL propuesto. Haciendo uso de componentes de transformación, estos scripts se trasladan a código ejecutable de forma automática, y a continuación el usuario los ejecuta.

Internamente la traslación de los scripts a código ejecutable, pasará a hacer uso de una librería tipo API, es decir, el código ejecutable hace referencia a los elementos de funcionalidad construidos en dicha librería. Finalmente, la librería API utiliza la funcionalidad Appium para convertir las instrucciones descritas en el DSL a eventos ejecutados en un emulador de dispositivos Android.

1.3 Estructura del documento

Con la finalidad de llegar a obtener y demostrar el uso de un DSL para pruebas unitarias, se considera la siguiente estructura del documento:

En el capítulo dos, se propone conformar el marco teórico sobre el cual se basa la resolución del problema propuesto. Inicialmente se analiza la construcción de aplicaciones Android, las implicaciones y referencias a la ejecución de pruebas, además como se indica anteriormente, en este capítulo se evalúan algunas de las herramientas disponibles, y partiendo de su alcance, se pueda evidenciar posibles mejoras a proponer como parte del trabajo. En este capítulo también se aborda de manera general las características de las herramientas que se van a utilizar Appium y Xtext.

El capítulo tres se dedica al desarrollo en sí de los aspectos relacionados con la solución del problema propuesto, colocando los puntos de análisis, diseño e implementación de los componentes de software requeridos para dar forma al DSL. Con el objeto de tener un marco de trabajo para la elaboración de la solución, se usa un primer ejemplo de aplicación de negocio, la cual implementa funcionalidades sencillas que pueden ser utilizadas para explicar y elaborar en detalle los pasos requeridos en la implementación del DSL propuesto.

En el capítulo cuatro, a manera de ejemplo de utilización, se añaden 2 ejemplos más, el primero evalúa el uso del DSL para realizar pruebas en una aplicación existente, y el segundo ejemplo propone el uso del DSL en el desarrollo de aplicaciones con enfoque BDD.

En el último capítulo se expone el criterio personal, los hallazgos, las mejoras y las conclusiones del trabajo, así como varias puntos o ítems que podrían usarse dentro de trabajos futuros relacionados con el tema de la disertación e investigación realizada.

2 Problema

2.1 Descripción del Problema

Proveer servicios de calidad, eficientes, y a tiempo, es una de las metas principales de las empresas en los tiempos actuales; cada vez más las aplicaciones móviles permiten lograr este objetivo, en consecuencia, los clientes finales se conviertan: en los usuarios de las aplicaciones, o forman parte importante en los procesos empresariales, y ya no únicamente los empleados o asociados de la empresa.

Entorno a esta realidad, existen varios trabajos en los que versan sobre la ejecución de pruebas en aplicaciones móviles, así [1] se propone la creación de una herramienta MobileTest, que sea capaz de generar pruebas del tipo caja negra sobre aplicaciones de software en teléfonos inteligentes, con el objetivo de poder automatizarlas. También, existen estudios comparativos, que permiten evaluar las diferentes técnicas de pruebas automatizadas, con la finalidad de determinar cuál podría ser la más adecuada con respecto a la aplicación construida, como es el caso de [2],en el que se propone un marco de referencia para lograr este objetivo.

Inclusive en el ámbito de este problema, se han planteado soluciones enfocadas a la elaboración de aplicaciones móviles bajo marcos de trabajo orientadas a expertos de negocios, que requiera mínimos conocimientos relacionados a la construcción de aplicaciones, como es el caso de [3]. Claro está que dada las características esperadas de una aplicación móvil su real aplicabilidad queda en duda (manejo de internet de las cosas, información del contexto, etc.), y las posibilidades de utilización de este tipo de marco de trabajo sería muy limitadas.

Muchos son los retos y problemas que se presentan al hablar del desarrollo de aplicaciones móviles [4] [5], van desde temas relacionados con la funcionalidad o servicios prestado como: la experiencia de usuario o un mayor rango de dispositivo compatibles, hasta ítems

que se relación con la productividad: reutilización de código, conocimiento de las herramientas de construcción, seguridad. En este sentido, como se indica a continuación las metodologías clásicas, aunque aún usadas en el ámbito de desarrollo de aplicaciones móviles [6], no son las más adecuadas. Así podemos mencionar algunas de las necesidades, por las cuales, se vuelve primordial contar con mecanismos de construcción ágiles son:

- Time to Market (oportunidad de negocio), el tiempo y la urgencia para que una funcionalidad nueva sea puesta en producción es crítica. No solo el tiempo de construcción, sino la oportunidad de ser el primero en proveer, significa una ventaja altamente rentable para las diferentes industrias, en relación a su competencia.

- Evolución tecnológica, no solo las plataformas y sus correspondientes dispositivos evolucionan, también lo hacen los usuarios, los procesos y las empresas. Esto provoca el aparecimiento vertiginoso de nuevas necesidades y requerimientos que deben ser cubiertos. Para los equipos de desarrollo es imposible contar con un esquema perfecto, en el cual se pueda proveer una solución final, sin que antes se hayan realizado adecuaciones, no solo funcionales (incluso al requerimiento inicial), sino también desde el punto de vista técnico (código, interacción, despliegue) al producto de software. Lo que conlleva la necesidad de mantener un ciclo ágil de interacción entre los usuarios finales y el equipo de desarrollo, que sea capaz de ajustar los correspondiente “intentos” con una funcionalidad final, a la que se le considere en cierto modo aceptable.

- Información Contextual, de manera general, y cada vez en más medida, las aplicaciones móviles incluyen información del contexto. Esto añade muchas más variables, que de alguna manera deben ser consideradas en el desarrollo. En la información contextual también se incluye la experiencia y preferencias del usuario. Este tipo de información añade complejidad al desarrollo, ya que el sin número de combinaciones posibles se vuelve impredecible, volviendo nuevamente a la necesidad de contar con mecanismos de respuesta rápida a posibles fallos o

> escenarios más comunes que no son soportados por la aplicación. La experiencia indica que, si bien podría haber escenarios no soportados, lo imperativo es soportar al 100% los escenarios más comunes y más utilizados por el usuario final.

Como referencia de la explicación, se puede añadir el trabajo de investigación [7] en donde, se propone, como marco de solución a los problemas que se plantea en la construcción de aplicaciones móviles, el estudio y comparación de las metodologías ágiles, todas estas hacen hincapié en la importancia de proveer aplicaciones robustas, lo que implica haber seguido un proceso de verificación y aseguramiento de la calidad del software.

Como se puede observar, todas las metodologías propuestas, proponen que las pruebas sean realizadas de forma paralela al desarrollo, y no al final del ciclo de construcción. La ejecución de pruebas tempranas, es una garantía de que, por un lado, la funcionalidad cumple con los requerimientos solicitados, y por otro, la provisión de dichas funcionalidades se lo hace de manera eficiente y oportuna.

El propósito de este trabajo está enmarcado en las pruebas funcionales que permitan verificar la usabilidad del programa haciendo uso de la automatización de los escenarios de prueba, utilizando como medio de descripción de los casos de pruebas a un lenguaje de negocio DSL. Es importante mencionar que la ejecución de estas pruebas sobre las aplicaciones móviles no persigue en sí encontrar “bugs”, su finalidad, en el ámbito de competencia, es validar que la aplicación cumpla con los requerimientos y objetivos de negocio, determinando cuál es la calidad del producto que se está ofertando al cliente.

A continuación, en el documento se procede a plantear los diversos ítems que forman el marco teórico, sobre el que se funda tanto el problema como la solución propuesta.

2.2 Desarrollo de Aplicaciones Móviles

En el desarrollo de aplicaciones móviles un componente muy importante es la usabilidad, con respecto a la funcionalidad provista para el usuario o cliente final. Generalmente las aplicaciones de este tipo se orientan o se usan para brindar servicios directos a los clientes finales, permitiendo que estos interactúen con la empresa de forma directa. En esta interacción se incluye además elementos como la retroalimentación, que en términos de negocio se traduce en satisfacción y calidad no sólo del producto de software, sino del o de los servicios que se otorgan. Como se describe en [8] la complejidad del desarrollo de aplicaciones móviles se enmarca entre algunos aspectos: en las permanentes modificaciones o ajustes funcionales, o la implementación "rápida" de nuevas funcionalidades, o en alcanzar compatibilidad con el mayor número de dispositivos existentes (tamaños, sistemas operativos, etc.). Además, a las complejidades anotadas se debe indicar que las aplicaciones móviles intrínsecamente incluyen la computación basada en el contexto, donde la funcionalidad brindada involucra también información del contexto, como la localización, preferencias, clima, accesibilidad, etc.

2.2.1 Tipos de Aplicaciones Móviles

Una aplicación móvil puede haber sido desarrollado en base a uno de los 3 principales esquemas actuales:

- Nativa, cuando la aplicación se ha creados usando directamente los elementos provistos por la plataforma (SDK), la principal ventaja de este tipo de aplicación es el rendimiento.
- Web, estas aplicaciones se despliegan sobre un servidor Web al que el usuario accede a sus funcionalidades a través de explorador de internet. En este caso la principal desventaja es que la aplicación se ve impedida de acceder a los elementos propios del dispositivo, como la cámara o sensores, y además tiene una alta penalidad en términos de rendimiento.

- Hibridas, estas aplicaciones prácticamente son una combinación de los 2 mecanismo anteriores, se construyen sobre tecnología Web (JavaScript, HTML, CSS) pero corren sobre contenedores nativos, los cuales permiten además tener funciones que acceden a los recursos propios del dispositivo. Su principal objetivo es que sean agnósticos de la plataforma de ejecución, por ende, fáciles de construir. Su principal desventaja es el rendimiento y la obsolescencia prematura, es casi imposible que su desarrollo vaya a la par con los avances realizados en las respectivas plataformas.

2.2.2 Desarrollo de Aplicaciones Android

Durante la década anterior se tenían varios competidores en el mercado de las aplicaciones móviles, cada una enfocada en posicionar además de su sistema operativos, su propio hardware. Es así que para esas fechas Symbian había sido el líder, se notaba que los esquemas de negocio propuestos por Apple/iOS y Google/Android, empezaban a tomar el liderazgo principalmente al advenimiento del iPhone y la popularidad creciente de los dispositivos Android (conjuntamente a los servicios provistos por Google) [4]. Y así sucedió, actualmente los dispositivos más populares y utilizada en el mundo trabajan con iOS o Android. Android originalmente [9] era un proyecto de código abierto liderado por Andy Rubin, que fue adquirido por Google Inc. en 2005, claro, sin dejar de ser un sistema operativo de código abierto. Posteriormente, recién en 2008, Google anuncia el lanzamiento de su primera versión, junto a OHA (Open Handset Alliance) persiguen la creación de estándares abiertos para el desarrollo de aplicaciones móviles. En este mismo año el primer teléfono con el sistema operativo Android sale al mercado [10]. En el caso de Google, su objetivo con Android, no era centrarse únicamente en los teléfonos inteligentes [11], sino usar este nuevo sistema operativo en cualquier otro tipo de dispositivo, este o no clasificado dentro ámbito móvil (televisiones, aparatos domésticos, etc.). Android fue construido sobre el sistema operativo Linux. Una de las principales razones por haber seleccionado Linux es por su confiabilidad, seguridad y robustez. [8]

Figura 2. Stack Plataforma Android

La figura anterior muestra la estructura de la plataforma, en la que la base del Kernel Linux, sobre el que se ejecuta el motor de funcionalidad Android, proveyendo librerías utilizadas para acceder e interactuar con el dispositivo. [10] Sobre esta capa existe el "Framework" de aplicación, que expone una seria de clases para implementar las aplicaciones Android.

Es preciso indicar que antes de 2017, [12]el desarrollo de aplicaciones Android utilizaba, de manera oficial, Java, pero a partir de este año Google adopta como lenguaje oficial de desarrollo de aplicaciones Android a Kotlin. En la actualidad existen 10 versiones del sistema operativo. Por lo cual, se añade que uno de los retos al construir aplicaciones, es hacer que estas, sean compatibles con el mayor número de versiones y dispositivos existentes en el mercado. Al transcurso de la historia de Android, el equipo de desarrollo, ha tratado de que la plataforma provea de alguna manera funcionalidades que permitan la portabilidad hardware, esto es que una misma aplicación pueda ser desplegada en la mayor cantidad de dispositivos, sin importar la capacidad de procesamiento o tamaño físico; pero no así en relación a la compatibilidad entre versiones de Android. De hecho, cuando se crea una aplicación se tiene que configurar el ítem "minor versión", el cual indica la versión

más baja de sistema operativo en la que se permite la instalación y uso; en este contexto se debe indicar que entre menor versión se seleccione mayor será el sacrificio en cuanto a funcionalidad disponible, lo que incluye, además, temas como la seguridad, rendimiento, almacenamiento o interacción con elementos físicos del dispositivo. Uno de los últimos elementos en Android, es Jetpack [13] una seria de componentes de tipo API, y librerías, que aceleran el proceso de desarrollo de aplicaciones, sólo es compatible a partir de la versión 9.0 (Oreo) [14]

En este punto cabe clarificar que la solución DSL propuesta, deberá funcionar de la misma manera, es decir sin dependencia alguna de la versión Android, o el lenguaje de desarrollo utilizado. Las únicas restricciones a considerar serán las que tenga o nazcan del uso de las librerías Appium, descrita más adelante.

2.3 Importancia de Pruebas

Las aplicaciones móviles, permiten a las empresas estar más cerca de sus clientes finales, con este se logra brindar servicios más directos, en muchos casos a medida, y facilitando en mucho su localización o accesibilidad (ubicuidad); en este contexto, como se indica en varias publicaciones [15] [16] [17] [18], tener aplicaciones: confiables, seguras, robustas, en otras palabras adecuadamente certificadas, garantiza una mayor fidelización de sus clientes, que se refleja en temas como la confianza y reputación de la empresa.

De otro lado se debe también considerar las diferentes configuraciones de despliegue para las aplicaciones; esto es la combinación de sistema operativo y hardware existentes. Según el reporte [19] existen más de 2 mil 500 millones de dispositivos que corren en Android, además para 2020 Android es el sistema operativo más usado en dispositivos móviles [20]. Esto conlleva mayor complejidad no sólo al diseño y construcción de aplicaciones, sino también a las pruebas que deben realizarse. Sería ingenuo pensar que sea posible prever en

dichas etapas, todas las posibles combinaciones en la que una aplicación móvil podría ser instalada.

Por estas razones, es de suma importancia que las aplicaciones móviles que son provistas por una empresa trabajen de una forma adecuada; garantizando su disponibilidad y usabilidad la mayor parte del tiempo (no existe aplicación de software libre de errores).

2.3.1 Pruebas de Software

El propósito de probar un componente de software es garantizar que la funcionalidad que provee, cumple con los requerimientos indicados, de una manera eficiente, sin causar problemas o incidencias no previstas. Podemos indicar que generalmente las pruebas se realizan una vez que el código requerido ha sido implementado, pero desde hace algún tiempo atrás ha aparecidos nuevas tendencias como TDD (desarrollo basado en pruebas) o su prima hermana BDD (desarrollo basado en el comportamiento) en donde la propuesta es construir los escenarios de prueba antes que el código, para así tener una mejor claridad y resultados el momento de la construcción de una aplicación.

Si bien se persigue tener una garantía sobre las funcionalidades del software implementado, es imposible probar todas las permutaciones, en referencia a los escenarios de uso del software [21] . Por esto han surgidos trabajos como [17] en los que se persigue obtener algún tipo de metodología o proceso que acorte el número de casos de prueba, y provea de alguna manera una certeza sobre la calidad de la aplicación.

2.3.2 Pruebas de Caja Negra

Conocido también como pruebas dirigidas por datos o basadas en las entradas/salidas generadas por el software. En este caso el componente de software se trata como una caja negra, de la que no se conoce mayor detalle sobre su proceso de construcción, ni tampoco se tiene acceso al código fuente. Los escenarios de prueba se basan en encontrar posibles problemas al usar como referencia las especificaciones, o requerimientos que deben ser

resueltos por la aplicación. [21] Entre las ventajas que se tiene con este tipo de pruebas, están [22]:

- No se requiere experiencia técnica, ni tampoco se requiere el código fuente, lo que también añade una separación entre las perspectivas de usuario y desarrollador.
- Un método eficiente para probar cantidades extensas de código.

Pero también se presentan varias desventajas dado el desconocimiento de cómo se construyó la aplicación, por ejemplo: que los escenarios de pruebas sólo pueden probar el código de forma parcial; algunas pruebas pueden ser incorrectas o mal enfocadas por el desconocimiento interno del software.

2.3.3 Pruebas de Caja Blanca

Para realizar este tipo de prueba se requiere conocer el funcionamiento interno del componente de software, y usando ese conocimiento se proponen y diseñan los diferentes escenarios a ejecutar. Se requiere conocimientos de programación, así como del funcionamiento interno de la aplicación. Algunas de las ventajas que se obtienen de este tipo de pruebas son:

- Eficiente manera de encontrar problemas y errores,
- Conocer el funcionamiento interno, mejora el alcance y las perspectivas de las pruebas a ejecutar.
- Permiten encontrar errores ocultos.
- Puede usarse como herramienta para optimizar el código.

Algunas de las desventajas son:

- Pueden omitir ciertos requerimientos que no han sido cubiertos,
- Requiere un alto conocimiento interno de la aplicación,
- Requiere tener acceso al código fuente.

2.3.4 Tipos de Pruebas en Aplicaciones Móviles

De acuerdo a [23] los tipos de pruebas más importantes en el desarrollo de aplicaciones móviles son:

- Pruebas funcionales, probar el software en función de las especificaciones.
- Pruebas de Rendimiento, tratan de probar la estabilidad y rendimiento de la aplicación.
- Pruebas de usabilidad, un usuario final prueba las funcionalidades de la aplicación.
- Pruebas de compatibilidad, este tipo de pruebas son requeridas debido a la gran cantidad de fabricantes, así como versiones o variantes del sistema operativo Android.
- Pruebas de seguridad, el objetivo es encontrar "puertas abiertas" a través de las cuales quede expuesta información personal.
- Pruebas de interoperabilidad, son pruebas clasificadas como no-funcionales, en las que se verifica la operatividad del software en función de la interacción que este posee con otros sistemas.

2.3.5 Niveles de Pruebas

Otro punto de vista de clasificación de las pruebas se puede encontrar en [24]. La propuesta se base en el nivel o alcance que cubren los escenarios a ejecutar, de esta forma tenemos:

- Pruebas unitarias, son las pruebas de código fuente, generalmente escritas por el mismo programador de la aplicación. Por lo generar estas pruebas usan componentes u objetos simulados (mocks) para complementar la funcionalidad esperada.
- Pruebas de Integración, la intención de estas pruebas es verificar el funcionamiento entre componentes o módulos de la aplicación.
- Pruebas de Aceptación (E2E testing), es el conjunto de pruebas preparadas por el equipo especializado en pruebas. Su objetivo es probar de forma integral la aplicación, usando como base las especificaciones funcionales.

Figura 3. Pirámide de pruebas, según sitio Android Development [13]

2.4 Herramientas para ejecutar pruebas

En esta sección del documento se realizará un estudio general de las herramientas existentes, sus principales características y la forma en que se usan, con la finalidad de extraer las posibles mejoras que podrían implementarse a través del nuevo DSL.

El término ASLT, en español se traduciría como librería de soporte para pruebas en Android, corresponde a un grupo de librerías base, sobre las que permiten la construcción de pruebas en aplicaciones Android. La ejecución, de los escenarios de prueba pueden realizarse sea localmente (usando componentes o funcionalidades simuladas, si fuese el caso) haciendo uso de un emulador de dispositivos, o en un dispositivo físico. Inclusive existen implementación que permiten realizar pruebas en la nube en la que se pueden combinar la relación versión de sistema operativo frente al hardware, esto último fuera del objeto o alcance del presente trabajo.

Con el objetivo de analizar las funcionalidades, ventajas y desventajas en uso de las herramientas existentes se provee una aplicación ejemplo, la cual esta descrita en detalle en el capítulo 3. En este sentido bastará con indicar que es una pequeña aplicación móvil que permite: a) calcular la cuota/pago de mensual de un préstamo, o b) el monto total que se puede solicitar en préstamo. Para este fin la aplicación permite ingresar: el número de períodos expresado en meses y la tasa de interés anualizada.

2.4.1 JUnit

Como se ha anotado anteriormente en este documento, las pruebas unitarias consisten en probar partes específicas de la aplicación de software. En la actualidad casi todo lenguaje de programación tiene su implementación X-Unit. La idea de JUnit viene de SUnit, un marco de referencia para realizar pruebas unitarias en Smalltalk [25]. En Android existe el componente Robolectric, una implementación de X-Unit para configurar y ejecutar pruebas unitarias. Aunque a partir de la implementación de Jetpack, se embebe dentro de la plataforma las anotaciones y componentes JUnit4 para ejecutar las pruebas, y se advierte que a futuro Robolectric dejará de ser soportado. [26]

Al ejecutar pruebas unitarias, lo más importante es validar directamente la mayor cantidad de bifurcaciones que se ha codificado en la correspondiente unidad de prueba, por esta razón es importante la separación de preocupaciones (conocido por su término en inglés concerns); su finalidad es de poder aislar la prueba hacia las funcionalidades específicamente implementadas en la unidad particular de código bajo observación. En este sentido aparece el requerimiento de implementar componentes “simulados” (mocks) los cuales provean la funcionalidad esperada o requerida como parte de la interacción con otras unidades de software o dependencias.

2.4.1.1 Como Trabaja JUnit

Para ejecutar un caso de prueba, se construye una clase en la cual se contienen, uno o más escenarios que se deseen ejecutar. Dentro de esta clase se pueden preconfigurar o instanciar elementos dependientes (stubs y mock) que posteriormente son utilizados por los métodos de prueba. Para realizar las correspondientes verificaciones, esta librería pone a disposición métodos de "afirmación" conocidos como Assert (término en inglés), encargados de evaluar condiciones que determinan si el caso fue correcto o fallido. Los siguientes son los métodos básicos son ofertados por la librería:

Método	Descripción
AssertTrue	Verifica que la condición booleana evaluada sea verdadera.
AssertFalse	Verifica que la condición booleana evaluada sea falsa.
AssertEquals	Verifica que las expresiones evaluadas sean similares (mismo valor), generalmente se usa para comparar una expresión contra un valor constante. Si la comparativa se realiza entre 2 objetos, se evalúa que su contenido sea similar.
AssertNull	Verifica que el resultado de expresión evaluada sea nulo.
AssertNotNull	Verifica que el resultado de expresión evaluada sea No nulo.
AssertSame	Al contrario que AssertEquals, la condición se cumple únicamente cuando las 2 expresiones provistas referencian al mismo objeto.
AssertNotSame	Al contrario de AssertSame, la condición se cumple cuando NO se referencia al mismo objeto.
Fail	Se puede usar este método para indicar, de manera "manual", una condición de NO cumplimiento.

Tabla 1 Métodos principales provistos por JUnit

Además de los elementos básicos, también existen 2 clases que extienden las funcionalidades de "afirmación":

- MoreAsserts, que proveen métodos para evaluar afirmaciones usando expresiones o elementos más complejos como: expresiones regulares, contenidos de lista de datos, etc.
- ViewAsserts, provee mecanismos de evaluación de características y elementos de View (Vistas) en la aplicación.

A continuación, un ejemplo para mostrar la estructura de una clase de prueba en Robolectric [26]

2.4.1.2 Ejemplo

Usamos la lógica de cálculo, para mostrar un ejemplo de JUnit. Dado el monto del préstamo, la tasa de interés y el número de períodos, se calcula el monto de la cuota o pago mensual.

```
@RunWith(AndroidJUnit4.class)
public class ExampleInstrumentedTest {

    @Test
    public void testPaymentAmountCalculation(){
        BigDecimal loanAmount = new BigDecimal("100000");
        BigDecimal interestRate = new BigDecimal("11");
        BigDecimal noOfPayments = new BigDecimal("360");
        LoanCalculatorData info = new LoanCalculatorData(loanAmount, BigDecimal.ZERO,
                                    interestRate, noOfPayments, true);
        LoanCalculator calculator = new CalculatorFactory().build(info, Local);
        BigDecimal result = calculator.getPaymentAmount();
        String message = "El Pago deberia ser 952.32 - {" + result + "}";
        assertTrue(message,result.compareTo(new BigDecimal("952.32")) == 0);
    }
}
```

2.4.1.3 Elementos Relevantes

Podemos clasificar a JUnit como un componente para realizar pruebas del tipo caja blanca, se requiere el código fuente, y se usa principalmente para realizar pruebas unitarias. A la vez puede ser usada para probar cualquier aspecto de la aplicación: seguridad, usabilidad, rendimiento, etc. Entre los aspectos más relevantes de las pruebas basadas en JUnit están: poder depurar el código fuente, encontrando errores de codificación, se puede usar, de ser el caso, para realizar mejoras estructurales (refactorización del código) o en cuestiones relacionadas al rendimiento, velocidad de procesamiento, y utilización de recursos

(memoria, procesamiento, almacenamiento, etc.). Se requiere conocimientos de programación para poder escribir los escenarios de prueba.

2.4.2 Robotium

Es un proyecto de código abierto, especializado en la generación de pruebas de aplicaciones Android, del tipo caja negra tanto en aplicaciones nativas y híbridas. Se usa para crear escenarios prueba sobre la interfaz de usuario de aplicaciones, esto es sobre los componentes visibles en las correspondientes actividades (Activities por su terminología en Android). Robotium también hace uso de JUnit, de esta manera los escenarios de prueba escritos heredan las funcionalidades provistas por JUnit.

2.4.2.1 Como funciona

Los escenarios de prueba están escritos en java, y se enfocan en evaluar los componentes gráficos que se encuentra visibles en la Activity. Para crear un escenario de ejecución se requiere una clase Java que implemente la interfaz ActivityInstrumentationTestCase2 a través de la cual se inyecta el nombre de la clase de la Activity a probar. Posteriormente usando la instancia del objeto Solo, y usando la API provistas, se acceden a los elementos de la pantalla para ejecutar las acciones requeridas. Los principales métodos provistos por la clase Solo son:

Método	Descripción
getView	Buscar un especificó objecto de tipo View en la Activity actual.
waitForText	Espera un tiempo hasta que aparezca un texto específico en la pantalla.
clickOnButton	Permite ejecutar la acción clic sobre un elemento de tipo Button.
sendKey	Envía la cadena de texto, emulando el ingreso por teclado, al elemento especificado.

clickOnText	Busca el texto especificado dentro de la Activity visible, y ejecuta la acción de clic sobre el elemento asociado.
searchText	Busca si el texto indicado es visible o no en la interfaz de usuario.
clickInList	Simular la selección del elemento dentro de una lista.
takeScreenshot	Permite capturar la imagen de la pantalla y grabarla como archivo.

Tabla 2 Métodos principales provistos por Robotium

Los métodos expuestos además de los elementos heredaros desde JUnit, permiten crear los escenarios de validación requeridos.

2.4.2.2 Ejemplo

```
public class TestingClass extends ActivityInstrumentationTestCase2<MainActivity> {

    private Solo solo;
...
    public void testPaymentAmountCalculation() {
        solo.enterText((android.widget.EditText) solo.getView(R.id.txtLoanAmount),
                                                 "100000");
        solo.enterText((android.widget.EditText) solo.getView(R.id.txtInterestRate),
                                                 "11");
        solo.enterText((android.widget.EditText) solo.getView(R.id.txtNoOfPayments),
                                                 "360");
        solo.clickOnText("Calcular");
        EditText result = (EditText) solo.getView(R.id.txtPaymentAmount);
        assertEquals(result.getText().toString(), "952.32");
    }
}
```

2.4.2.3 Elementos Relevantes

Robotium es una herramienta que permite realizar pruebas de caja negra, no se requiere poseer el código fuente. Para ejecutar Robotium se requiere necesariamente o un emulador del dispositivo, o el dispositivo físico. No es posible ejecutar Robotium sobre el código fuente. Documentación y guías de usuario ya no se pueden encontrar con facilidad. Para escribir los escenarios de prueba se requiere conocimiento técnico.

2.4.3 Espresso

Espresso es una herramienta que permite realizar pruebas instrumentadas, esto es de tipo caja negra sobre aplicaciones Android. De acuerdo a [27] es la herramienta más utilizada por los desarrolladores para ejecutar pruebas debido a su versatilidad, soporte de Google, y también por ser una herramienta de tipo código abierto. Espresso hace uso de la funcionalidades JUnit para implementar los casos de prueba. Los escenearios de pruebas se pueden escribir ya sea en Java o Kotlin

2.4.3.1 Como funciona

La base de Espresso también es JUnit, lo que permite ejecutar los distintos escenarios de prueba. Los casos de prueba pueden o no ser parte del código fuente de la aplicación. Los componentes principales en Espresso son:

Componente	Descripción
ViewMatchers	Estas classes permiten, con varias funciones, buscar elementos de la interfaz de usuario que coincidan con los datos indicados, retornando como resultado objetos del tipo ViewInteraction, y a través de estos se puede interactuar programáticamente con dichos elementos de interfaz usando ViewActions.
ViewActions	Son clases que permiten interactuar con los elementos de interfaz seleccionados usando principalmente el método perform.
ViewAssert	Similar a las clases de ViewActions, estas permiten en cambio verificar el estado, contenidos o atributos de un elemento de interfaz.

Tabla 3 Componentes principales en Espresso

La combinación de la anotación Rule más el uso de la clase ActivityTestRule permite indicar, en el correspondiente clase de prueba, la Activity de la aplicación sobre la que se

ejecutan los escenarios. Para escribir una prueba se debe hacer uso de los elementos de tipo Matcher, Actions y Asserts. Se describe a continuación los componentes más relevantes de cada uno de los tipos indicados.

Matchers

Matcher	Descripción
withId	Permite localizar un elemento de interfaz dado algún tipo de valor de identificación, por ejemplo su Android Id.
withText	permite localizar un elemento de interfaz usando un texto con el que se despliega en la pantalla.
isRoot	utilidad para localizar el elemento de interfaz padre de la Activity.

Tabla 4 Componentes tipo Matcher en Espresso

Otros utilidades requeridos para localizar elementos de interfaz, son hasFocus, isEnabled, isFocusable, isClickable, isSelected, isChecked, withChild. Existe además de la interfaz Custom Matcher, a través de la cual se puede implementar clases de búsqueda de elementos de interfaz de usuario personalizados.

Asserts

El método check, es el método principal que recibe como argumento con un elemento de tipo Assert, para realizar la verificación correspondientes. Los diferentes asserts que pueden usar se listan a continuación:

Método	Descripción
doestNotExist	Se utiliza para verificar si se pudo localizar una elemento de interfaz de usuario.

matches	Permite verificar si el elemento localizado contiene o coincide un texto indicado.

Tabla 5 Componentes tipo Assert en Espresso

Actions

Como se explicaba anterior, otro de los componentes de Espresso, es la utilización de los Actions que permiten interactuar con los elementos de interfaz, entre las acciones más utilizadas se encuentra:

Método	Descripción
typeText	Permite digitar texto sobre el elemento de interfaz seleccionado, si el elemento ya contiene un texto, el nuevo texto se puede insertar de forma arbitraria si un orden predeterminado.
clearText	Limpia el contenido del elemento de interfaz.
pressKey	Se permite usarla para similar la presión una tecla, por ejemplo ENTER.
Click	Invoca a la acción clik del elemento de interfaz.
scrollTo	Permite deslizar la pantalla hasta llegar al elemento de interfaz.

Tabla 6 Componentes tipo Action en Espresso

2.4.3.2 Ejemplo

```
@RunWith(AndroidJUnit4.class)
@LargeTest
public class ActivityCalculationTest {
    @Rule
    public ActivityTestRule<MainActivity> mActivityTestRule = new
                            ActivityTestRule<>(MainActivity.class);
    @Test
    public void calculate_PaymentAmount() {
        onView(withId(R.id.txtLoanAmount)).perform(typeText("100000"),
                                    closeSoftKeyboard());
        onView(withId(R.id.txtInterestRate)).perform(typeText("11"),
                                    closeSoftKeyboard());
        onView(withId(R.id.txtNoOfPayments)).perform(typeText("360"),
                                    closeSoftKeyboard());
        onView(withId(R.id.btnCalculate)).perform(click());
        onView(withId(R.id.txtPaymentAmount)).check(matches(withText("952.32")));
    }
```

```
}
```

2.4.3.3 Elementos Relevantes

Espresso permite realizar pruebas de tipo caja negra. Una de las características de este marco de referencia es que es extensible, y se puede añadir nueva funcionalidad que se requiera en la ejecución de pruebas. El soporte a estas librerías viene del propio del Google, e inclusive ya forma parte de las nuevas librerías AndroidX. También tiene soporte de compatibilidad hacia atrás, es decir se pueden realizar pruebas sobre aplicaciones creadas con versiones antiguas de Android. Existe muy buena documentación, y relativamente fácil de aprender e implementar. No se requiere el código fuente de la aplicación para probarla. [28] Espresso sólo puede operar dentro de aplicación que está siendo probada. Esto no es posible: Verificar notificaciones, o acceder a la configuración del sistema, o navegar hacia otras aplicaciones para realizar pruebas.

2.4.4 Cucumber

Cucumber es una herramienta especializada para ejecutar pruebas de usuarios de forma automática, y puede ser utilizado en el desarrollo de aplicaciones con enfoque BDD. A través de Cucumber la comunicación entre los expertos de negocio y los programadores, mejora, dado que permite describir los casos de prueba usando idioma natural, y que los programadores puedan comprender de mejor manera el requerimiento que debe ser cubierto por el software que se está desarrollando. [29] Cucumber provee librerías que puede ser utilizada en programas Java, JavaScript, Ruby y Kotlin.

2.4.4.1 Como funciona

Cucumber propone el uso de un DSL llamado Gherkin. Usando dicho DSL el experto de negocio puede expresar, en lenguaje natural, los diferentes escenarios o casos de prueba que deben ser cubiertos por la aplicación. Para usar esta herramienta requiere dos elementos:

- el primero es el archivo de texto que contiene la característica y los elementos de cada escenario a probar, y

- de otro lado un programa en el que se codifica las invocaciones a la lógica que implementan cada uno de los pasos descritos en los escenarios de prueba.

En este contexto, Cucumber toma los escenarios descritos en el archivo de características (feature), y localiza, en el programa de invocaciones, la sección de código que debe ejecutarse para cada uno de los pasos descritos en dicho escenario.

En la siguiente figura se muestra cómo trabaja Cucumber en tiempo de ejecución:

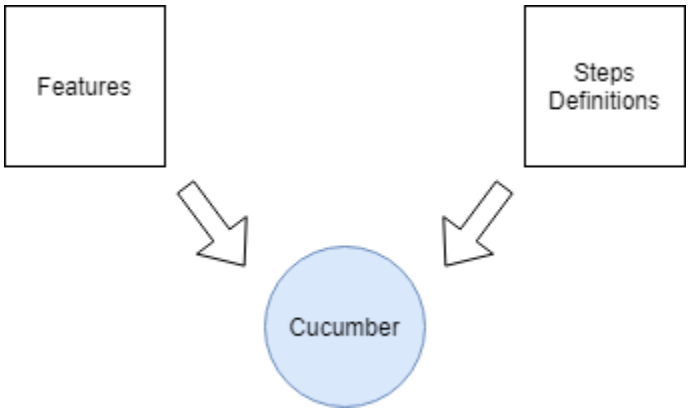


Figura 4. Elementos de Cucumber

El archivo de características es un archivo de texto plano que describe los escenarios utilizando Gherkin. A continuación, se listan los componentes más relevantes del lenguaje Gherkin:

- Feature, se usa para describir la característica a probar, permite agrupar la seria de escenarios a probar con relación a dicha característica.
- Scenario/Example, permite describir los diferentes pasos que deben ejecutarse en el correspondiente caso de prueba. Un escenario está compuesto de tres elementos: Given, When y Then que referencian como Steps.
- Steps (Given When Then And But),
    - Given, se usa para describir el estado, o las precondiciones que deben cumplirse. Al ejecutarse las instrucciones de tipo Given, se asemeja a

realizar configuraciones de parámetros o valores utilizados posteriormente para la ejecución del escenario.

- When, describe una acción o la ejecución de evento sobre el software realizado ya sea por un usuario o por otro sistema.
- Then, permite describir los pasos de verificación, es decir los resultados a ser validados en el escenario.
- And, But, se usa para concatenar pasos que corresponden al mismo nivel.

- Background, en ocasiones más de un escenario dependen de precondiciones similares, se utiliza Background para definir las instrucciones Given que serán usadas por todos los escenarios descritos.

Además, para los escenarios se permite agregar partes variables, conocidas como “Doc String” o “Data Table”.

feature	Característica
background	Antecedentes
scenario	Escenario
scenarioOutline	Esquema del escenario
examples	Ejemplos
given	* Dado Dada Dados Dadas
when	* Cuando
then	* Entonces
and	* Y E

but	* Pero

Tabla 7. Elementos Cucumber en español

2.4.4.2 Ejemplo

```
Feature: Calculate Loan Amount

  @loanAmount-calculation
  Scenario Outline: Calculate Loan Amount
    Given I start the application
    When I click Importe Préstamo
    And  I enter loan amount <loanAmount>
    And  I enter interest rate <interestRate>
    And  I enter number of payments <noOfPayment>
    And  I click CALCULAR
    Then I expect to see the value <paymentAmount> at Cuota Mensual

    Examples:
      | loanAmount | interestRate | noOfPayment | paymentAmount
      |  100000 | 12              | 240         | 11010.86
```

Clase de instrumentación

```
public class LoanAmountCalculation {

...

    @Given("^I start the application$")
    public void I_start_application() {
        assertNotNull(activity);
    }

    @When("^I click Importe Préstamo$")
    public void I_click_ImportePrestamo() {
        onView(withId(R.id.chkpaymentAmount)).perform(click());
    }

    @And("^I enter loan amount (\\\\S+)$")
    public void I_enter_loanAmount(String loanAmount) {
        onView(withId(R.id.txtLoanAmount)).perform(
                    typeText(loanAmount),closeSoftKeyboard());
    }

    @And("^I enter interest rate (\\\\S+)$")
    public void I_enter_interestRate(final String interestRate) {
        onView(withId(R.id.txtInterestRate)).perform(
                    typeText(interestRate),closeSoftKeyboard());
    }

    @And("^I enter number of payments (\\\\S+)$")
    public void I_enter_noOfPayments(final String noOfPayments) {
        onView(withId(R.id.txtNoOfPayments)).perform(
                    typeText(noOfPayments),closeSoftKeyboard());
    }
```

```
    @And("^And  I click CALCULAR$")
    public void I_click_CALCULAR() {
        onView(withId(R.id.btnCalculate)).perform(click());
    }

    @Then("^I expect to see the value (\\\\S+) at Cuota Mensual$")
    public void I_should_see_paymentAmount(final String payment) {
        onView(withId(R.id.txtPaymentAmount)).check(matches(withText(payment)));
    }

}
```

2.4.4.3 Elementos Relevantes

Aunque el contexto inicial de Cucumber es el desarrollo de aplicaciones dirigido a comportamiento (BDD), en el ámbito de ejecución de pruebas es una herramienta que permite realizar tanto pruebas de tipo caja blanca como de tipo caja negra.

Se requiere experiencia técnica, ya que la lógica requerida para ejecutar escenarios debe ser escrita en código fuente, este código debe hacer uso de algún tipo de interfaz (instanciación de clases, invocación a metodos, etc.) para interactuar con la aplicación bajo prueba. Está claro que la funcionalidad más relevante de Cucumber, es la transformación y convergencia entre texto plano, y las instrucciones ejecutables descritas en programas, utilizando, opcionalmente parámetros o datos de entrada (tablas de datos). Otro aporte de Cucumber es la definición de Gherkin, herramienta útil para extrapolar los requerimientos descritos en términos de negocio, al entendimiento requerido, por parte del programador, para implementarlos en código fuente.

2.4.5 Calabash

Es un proyecto de código abierto que permite ejecutar escenarios de pruebas tanto en aplicaciones Android, e iOS. Uno de sus objetivos es que los escenarios puedan ser escritos por una persona sin conocimientos técnicos. Calabash está orientado a permitir el desarrollo de aplicaciones móviles usando BDD. [30] Calabash necesita Cucumber como base, y el lenguaje de programación Ruby como motor de ejecución. Para el caso de Android existe el proyecto calabash-android.

2.4.5.1 Como funciona

Para usar Calabash se requiere un archivo texto con extensión feature, en este archivo, usando Gherkin se colocan los diferentes escenarios de prueba a ejecutar, para la correspondiente característica. Además, se requiere un programa Ruby en el cual se implementen la lógica que relaciona los pasos descritos en cada escenario con la lógica que debe ejecutarse o validarse en la aplicación. No existe una documentación formal de las acciones que se pueden ejecutar en una aplicación desde Calabash, pero a continuación, se resumen las acciones, más relevantes, a partir de la referencia existente en [31]:

Acción	Descripción
Touch	Seleccionar o tocar sobre un elemento de la pantalla.
Sleep	Detener la ejecución por un cierto número de segundos.
Query	Consultar si un elemento está o no presente en la pantalla.
tap_keyboard_action_key	Mostrar u ocular el teclado virtual
wait_for_keyboard	Esperar a que aparezca el teclado.
keyboard_enter_text	Ingresar un texto.
clear_text	Limpiar el texto de una caja de ingreso
wait_for	Junto a element_exists para esperar hasta que el elemento sea visible en la pantalla.

Tabla 8 Elementos relevantes en Calabash

Con los métodos indicados, se crean los scripts de ejecución para relacionar los escenarios con las acciones a ejecutar en la aplicación, pero además Calabash provee, al igual que en otros marcos de referencia, formas de realizar verificaciones o "Asserts", para esto provee las instrucciones:

- query,
- element_exists
- check_element_exists
- check_element_does_not_exist

2.4.5.2 Ejemplo

Si tenemos el siguiente archivo feature:

```
Scenario Outline: Calculate Loan Amount
    Given I start the application
    When I click Importe Préstamo
    And  I enter loanAmount into the loan amount field
    And  I enter interestRate into the interest rate field
    And  I enter noOfPayment into the number of payments field
    And  I click CALCULAR button
    Then I expect to see the value paymentAmount at Cuota Mensual
```

La implementación en Calabash sería:

```
require 'calabash-android/calabash_steps'
When /^I enter "([^\"]*)" into the "([^\"]*)" field$/ do |text_to_type,
field_name|
  touch("textField marked: '#{field_name}'")
  wait_for_keyboard
  keyboard_enter_text text_to_type
  sleep(STEP_PAUSE)
end

Then /^I click "([^\"]*)"$/ do |field_name|
  touch("checkBox marked: '#{field_name}'")
end

Then /^I enter "([^\"]*)" into the loan amount field$/ do
|text_to_type|
  touch("textField marked: 'Importe del Préstamo'")
  wait_for_keyboard
  keyboard_enter_text text_to_type
  sleep(STEP_PAUSE)
end

Then /^I enter "([^\"]*)" into the interest rate field$/ do
|text_to_type|
  touch("textField marked: 'Tasa de Interés'")
  wait_for_keyboard
  keyboard_enter_text text_to_type
  sleep(STEP_PAUSE)
end

Then /^I enter "([^\"]*)" into the number of paymens field$/ do
|text_to_type|
  touch("textField marked: 'Número de Pagos'")
  wait_for_keyboard
  keyboard_enter_text text_to_type
  sleep(STEP_PAUSE)
end

Then /^I (?:press|touch) the "([^\"]*)" button$/ do |name|
```

```
  touch("Button touched: '#{name}'")
  sleep(DELAY_IN_SECONDS)
end

Then /^I expect to see the value "([^\"]*)" at Cuota Mensual$/ do
|expected_mark|
  wait_for(TIMEOUT) { view_with_mark_exists( expected_mark ) }
end
```

2.4.5.3 Aspectos Relevantes

En principio hereda las características y puntos relevantes de Cucumber. Pero además se debe anotar que, si bien se indica que Calabash está orientada a un usuario que no requiere conocimientos técnicos, de alguna forma para crear el programa de Ruby, si lo es; caso contrario es bastante difícil que una persona sin experiencia en Ruby pueda implementar los programas requeridos. Al momento de usar Calabash en una aplicación existente, se requiere de algún tipo de mecanismo que permita la introspección de los elementos de la aplicación, por ejemplo: Appium Inspector Studio, con el cual obtener los correspondientes identificadores de los componentes (Views) en cada Activity a probar.

2.5 Análisis y evaluación de las alternativas

Como se puede observar en el análisis, cada una de las herramientas descritas aportan diversas funcionalidades requeridas para realizar la automatización de pruebas de aplicaciones móviles. El primer resultado de la evaluación de las herramientas existentes, es que se requiere conocimiento técnico para poder hacer uso de los elementos de pruebas propuestos. Cucumber y Calabash serían las herramientas más óptimas el momento de evaluar la elaboración de pruebas sobre aplicaciones móviles, por parte de expertos de negocio, sin conocimientos de desarrollo de aplicaciones, pero, aun así, la integración de los escenarios con las funcionalidades a probar, requiere de la escritura de programas, para lo que se requiere cierta pericia técnica.

Lejos está también usar JUnit, debido a su enfoque totalmente técnico, en el que se requiere el análisis y conocimiento del código fuente. De otro lado Robotium tampoco es una opción, no sólo por los requerimientos de conocimientos necesario, sino además por la poca documentación y ejemplos existentes, al parecer Robotium es un producto

descontinuado y de muy poco uso. En esta misma línea se debe indicar que Calabash, tampoco posee una documentación robusta, el realizar el ejemplo de este documento fue relativamente costoso, en relación al objetivo de usarlo; al igual que Robotium es un producto poco usado.

A partir del análisis y evaluación realizada de las herramientas disponibles, para implementar la solución del DSL propuesto en este documento, se hará uso de componentes y características proporcionadas por Cucumber y Espresso. En el primer caso, se opta por armar el DSL siguiendo lineamientos o un esquema similar al propuesto por Gherkin, y en el segundo caso, se hará uso de los componentes de automatización de pruebas provistos por Android (UIAutomator), esto como resultado de la decisión de usar librerías Appium, descrita a continuación.

2.6 Appium

[32]Appium es un proyecto de código abierto, utilizado para la automatización de pruebas sobre aplicaciones móviles. Consiste en una aplicación servidor escrita sobre node.js, similar enfoque a Selenium WebDriver. Las principales características de esta librería son:

- Puede usarse para probar aplicaciones Android y iOS.
- Pueden probarse aplicaciones nativas, híbridas y de tipo Web.
- La ejecución de las pruebas puede realizar en plataforma Windows, Linux o MAC iOS.

2.6.1 Como funciona Appium

A continuación, se describe de forma breve, la manera en la que Appium trabaja, iniciando por su arquitectura sobre Android.

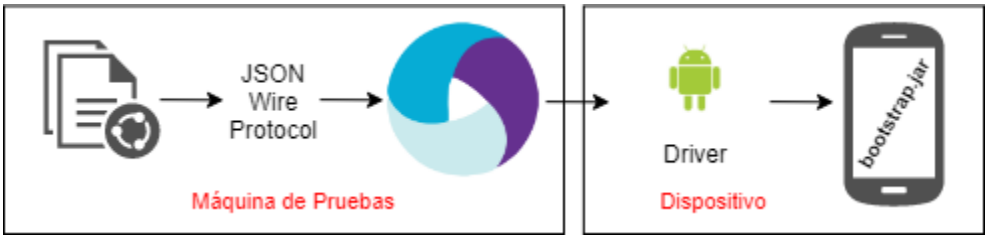


Figura 5. Arquitectura de Appium para Android [32]

Del lado derecho de la figura se encuentra el script o programa que describe la prueba a ejecutar, más el aplicativo que provee los servicios Appium (servidor). Los programas se escriben usando las librerías de tipo cliente, que internamente, usan la implementación Mobile JSON Wire Protocol [33] del estándar W3C Webdriver Spec (por sus siglas en inglés). Posteriormente el servidor Appium se conecta hacia un cliente TCP en el dispositivo, a través de "WebDriver" (UI Automator/Espresso) el cuál interpreta finalmente las instrucciones.

Los casos de prueba de Appium se pueden escribir en Ruby, C#, Python, Java, JavaScript, Objective-C, PHP. La ejecución puede realizar de forma local, haciendo uso de emuladores de dispositivos, o se pueden realizar de forma remota, lo que conlleva la opción de poder utilizar infraestructura en la nube. A través del Driver se puede tener acceso a los diferentes componentes de la pantalla, y desde este API se puede realizar evaluación de condiciones (contenido, valores de atributos), o también se puede ejecutar acciones sobre estos elementos (clic, ingreso de datos, etc.).

2.6.2 Configuración Inicial

Los siguientes elementos de configuración que conforman las "desired capabilities" son obligatorias para iniciar la sesión en el caso de aplicaciones nativas o híbridas:

```
{
  "deviceName": "emulator-5554",
  "platformName": "android",
  "appPackage": "edu.uned.missi.calcprestamo",
  "appActivity": "edu.uned.missi.calcprestamo.MainActivity",
  "noReset": true
}
```

Donde:

- deviceName, es el identificador del dispositivo en el que se ejecutará la prueba.
- platformName, corresponde al sistema operativo del dispositivo.
- appPackage, nombre de la aplicación Android.
- appActivity, identificador de la Activity inicial desde la que se inicia
- noReset, el proceso de reinicio indica que antes de iniciar la sesión de prueba se debe asegurar que la aplicación móvil también haya sido reiniciada.

2.6.3 Acceso a los elementos de la interfaz

Para tener acceso, a través del API, a los elementos de la interfaz se debe hacer uso de la clase MobileElement, esta clase provee métodos como:

Método	Descripción
findElement	Localizar un elemento de interfaz haciendo uso de búsqueda a través de la evaluación de sus atributos. La principal forma de localizar es utilizando su identificador único
click	Seleccionar o realiza un clic sobre el elemento de interfaz
clear	Limpiar el contenido
isEnabled, isDisplayed	Comprobar si el elemento esta activo, y el segundo elemento si es visible en la interfaz.
sendKeys	Ingresar o enviar valores del teclado al componente

Tabla 9 Principales métodos de interacción en Appium

2.6.4 Limitantes

Appium tiene algunas limitaciones que deben tomarse en consideraciones antes de decidirse usarse:

- La versión mínima soportada es Android 4.2
- No provee funcionalidad apropiada para trabajar con imágenes.
- Algunos tipos de gestos móviles no son soportados

2.6.5 Ejemplo

```
MobileElement el2 = (MobileElement)
driver.findElementById("edu.uned.missi.calcprestamo:id/txtLoanAmount");
el2.click();
el2.sendKeys("100000");
MobileElement el3 = (MobileElement)
driver.findElementById("edu.uned.missi.calcprestamo:id/txtInterestRate"
);
el3.click();
el3.sendKeys("12");
```

```
MobileElement el4 = (MobileElement)
driver.findElementById("edu.uned.missi.calcprestamo:id/txtNoOfPayments"
);
el4.click();
el4.sendKeys("240");
MobileElement el5 = (MobileElement)
driver.findElementById("edu.uned.missi.calcprestamo:id/btnCalculate");
el5.click();
MobileElement el6 = (MobileElement)
driver.findElementById("edu.uned.missi.calcprestamo:id/txtPaymentAmount
");
BigDecimal result = new BigDecimal(el6.getText());
String message = "El Pago deberia ser 1101.09 - {" + result + "}";
assertTrue(message,result.compareTo(new BigDecimal("1101.09")) == 0);
```

2.6.6 Criterios de Selección

El seleccionar las librerías Appium como parte del trabajo, nos provee de las siguientes características:

- Solapar la complejidad de interactuar con las librerías base para la ejecución de pruebas como UI Automator/Espresso.
- Se puede automatizar pruebas sobre cualquier tipo de aplicación móvil, sea esta nativa, hibrida o web.
- Las pruebas se pueden realizar sobre emuladores o sobre dispositivos reales.
- Los scripts de prueba pueden escribir en varios lenguajes de programación: Java, Ruby, Python, JavaScript, PHP y Objective C)
- De ser el caso, se podría usar para, extendiendo su funcionalidad, trabajar con aplicaciones hechas de iOS.
- Permite realizar pruebas del tipo Caja Negra. Es decir, se puede probar únicamente la lógica que ha sido expuesta como funcionalidad en el aplicativo, con Appium no es posible probar la lógica interna en detalle.
- También se permite probar funcionalidad del dispositivo, como simular que el dispositivo ha dado la vuelta, o se presionó el botón “inicio/home”

2.7 DSL

La otra arista que se requiere evaluar para proponer una solución, es los conceptos y elementos relacionados con la implementación de DSL. En esta sección se analizarán los principales elementos relacionados con los lenguajes específicos de dominio o DSL por sus siglas en inglés.

2.7.1 Descripción DSL

Un DSL se refiere a un lenguaje de programación (implementable o no) que provee sentencias y proposición especializadas en resolver un problema de un área específica [34]. El DSL ofrece un nivel alto de abstracción, debido a que su objetivo es centrarse en el ámbito del problema, proveyendo una serie de elementos particularmente relacionados con su particular área de dominio. Por esto se puede afirmar que uno de los propósitos de la construcción de un DSL es disminuir las brechas entre la implementación técnica, y el requerimiento planteado; permitiendo, que usuarios expertos en el problema o dominio, puedan generar soluciones computacionales a sus requerimientos, sin necesidad de conocer en detalle su implementación (código fuente, dependencias, librerías, arquitectura, etc.).
Algunos ejemplos clásicos que pueden ser clasificados como DSL, son:

- SQL, manejar datos en motores de base de datos relacionales
- Ant, Rate o Make, crear scripts para la construcción de sistemas de software.
- CSS, hoja de estilos para aplicar formatos a páginas HTML
- HTML, lenguaje marcado para construcción de páginas Web.

Alternativamente a la creación de un DSL, es la implementación de marcos de trabajo o librerías, se aconseja por ejemplo que cuando el problema a resolver es muy reciente, o se posee poco conocimiento o usuarios expertos, evaluar la posibilidad de implementar librerías en lugar de DSL. [35]

2.7.2 Clasificación

La manera más aceptada para clasificar los DSL, se base en la forma en la que se lo implementa. Si el DSL utiliza para su ejecución el mismo lenguaje en el que se crea,

entonces se lo conoce como DSL interno o embebido, al contrario, sí el DSL debe transformarse para ser ejecutado en otro lenguaje entonces se lo conoce como DSL externo. Un ejemplo de DSL interno es Rails: este lenguaje se utiliza para el desarrollo de aplicaciones web, es una extensión de Ruby, se compila y ejecuta sobre el mismo ámbito del programa Ruby. Em cambio en un DSL externo el lenguaje en el que se construye es diferente al de implementación, y por ende se requiere no solo mecanismo externo para escribir y validarlo, sino también procesos que permitan ya sea la transformación (compilación) o la interpretación de las instrucciones.

2.7.3 Aspectos del Diseño

En la implementación de un DSL se pueden usar: textos, símbolos, tablas, elementos gráficos; con la intención de que el lenguaje sea realmente usable por los expertos de dominio, y permitan una adecuada abstracción de los componentes del problema.

Para resolver el problema planteado en este documento, nos enfocaremos en los DSL de tipo externos, debido a que requerimos la interacción con librerías que ejecutarán las instrucciones descritas en nuestro nuevo DSL.

En este sentido, se describe que los DSL externos se pueden implementar usando 2 enfoques: a) el primero es transformación, un mecanismo traslada (compila o convierte) las instrucciones a código que posteriormente se ejecuta sobre el lenguaje final, o b) interpretativo donde existe un motor o interprete que va evaluando y ejecutando cada una de las sentencias del programa DSL.

También es importante anotar las cualidades que deben cumplir un buen diseño de DSL deben ser [34]:

- Minimalismo, debe implementarse únicamente aquellos aspectos identificador por el experto de dominio.
- Destilación, la abstracción de elementos debe ser lo más simple posible, ocultando u omitiendo de la interfaz, todos aquellos detalles no esenciales,
- Extensibilidad, el diseño debe permitir que el lenguaje sea extendido para implementar nuevas funcionalidades, o requerimientos más especializados, sin que los programas existentes sean afectados.

- Composición, de debe lograr que los elementos a construir se puedan reordenar o juntar para formar parte de componente más complejos.

2.7.4 Implementación de un DSL externo

Utilizando el gráfico siguiente, se puede deducir la forma en la que se implementa una solución basada en un DSL Externo:

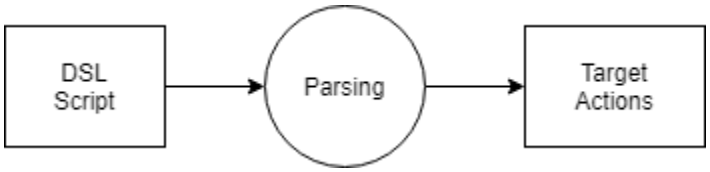


Figura 6. La manera más simple de implementación de un DSL externo [34]

La implementación de un DSL consiste en identificar los elementos que conformarán el lenguaje, en este punto es imprescindible la participación del experto del dominio, con el cual se debe acordar los diferentes elementos. A partir de estos elementos se genera y acuerda la sintaxis y semántica (acciones asociadas) del lenguaje. De este modo tenemos la sintaxis abstracta que contiene la lista, en detalle, de los componentes textuales del lenguaje, incluidos los caracteres de separación, espacios en blanco y posiciones.

Si bien el uso de un DSL [36] se enfoca a ser utilizado por un usuario experto, no es viable que este expresado en lenguaje natural, debido a las ambigüedades que puedan existir; pero el diseño del DSL debe considerar una sintaxis que permita expresar la intencionalidad. De esta manera se logra que la implementación computacional del DSL sea más sencilla de lograr. Ahora, esto genera un dilema en el que se debe propender a un equilibrio, entre la claridad (facilidad de lectura) para el usuario final, y la rigidez requerida en la sintaxis.

3 Solución Propuesta

3.1 Entendiendo el Problema

En esa sección ser realiza la mayor parte del trabajo que consiste en implementar de una manera descripta la solución al problema propuesto.

3.1.1 Aplicación Móvil

Como punto de partida, que permita delimitar el alcance de la solución, se debe indicar que en esta propuesta se considera como aplicación móvil a aquellas aplicaciones que forman parte de un sistema de información, en los que un usuario proporciona información y datos, para que luego de ser procesados, la aplicación generé un resultado. En el contexto del problema, diríamos son aplicaciones están compuestas de elementos de interfaz, (que los vamos a conocer como widgets para utilizar la jerga Android) que permiten al usuario ingresar información al sistema, y también ordenar la ejecución de procesos, de los cuales, usando los mismos widgets, se obtiene una respuesta.

3.1.2 Anatomía del problema

En el problema propuesta podemos identificar 3 elementos: a) el usuario que ejecuta o utiliza la aplicación, b) la aplicación móvil, y c) el escenario o caso de prueba.
El caso de prueba o escenario, es el que describe el uso de estos elementos, ya sea para realizar la ejecución de una acción (ingreso, selección, etc.) o para verificar el estado del mismo (es visible, está activo, contiene un dato, etc.). En este sentido tenemos como resultado la siguiente figura, que ilustra los elementos que conforman el problema planteado y la manera en la que interactúan.

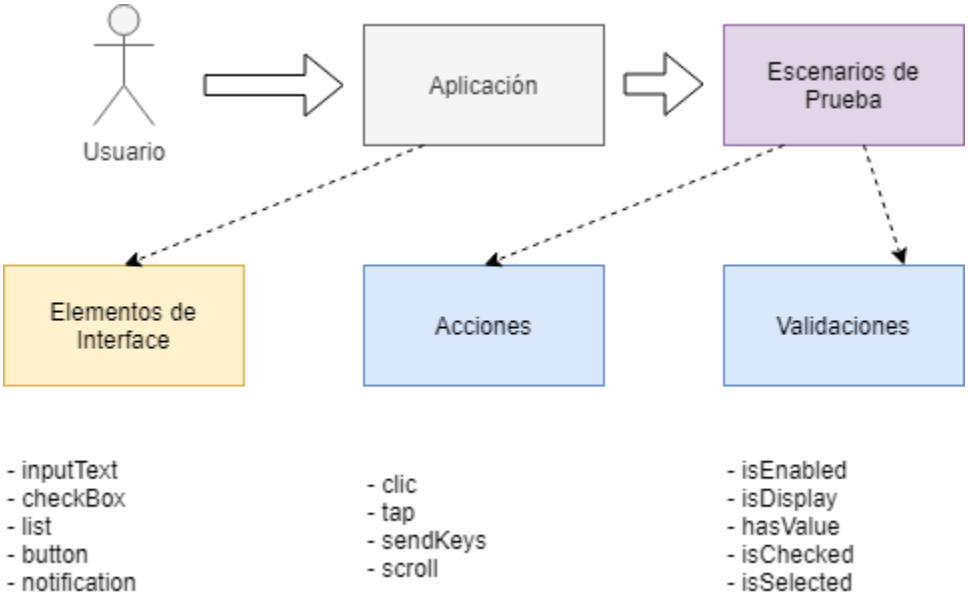


Figura 7. Anatomía del Problema a Resolver

Nota importante. - Es necesario aclarar en este punto, que se usará el idioma inglés para hacer la definición del DSL, primero por la universalidad del mismo (mayoría de documentación, implementaciones, software se trabaja en este idioma), y segundo porque permite simplificar el proceso de desarrollo de la propuesta, ya que es compatible con las librerías y lenguajes de programación existentes.

3.2 Aplicación de Prueba

Con la finalidad de llevar a cabio la implementación del DSL de una forma descriptiva, se propone hacer uso una pequeña aplicación que permite el cálculo de valores, de acuerdo a opciones, del valor de una hipoteca. De acuerdo a la opción seleccionada, se puede calcular el importe total, o el pago mensual.

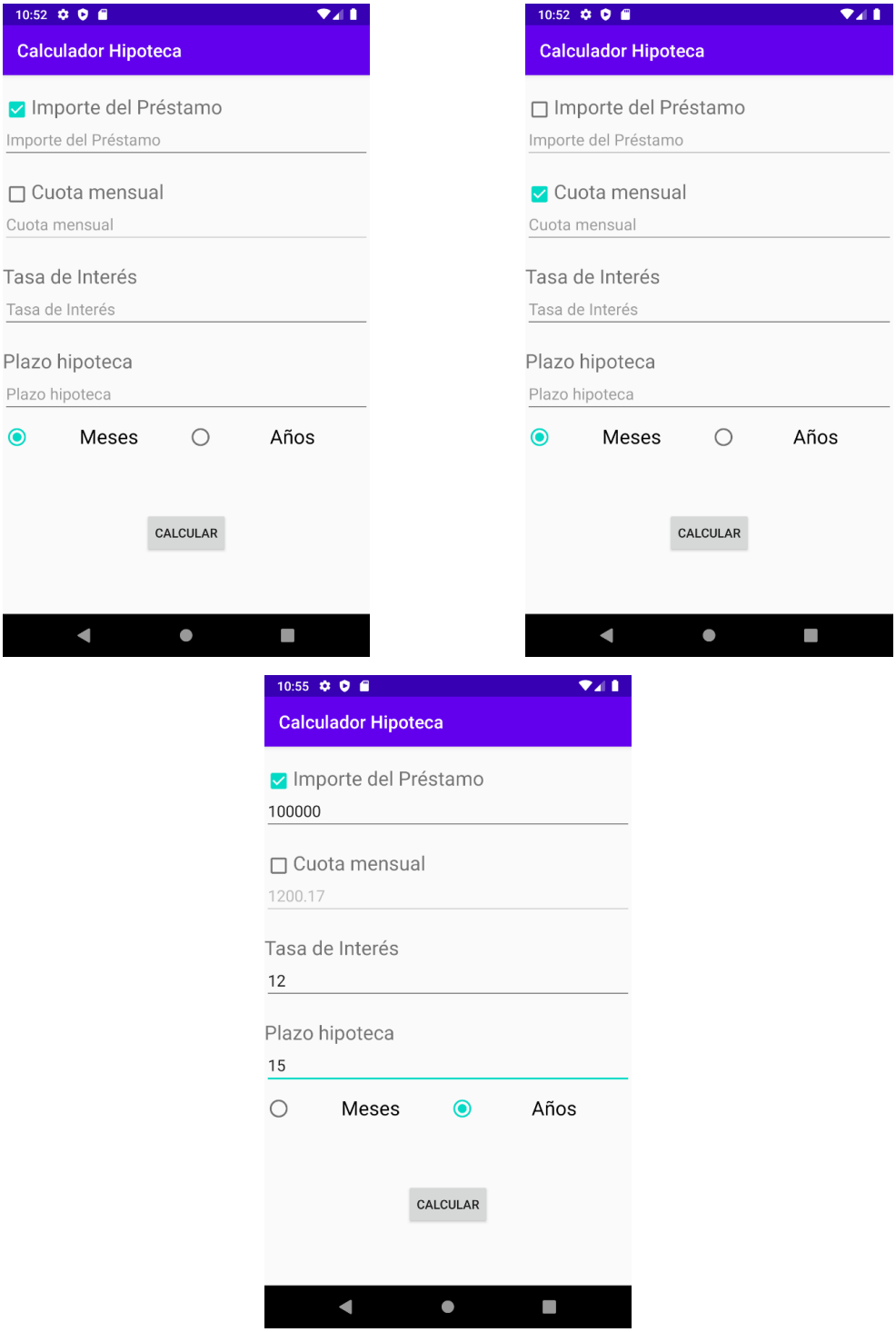


Figura 8. Interfaz de usuario de aplicación de Ejemplo

Opcionalmente el usuario puede seleccionar la convención a usar para tomar el valor de plazo hipoteca, el cual pueda estar expresado en años o meses, según corresponda.

3.3 Esquema general de la solución

En la siguiente figura se esboza el esquema de solución: a partir de un archivo escrito en lenguaje DSL, un proceso de "Parsing" y generación, lo transformará en código fuente JAVA; este a su vez, usando librerías APPIUM, permitirá ejecutar los pasos requeridos para verificar el escenario de prueba.

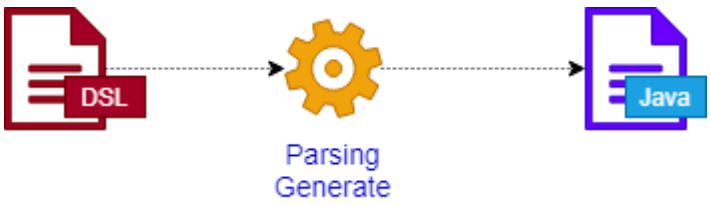


Figura 9. Modelo de implementación DSL

Acorde al enfoque y objetivos de un DSL, el DSL debe ser descrito o implementado usando un lenguaje tan parecido, como sea posible, al natural, incluyendo a la vez, nociones lo más cercanos (a los utilizados) al dominio de negocio que pretende resolver. También se debe considerar que el DSL debe tener un enfoque abierto (extensible), lo cual permita que se puedan crear especializaciones del DSL para adaptarlos a requerimientos particulares.
En las siguientes secciones, se procede a elaborar la propuesta de DSL; se utiliza un enfoque Top-Down, iniciando de lo más general, hasta llegar a un diseño particular.

3.4 Casos de prueba

Un caso de prueba o escenario, en el contexto de la solución, será la unidad básica sobre la que se plantea la verificación de una funcionalidad en particular del software. Un ejemplo, es validar que la clave de acceso a una aplicación cumpla las características requeridas, y sólo como efecto de ser válido se active la opción de Ingresar a la aplicación.

3.4.1 Estructura

Basado en la propuesta de Gherkin, para cada funcionalidad o característica implementada en la aplicación le puede corresponder uno o más casos de prueba o escenarios. De aquí en

adelante se usa la palabra escenario como un sinónimo de caso de prueba. Para cada escenario de prueba se tiene: a) unas precondiciones, que describen el estado del sistema antes de iniciar, b) una lista de pasos o acciones que se realizan, y c) finalmente una lista de pasos, acciones o verificaciones que permiten validar el cumplimiento o no de las condiciones esperadas.

En la siguiente tabla se describen cada uno de los elementos que se usarán al referirse a un caso de prueba.

Característica	Describe la funcionalidad o característica relacionados a los diferentes escenarios de prueba.
Escenario	Se usar para identificar de manera única al caso de prueba, y este agrupa. Una Característica puede requerir uno o más escenarios.
Descripción	Descripción corta que permite describir la funcionalidad a validar con la finalidad de documentación. Un escenario requiere una descripción
Precondiciones	Las precondiciones describen el estado actual de los componentes del software antes de que la que prueba inicie.
Pasos de Prueba	Se describe uno a uno los pasos que se realiza en la aplicación para llevar a cabo la verificación solicitada.
Resultado Esperado	Describe las condiciones finales que debe tener el sistema o uno de sus componentes para dar por válido o no el caso de prueba.

Tabla 10 Elementos a implementar en el DSL

Así, utilizando la aplicación de prueba, podemos describir los siguientes escenarios de prueba, basados en las características implementadas.

3.4.2 Ejemplos

Usando la definición anterior, y la aplicación de prueba, se describe a continuación, para una característica implementada, los escenarios de prueba:

Característica:

Calcular el importe de la hipoteca

Escenario: **1**

Calcular el importe de hipoteca dado el plazo en meses

Descripción:

El usuario ingresa los datos requeridos, y selecciona la opción de meses para indicar el plazo de la hipoteca.

Precondiciones:

La opción "Importe del Préstamo" ha sido seleccionada

El campo de ingreso "Cuota mensual" no está habilitado

Pasos de Prueba:

Se ingresa el valor \1000 en el campo "Importe del Préstamo"

Se ingresa el valor \12 en el campo "Tasa de Interés"

Se ingresa el valor \240 en el campo "Plazo hipoteca"

Se selecciona la opción \Meses

Se presiona sobre el botón Calcular

Resultados:

Se observa que el campo Cuota mensual muestra 1200.17

Escenario: **2**

Para calcular el importe de hipoteca dado el plazo en meses, es obligatorio haber ingresado un valor para Tasa de Interés

Descripción:

El usuario ingresa los datos requeridos, excepto la Tasa de Interés, y selecciona la opción de meses para indicar el plazo de la hipoteca

Precondiciones:

La opción "Importe del Préstamo" ha sido seleccionada

El campo de ingreso "Cuota mensual" no está habilitado

Pasos de Prueba:

Se ingresa el valor \1000 en el campo "Importe del Préstamo"

Se ingresa el valor \240 en el campo "Plazo hipoteca"

Se selecciona la opción \Meses

Se presiona sobre el botón Calcular

Resultados:

```
Se observa una notificación que indica que el valor de Tasa de
Interés No fue ingresado.
```

Aunque los ejemplos desplegados se encuentran descritos en lenguaje natural, no es viable usarlo directamente como solución. Se tendrá que encontrar un equilibrio entre la flexibilidad a permitir, y la simplicidad que se espera para su implementación. Entre más apegado al lenguaje natural, el universo de posibilidades en cuestión de configuración utilizables, hará más difícil la implementación, es decir será mucho más complejo convertir el DSL en código ejecutable.

3.5 Conceptos del DSL

3.5.1 Primer Nivel

Extrayéndolos de los ejemplos propuestos en el punto anterior, podemos anotar que los elementos de primer nivel de nuestro DSL serán:

Término	DSL
Característica	Feature
Escenario	Scenario
Descripción	Scenario Description
Precondiciones	Given
Pasos de Prueba	When
Resultados	Then

Tabla 11. Conceptos principales el DSL

Si se usa un diagrama de clases, podríamos representar los conceptos del DSL en la siguiente figura:

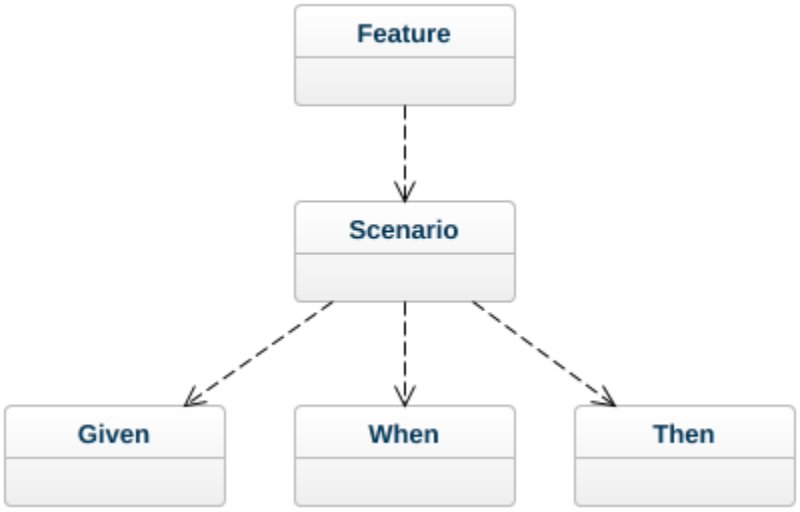


Figura 10. Diagrama de Clases para Conceptos DSL

Para cada característica (Feature) le corresponde uno o más escenario (Scenario), y los escenarios están compuestos por 3 secciones: precondiciones (Given), pasos de Prueba (When) y verificación de resultados (Then).

3.5.2 Segundo Nivel

En segunda instancia se puede observar que para las secciones Given, When y Then se conforman de instrucciones de interacción con los elementos de interfaz. En el caso de Given y Then, las instrucciones a considerar, validan el estado de la aplicación, esto es, verifican el cumplimiento de una condición. En la sección When se hace uso de instrucciones que tiene que ver con la ejecución de comandos o acciones sobre los elementos de interfaz.

De este texto podemos deducir que tendremos 2 tipos de instrucciones (pasos):

- Instrucciones de Validación (Validation Statement)
- Instrucciones de Acción (Action Statement)

Entonces tenemos que las instrucciones, ya sea de acción o validación, deben hacer referencia a los métodos, y atributos disponibles en cada componente de interfaz.

Para trasponer el escenario de prueba descrito en lenguaje natural, a un DSL, se requiere en primer lugar, limitar el alcance tanto de los componentes de interfaz permitidos, como de las acciones que se permiten utilizar. Así tenemos, que en la implementación permitiremos:

- TextView. – Corresponde a todos los tipos de caja de texto, a través de la cual el usuario ingresa información al sistema.
- Button. – Corresponde a todos los tipos de botones de comando, generalmente usados para iniciar un proceso, o confirmar información.
- CheckBox/RadioButton. – Tipo de elemento para selección.
- ListView. – Lista de valores, de la que se puede seleccionar una opción.

3.5.2.1 Instrucciones de Acción

Para cada uno de los elementos de interfaz de usuario le corresponden instrucciones de acción, estas se exponen en la siguiente tabla:

Elemento	Acción	Descripción
TextView	type input	Ingreso de datos desde el teclado.
Button	tap press click	Acción clic sobre el elemento.
CheckBox o RadioButton	choose	Seleccionar un valor
ListView	select	Seleccionar un valor
Toast Notification	show see	Mostrar un mensaje de notificación

Tabla 12. Lista de acciones por elemento de interfaz

La intencionalidad que se persigue es relacionar un concepto de acción, con el tipo de elemento. Así, un Button no permite el uso del método type, o un elemento de tipo ListView una acción tap.

Usando, los conceptos propuestos, se pueden armar las siguientes instrucciones:

```
# Ingresar un valor en un elemento tipo TextView
I type \1000 into Importe Préstamo

# Ingresar un valor en un elemento tipo TextView
I input \1000 to Importe Préstamo

# Clic o Selección de un elemento tipo Button
I tap over Calcular

# Seleccionar un valor de una lista de valores
I select Pago a Plazos in Lista Meses

# Seleccionar un valor de entre un grupo de CheckBox o RadioButtons
I choose Meses
```

3.5.2.2 Instrucciones de Validación

Las instrucciones de validación, permiten verificar el estado o contenido del elemento de interfaz. Desde el punto de vista técnico se basan en evaluar los valores actuales de los atributos o propiedades de un componente de interfaz.

Listamos los atributos que pueden ser verificados de acuerdo a cada tipo de componente soportado:

Elemento	Atributo	Descripción
TextView	enabled	Indica si el componente se encuentra habilitado para ser utilizado
	visible	Indica si el componente se encuentra visible
	content	Indica el texto contenido

Button		No tiene ninguna validación en particular.
CheckBox o RadioButton	checked	Indica si el elemento esta seleccionado o no.
ListView	selected	Indica el texto que esta seleccionado en la lista de valores.
Notification	name	Texto que se muestra en el componente de notificaciones

Tabla 13. Atributos o propiedades que se pueden verificar

Los atributos Enable y Visible, son aplicable a todos los componentes, por eso se marcan en un color diferente en la tabla.

Ahora, usando las instrucciones de validación se permite:

```
# Validar que el componente está visible
Component "Importe Préstamo" is visible

# validar que el componente está activo
Component "Tasa de Interés" is enabled

# validar que el checkBox Mes está seleccionado
Option "Mes" is checked

# validar que el valor "Pago Mensual" este seleccionado en la lista de
# valores
Value "Pago Mensual" is selected

# Mensaje "Tasa de interés es obligatorio"
Message "Tasa de interés es obligatorio" is showed
```

La evaluación del contenido de un textView puede requerir diferentes formas de condicionamiento, ya que dicha validación podría basarse en comparaciones parciales. Así tenemos:

- equals, el contenido es exactamente igual a.
- startsWith, el contenido inicia con.
- endsWith, el contenido finaliza con.
- contains, el texto contiene (está compuesto) por.

Cuando el contenido sea de tipo numérico se podrá usar las comparaciones:

- greaterThan, greaterEqualsThan .- Mayor que, Mayor igual que.
- lessThan, lessEqualsThan .- Menor que, Menor igual que.

```
# TextView contiene el valor "2034.11"
Content "Cuota Mensual" equals to "2034.11"

# Contenido TextView inicia con "2"
Content "Cuota Mensual" startsWith "2"
```

3.6 Lenguaje DSL

En primero lugar llega el momento de colocar un nombre al lenguaje, de aquí en adelante la referencia será aatDSL (Android Application Testing DSL), luego se proponen los elementos principales, que son:

```
Feature:
      [Descripción]
Scenario:
      [Identificación numérico y descripción]
Given:
      [Instrucción de Validación]
When:
```

```
      [Instrucción de Acción]
Then:
      [Instrucción de Validación]
```

Tabla 14. Palabras clave primer nivel del DSL

En este punto, antes de pasar a elaborar la sintaxis propuesta, es pertinente indicar las siguientes aclaraciones importantes sobre la sintaxis:

- Se usa el símbolo \ para diferenciar en donde empieza un valor, un artificio para lograr por ejemplo permitir espacios en blanco o texto vacíos.
- En donde se encuentre el texto [elemento], se entiende que corresponde al identificador del componente de interfaz relacionado a la instrucción.

Seguidamente detallamos, para las secciones que corresponden, los dos tipos de instrucciones interacción propuestas:

Action Statements, puede estar compuesto por uno o más de los siguientes elementos:

```
I [type|input] \value into [elemento]
I [tap|press|click] over [elemento]
I choose \value
I select \value in [elemento]
```

Tabla 15. Palabras clave instrucciones de acción del DSL

Validation Statements, puede estar compuesto por uno o más de los siguientes elementos:

```
[elemento] is enabled
[elemento] is visible
Option \value is checked
Value \value is selected at [elemento]
Message \value is showed
Content [elemento] [equals|startsWith|endsWith..] \value
```

Tabla 16. Palabras clave instrucciones de validación del DSL

Para asociar más de una instrucción de validación se puede usar los operadores AND o BUT. No así las instrucciones de tipo acción que se asocian como pasos, que se ejecutan uno tras otro, y se definen en diferentes líneas del archivo.

Finalmente se usa el símbolo # para remarcar una línea de tipo comentario, es decir, que se entiende como documentación, y se ignora en el proceso de Parsing.

3.7 Gramática de aatDSL

Para implementación del DSL se usa la funcionalidad Xtext [37], la gramática escrita en Xtext se coloca a continuación:

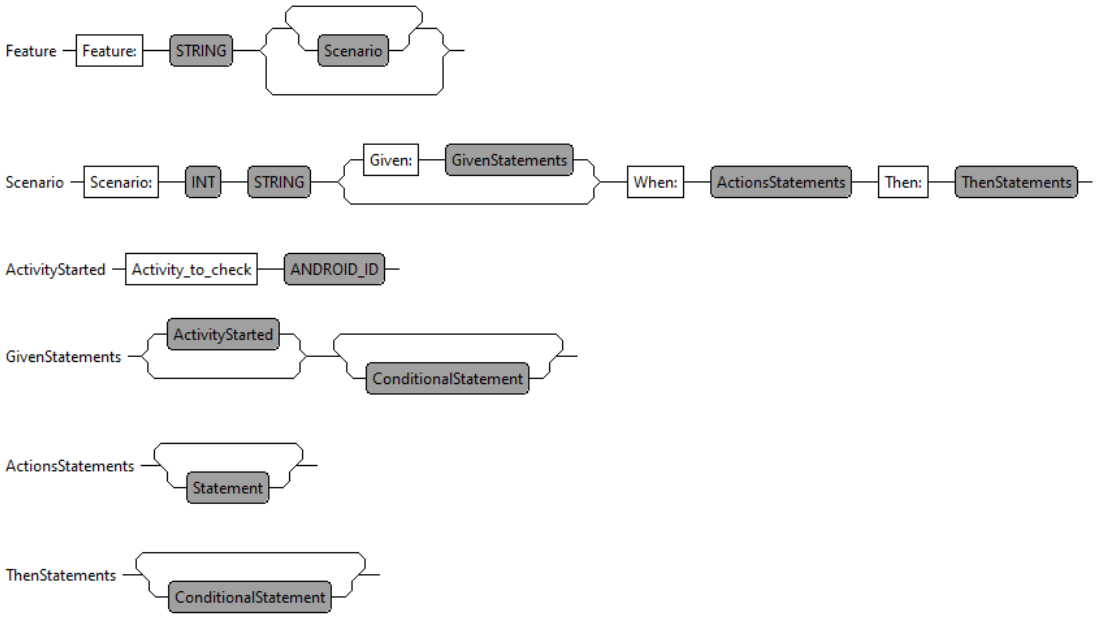


Figura 11. Gramática DSL, alto nivel

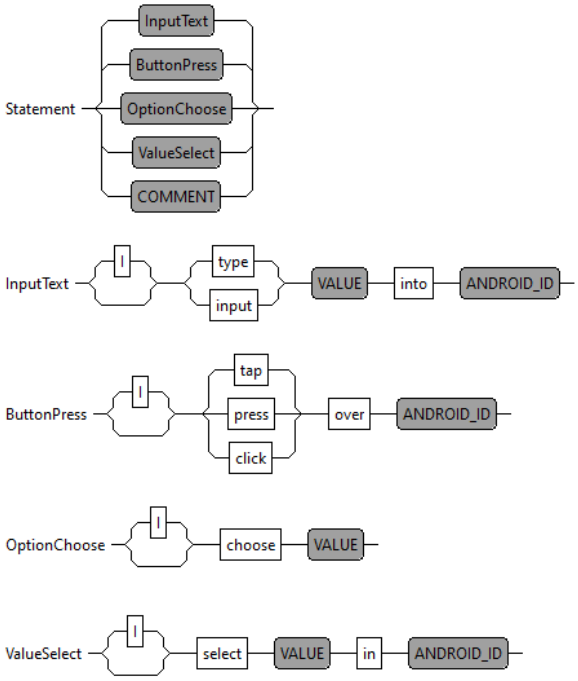
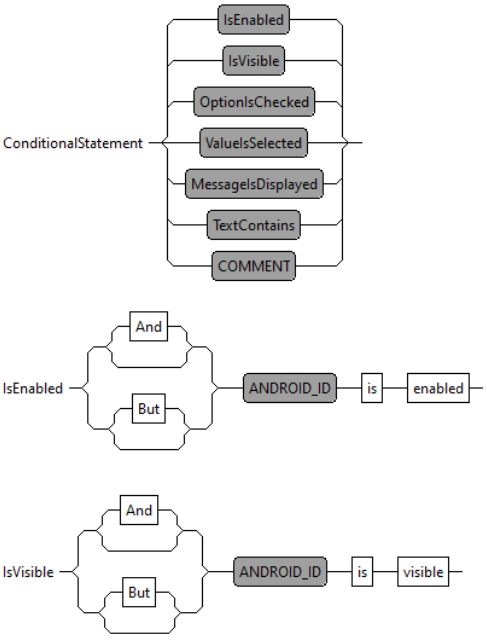


Figura 12. Gramática de Instrucciones de Acción



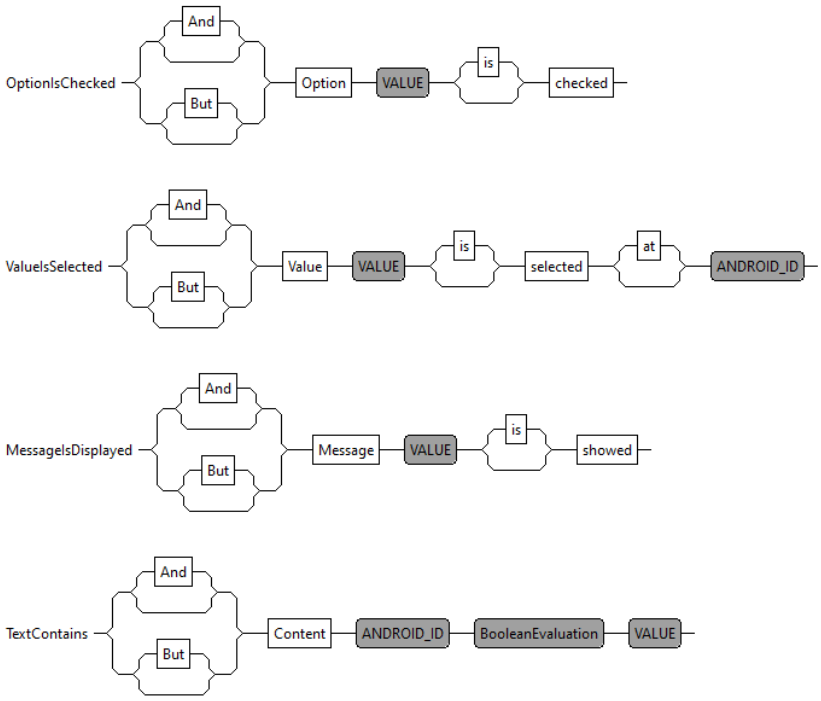


Figura 13. Gramática de Instrucciones de Validación

Usando la gramática propuesta se permite escribir archivos (con extensión feature) que describen los diferentes escenarios de prueba. A continuación, siguiendo en la línea de usar el ejemplo de aplicación propuesto, se colocan algunos "scripts" escritos utilizando aatDSL.

```
Feature:
       "Calcular el pago mensual de la hipoteca"

//
Scenario: 1
       "El usuario ingresa los datos requeridos, y selecciona la opción de meses para
indicar el plazo de la hipoteca"
Given:
       Option \"Importe del Préstamo" is checked
       But Option \"Cuota Mensual" is checked
When:
       I type \"1000" into Importe.del.Prestamo
       I type \"12" into Tasa.de.Interes
       I type \"240" into Plazo.Hipoteca
       I choose \"Mes"
       I press over Calcular
Then:
```

```
        Content Cuota.Mensual equals \"1200.17"

//
Scenario: 2
        "El usuario ingresa los datos requeridos, excepto la Tasas de Interés, y
selecciona la opción de meses para indicar el plazo de la hipoteca"
Given:
        Option \"Importe del Préstamo" is checked
        But Option \"Cuota Mensual" is checked
        Content Tasa.de.Interes equals \""
When:
        I type \"1000" into Importe.del.Prestamo
        I type \"240" into Plazo.Hipoteca
        I choose \"Mes"
        I press over Calcular
Then:
        Message \"Tasa de Interés No fue ingresado" is showed
```

3.8 Macro de Trabajo (Interpretación)

Una vez resuelto el primero elemento de la solución, esto eso aatDSL, a continuación, se explica la implementación de componentes encargado de la ejecución de los programas/archivos escritos con aatDSL.

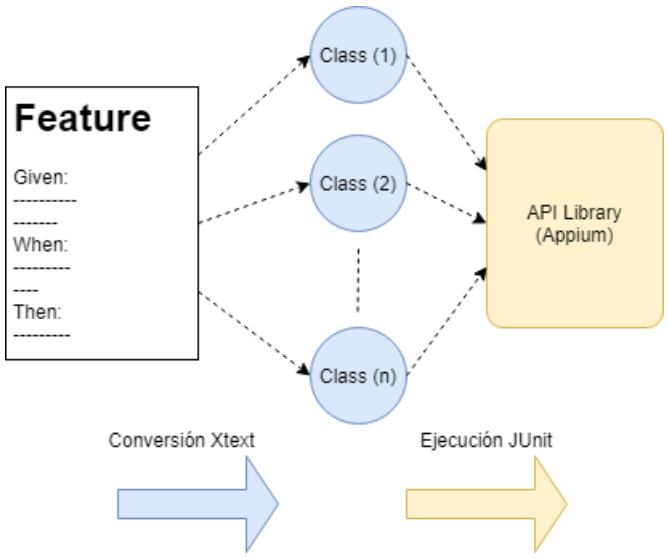


Figura 14. Esquema de generación/conversión y ejecución

Como se indica en la figura anterior, el esquema de solución toma cada uno de los escenarios de prueba descritos en el archivo se transforman en clases Java. Una clase por cada escenario descrito en el archivo de característica (feature). En cada una de las clases se convierten las instrucciones aatDSL en invocaciones a los métodos correspondientes de la librería API creada. A continuación, se describe en detalle los mecanismos de interpretación y ejecución usados para implementar la solución.

3.8.1 Esquema de la solución

Para simplificar la implementación de la lógica requerida, se propone crear un pequeño marco de trabajo basado en las librerías Appium. Las principales razones de proponer construir este componente son:

- Simplificar el proceso de conversión de lógica DSL a la requerida para la ejecución de la prueba.
- Extensibilidad, permitir a futuro se puedan implementar nuevos conceptos del DSL de una manera fácil.
- El uso de Appium presenta al menos dos complejidades que se desean solapar:
    - Identificación de los elementos, para identificar un elemento en la pantalla se requiere conocer o su nombre, o identificación interna.
    - Los tiempos de espera, esto involucra tiempos muertos en los que el escenario de prueba debe hacer una pausa o espera de resultado o refresco de los datos de la pantalla.

Para lograr la extensibilidad se propone que exista una concordancia uno a uno, entre las instrucciones implementadas en el DSL y los elementos existentes en el marco de trabajo. Esto es, que existirá 1 elemento en el marco de trabajo que embeba la funcionalidad Appium que deba ser invocada, así tenemos:

DSL	Librería Appium
`Id.Componente is visible`	`isVisible("Id.Componente")`
`Id.Componente is enabled`	`isEnabled("Id.Componente")`

Option \"[valor]" **is checked**	isChecked("[valor]")
Value \"[valor]" **is selected at** Id.Component	valueSelected("Id.Component","[valor]")
Message \"[mensaje]" **is showed**	isMessageDisplayed("[mensaje]")
Content Id.Component **contains** \"[valor]"	compare("Id.Component","contains","[valor]")

Tabla 17. Correspondencia DSL vs Librería Appium - Condicionales

DSL	Librería Appium
But Id.Componente **is visible**	isNotVisible("Id.Componente")
But Id.Componente **is enabled**	isNotEnabled("Id.Componente")
But Option \"[valor]" **is checked**	isNotChecked("[valor]")
But Value \"[valor]" **is selected at** Id.Component	valueNotSelected("Id.Component","[valor]")
But Message \"[mensaje]" **is showed**	isMessageNotDisplayed("[mensaje]")
But Content Id.Component **contains** \"[valor]"	notCompare("Id.Component","contains","[valor]")

Tabla 18. Correspondencia DSL vs Librería Appium – Condicionales Negados

DSL	Librería Appium
I type \"[valor]" **into** Id.Componente	type("Id.Componente","[valor]")
I press over Id.Componente	pressButton("Id.Componente")
I choose \"[valor]"	chooseOption("[valor]")
I select \"[valor]" **in** Id.Componente	selectValue("Id.Componente","[valor]")

Tabla 19. Correspondencia DSL vs Librería Appium - Acciones

3.8.2 Clases de tipo escenario

Para cada escenario propuesto, se crea una clase Java, que implementa las llamadas a los diferentes APIs de las librerías creadas sobre Appium. Cada clase, será implementada con

3 métodos, uno por cada sección del DSL. Y un método de invocación. Estas clases se crean de forma automática, y contendrán las correspondientes instrucciones que componen el escenario de prueba.

De esta forma, la clase implementada para el primer ejemplo del DSL utilizado en la sección "Gramática de aatDSL", quedaría de la siguiente manera:

```
//
//Given
//
void given(){
        isChecked("Importe del Préstamo")
        isEnabled("Importe.del.Prestamo")
        isNotChecked("Cuota Mensual")

}

//
// When
//
void when(){
        type("Importe.del.Prestamo","1000")
        type("Tasa.de.Interes","12")
        type("Plazo.Hipoteca","240")
        chooseOption("Mes")
        pressButton("Calcular")

}

//
// Then
//
void then(){
        compare("Cuota.Mensual","equals","1200.17")

}
```

3.8.3 Implementación Librería Intérprete de Appium

La implementación debe considerar los siguiente componentes o elementos:

- Android Driver, mecanismo a través del cual, Appium permite la interacción con los elementos de la aplicación Android.
- Escenario, corresponde a la seria de pasos a ejecutar y validar en un caso de prueba, el mismo que se encuentra compuesto por acciones y validaciones.
- Acción, representa a un evento a ejecutar sobre el elemento de pantalla.

- Validación, representa a evaluar una condición sobre el estado del elemento en pantalla.

Basados en estas consideraciones, e implementando un esquema flexible y extensible, se propone la solución descrita en el siguiente diagrama:

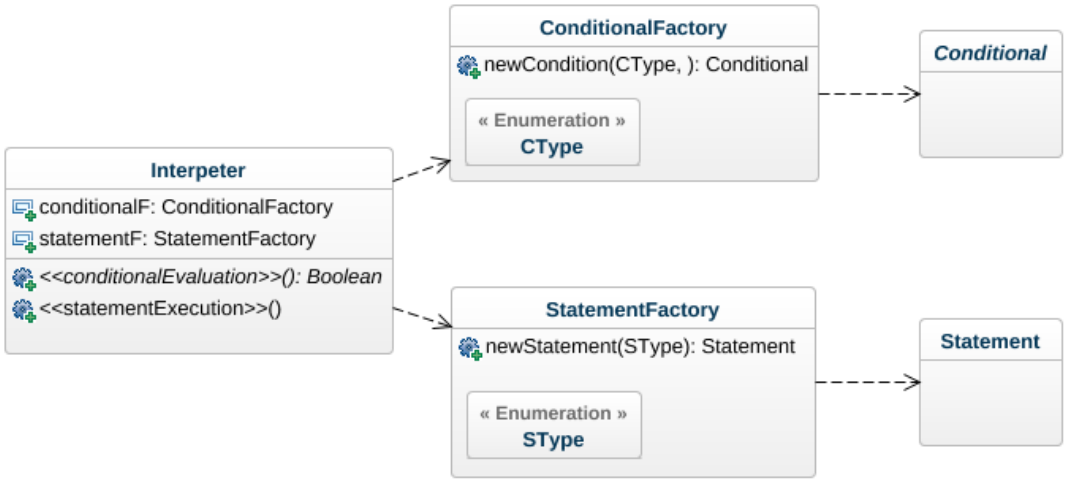


Figura 15. Diagrama de Clases, alto nivel, implementación AppiumApi

La clase principal será Interpeter, en esta se implementará un método por cada sentencia permitida en el DSL. Los métodos que invocan a la resolución de sentencias condicionales se hacen referencia con <<conditionalEvaluation>>, y los que se relacionan a las sentencias de ejecución se indican con <<statementExecution>>. Se utiliza 2 clases que, junto al uso de enumeraciones, implementan el patrón Factory con el objetivo de crear una instancia de la Condición o Instrucción que corresponda. La clase abstracta Conditional encierra la lógica común entre las instrucciones de evaluación de condiciones, y de manera similar la clase abstracta Statement para las instrucciones de ejecución.

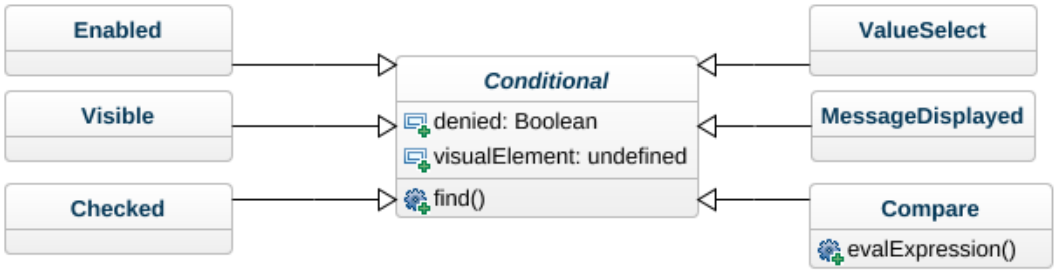


Figura 16. Diagrama de Clases, instrucciones de evaluación de condiciones

Dos atributos importantes de la una condición son:

- Negado (denied), usado para instanciar una instrucción que se acompaña de la palabra reservada But en el DSL.
- Elemento Visual (visualElement), el que representa al elemento de interfaz sobre el que se evalúa la condición.

Se debe notar que el caso de la instrucción Compare se requiere agregar lógica que permita evaluar la expresión asociada (igual, mayor que, contiene, etc.)

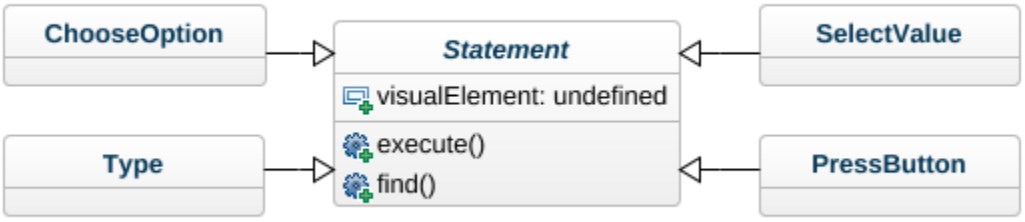


Figura 17. Diagrama de Clases, instrucción que ejecutan acciones

Tanto en las instrucciones de condición y ejecución aparece la necesidad de un método find, encargado de localizar, usando la funcionalidad Appium, el elemento sobre el cuál se debe aplicar la correspondiente instrucción

3.8.3.1 Buscar un elemento

El primer requerimiento a implementar es la búsqueda de un elemento en la pantalla. Para esto se plantea poder hacer la búsqueda utilizando ya sea uno de los atributos del mismo:

- resource-id. Corresponde al identificador único del elemento, o
- content-description. Atributo utilizado para describir al elemento, o
- Xpath. Se accede a través del árbol de elementos en la correspondiente actividad, en el que se incluye el tipo de componente y su descripción.

3.8.3.2 Ejecutar una Acción

Al ejecutar una acción interviene el Escenario, el Intérprete y las librerías Appium. La intención del Intérprete, es que sea extensible, y para esto, se propone que haya una implementación por cada instrucción de tipo acción.

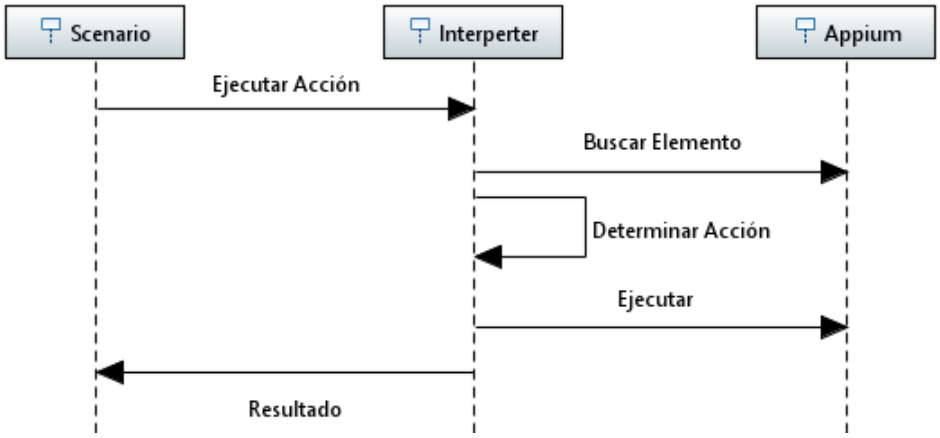


Figura 18. Diagrama de Secuencia, operación Ejecutar Acción

3.8.3.3 Ejecutar una Validación

Similar a la ejecución de acciones, la ejecución de una validación también debe ser extensible, para lo cual se propone la relación 1 a 1 entre la sentencia del escenario y la implementación, por tal motivo a cada sentencia de validación o acción le debe corresponder una implementación. En la validación se posee también un mecanismo de evaluación, el que comprueba o no que se cumpla la condición indicada. Así el diagrama de secuencia se expresa de la siguiente manera:

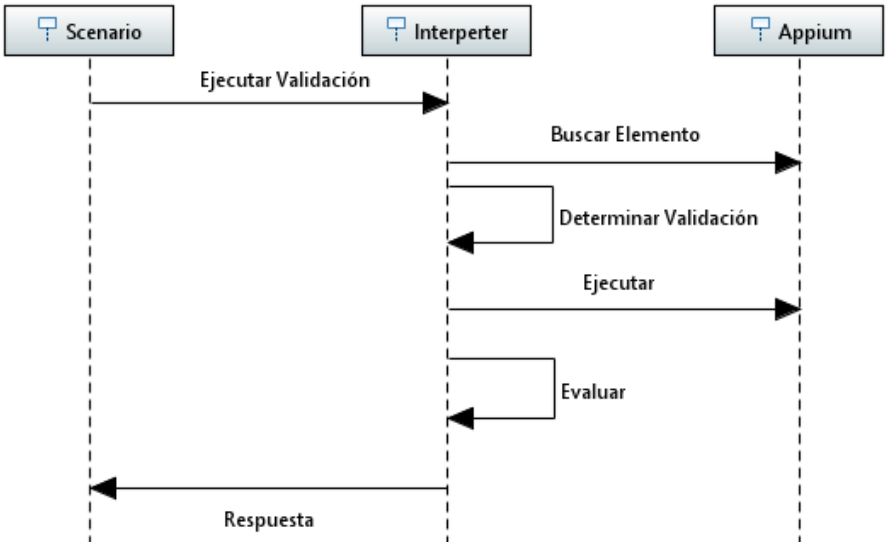


Figura 19. Diagrama de Secuencia, operación Ejecutar Validación

3.8.3.4 Escenario

Finalmente se implementa la clase abstracta escenario, que tiene 2 atributos:

- Settings, para cargar e instanciar al Appium Driver, con los elementos de inicialización (Tipo de dispositivo, Sistema operativos, Aplicación, Actividad). Se implementa usando una clase Driver que utiliza el patrón de diseño Singleton. Esta clase toma los elementos requeridos para crear una instancia de Appium Driver, desde un archivo de propiedades aatDSL.properties
- Interpreter, permite interactuar ya sea con las instrucciones de validación o de acción.

Esta clase, para mantener la abstracción, implementa un método que corresponda a cada instrucción posible en el DSL.

Figura 20. Clase Scenario, métodos abstractos

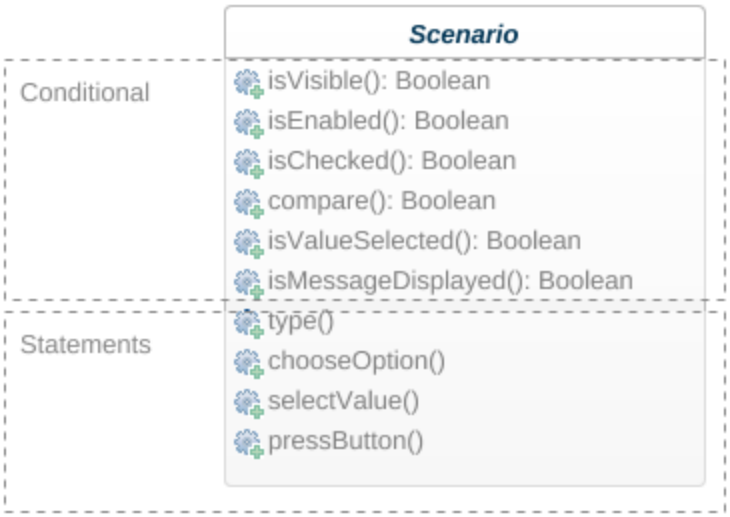


Figura 21. Clase Scenario, métodos que ejecutan operaciones

Cada clase <<ScenarioImpl>> que herede de la clase escenario, deberá implementar los métodos given, when y then. Para esto, se hará uso de las referencias a los métodos indicados en la figura los cuales corresponden a los mecanismos de ejecución para ya sea realizar las validaciones o la ejecución de eventos sobre los elementos de la interfaz de usuario.

3.9 Desarrollo

El esquema de la solución a implementar, se puede expresar utilizando el siguiente diagrama de componentes. Se provee al usuario final una interfaz basada en eclipse EMF (obtenida intrínsicamente del uso de Xtext), a través de la misma el usuario escribirá los "scripts" de prueba (archivos feature) utilizando el DSL propuesto. A continuación, Xtext genera el código Java, creando una instancia de la clase de conversión de código. Estas clases Java se basan en JUnit, y podrán ser ejecutadas desde el mismo IDE.

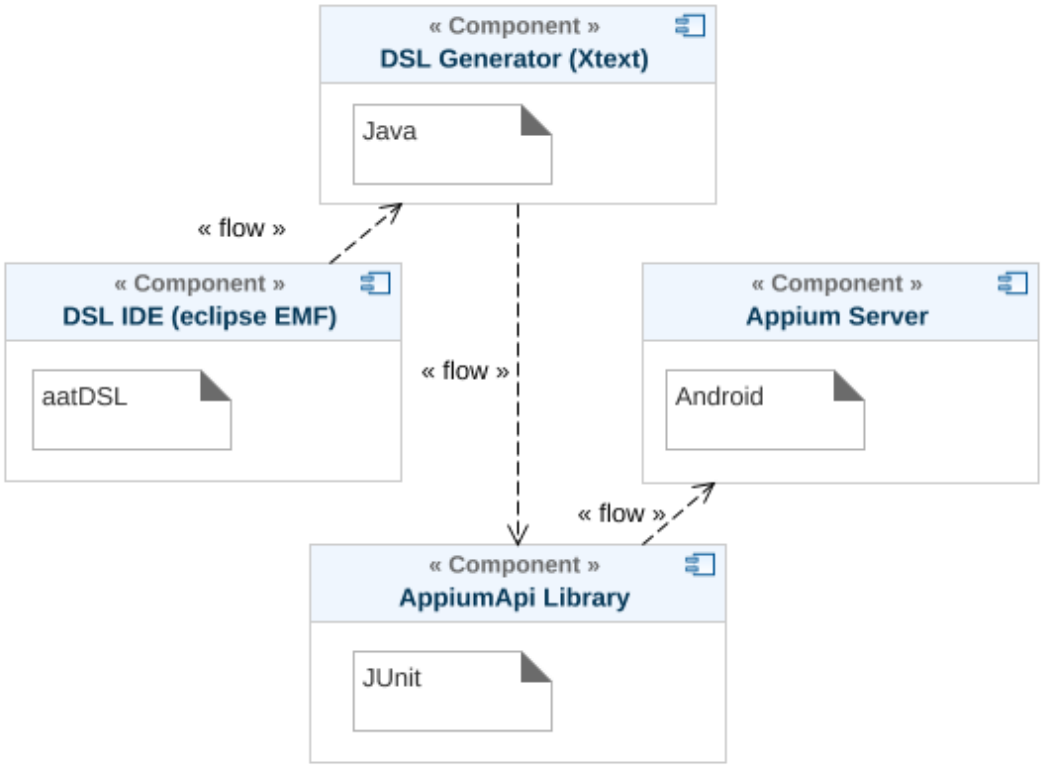


Figura 22. Diagrama de Componentes solución aatDSL

En tiempo de ejecución, el código Java se comunica con la librería AppiumApi, que a su vez resuelve sentencia a sentencia contra las definiciones o elementos provistos por el servidor Appium

3.9.1 Utilización

Al usar Xtext se obtiene además las ventajas de EMF (Eclipse Modeling Framework), lo que permite crear una instancia eclipse IDE como interfaz de usuario. Xtext genera todo el entorno de trabajo requerido, en este entorno el usuario puede crear archivos con extensión feature, los cuales deben cumplir con las reglas de la gramática propuesta.

Una vez que el archivo es grabado, se disparará la lógica que convierte las instrucciones DSL en clases java (Junit), en el directorio src-gen.

A este mismo proyecto se la anexa como librería al componente “Appium Librería”, descrito en la sección anterior, y haciendo uso del mismo entorno el usuario puede ejecutarlas.

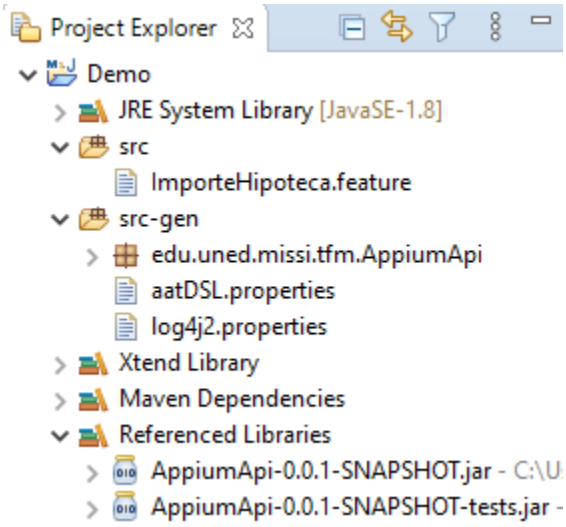


Figura 23. Configuración proyecto Eclipse – DSL

Donde:

- src corresponde a los archivos fuente del DSL,
- src-gen corresponde a los archivos Java generados

Se puede observar 2 archivo con extensión properties, descritos a continuación:

- aatDSL. archivo que contiene las propiedades de inicialización Appium

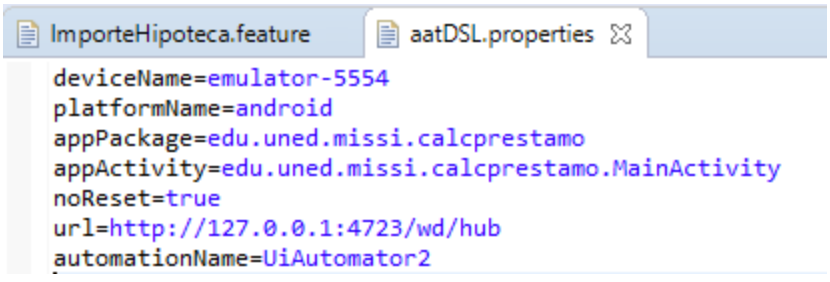


Figura 24. Archivo de propiedades, conexión y uso de servidor Appium

- log4j2. archivo con las definiciones Log4J2

En tiempo de ejecución, el usuario mira una interfaz similar a la mostrada en la siguiente figura.

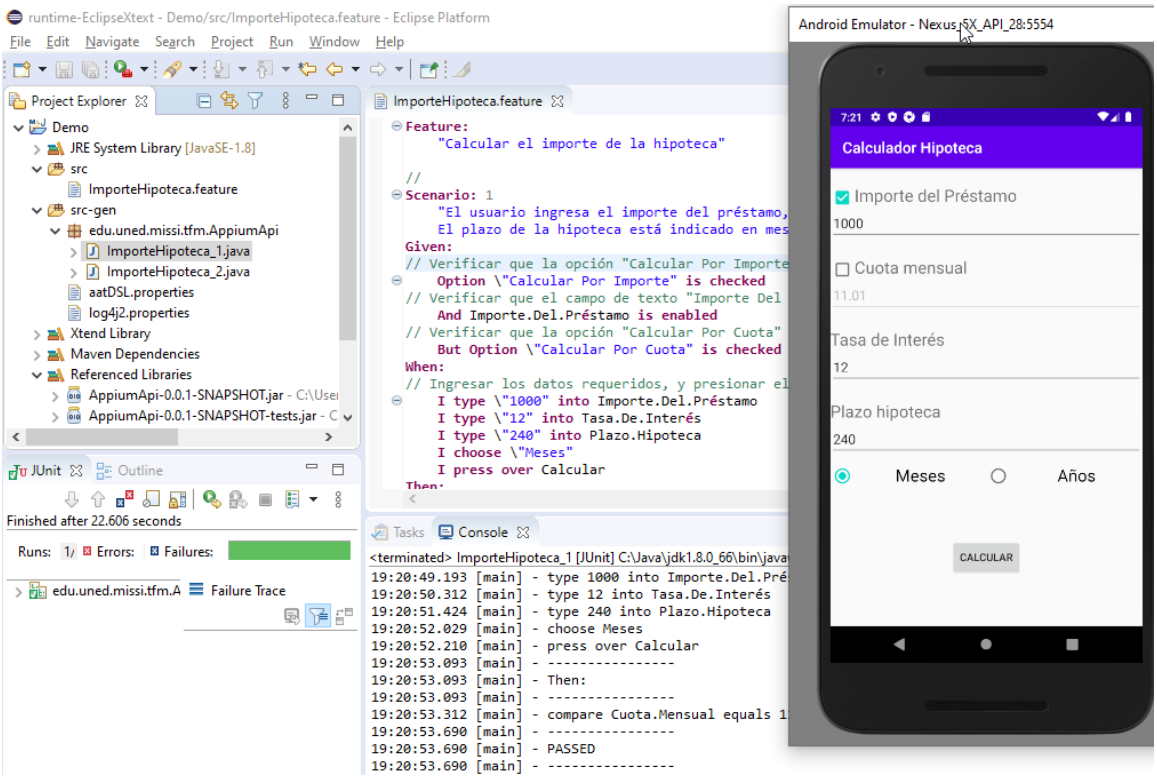


Figura 25. aatDSL en tiempo de ejecución

Como se puede observar en la figura anterior, la ventana Console muestra el avance de la prueba, mientras Appium va resolviendo las instrucciones, en este ejemplo, sobre un simulador de dispositivos.

Nota. – Al proyecto se le provee con las características Maven, para poder agregar el manejo de las dependencias requeridas.

3.9.2 Uso sobre Aplicación Ejemplo

Primero definimos nuestro archivo de escenarios. En este ejemplo hacemos referencia a la característica de calcular el pago mensual, dado el importe de la hipoteca.

```
Feature:
       “Calcular pago mensual de la hipoteca”

//
Scenario: 1
       “El usuario ingresa el importe del préstamo, la tasa, y plazo.
       El plazo de la hipoteca está indicado en meses”
Given:
// Verificar que la opción “Calcular Por Importe” esté seleccionada
       Option \”Calcular Por Importe” is checked
// Verificar que el campo de texto “Importe Del Préstamo” esté activo
       And Importe.Del.Préstamo is enabled
// Verificar que la opción “Calcular Por Cuota” NO estés seleccionada
       But Option \”Calcular Por Cuota” is checked
When:
// Ingresar los datos requeridos, y presionar el botón calcular
       I type \”1000” into Importe.Del.Préstamo
       I type \”12” into Tasa.De.Interés
       I type \”240” into Plazo.Hipoteca
       I choose \”Meses”
       I press over Calcular
Then:
// La cuota calculada debería ser 11.01
       Content Cuota.Mensual equals \”11.01”


Scenario: 2
       “El usuario ingresar los datos requeridos, excepto la Tasas de Interés,
     El sistema debe indicar que la Tasa de Interés de obligatorio”
Given:
// Verificar las precondiciones
       Option \”Calcular Por Importe” is checked
       But Option \”Calcular Por Cuota” is checked
       Content Tasa.De.Interés equals \””
When:
// Ingresar los datos requeridos, y presionar Calcular
       I type \”1000” into Importe.Del.Préstamo
       I type \”240” into Plazo.Hipoteca
       I choose \”Meses”
       I press over Calcular
Then:
// Se debe mostrar la notificación
       Message \”Tasa De Interés es obligatorio.” Is showed
```

El nombre del archivo es PagoMensual.feature. Una vez grabado el archivo, se generan 2 clases Java, un por cada escenario de prueba.

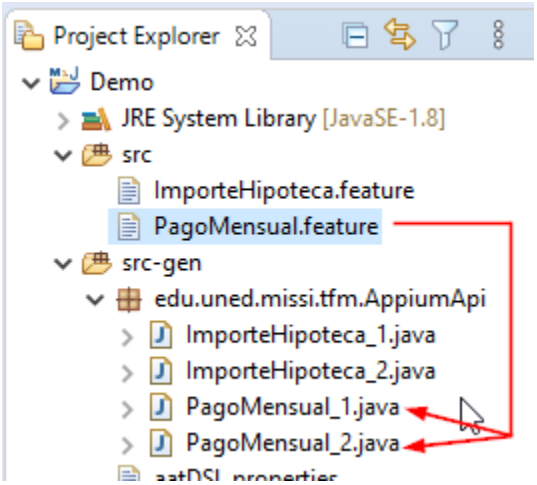


Figura 26. Archivo de prueba feature, genera 2 clases (1 por escenario)

```
/*
 No: 1
 Description:
 El usuario ingresa el importe del préstamo, la tasa, y plazo.
      El plazo de la hipoteca está indicado en meses
 Sun Jun 07 12:53:17 COT 2020
*/
public class PagoMensual_1 extends Scenario {

	//
	// Given
	//
	public void given() {
		isChecked("Calcular Por Importe");
		isEnabled("Importe.Del.Préstamo");
		isNotChecked("Calcular Por Cuota");

	}

	//
	// When
	//
	public void when() {
		type("Importe.Del.Préstamo", "1000");
		type("Tasa.De.Interés", "12");
		type("Plazo.Hipoteca", "240");
		chooseOption("Meses");
		pressButton("Calcular");

	}

	//
	// Then
```

```
	//
	public void then() {
		compare(“Cuota.Mensual”, Expression.equals, “11.01”);

	}

}


/*
 No: 2
 Description:
 El usuario ingresar los datos requeridos, excepto la Tasas de Interés,
      El sistema debe indicar que la Tasa de Interés de obligatorio
 Sun Jun 07 12:53:17 COT 2020
*/
public class PagoMensual_2 extends Scenario {

	//
	// Given
	//
	public void given() {
		isChecked(“Calcular Por Importe”);
		isNotChecked(“Calcular Por Cuota”);
		compare(“Tasa.De.Interés”, Expression.equals, “”);

	}

	//
	// When
	//
	public void when() {
		type(“Importe.Del.Préstamo”, “1000”);
		type(“Plazo.Hipoteca”, “240”);
		chooseOption(“Meses”);
		pressButton(“Calcular”);

	}

	//
	// Then
	//
	public void then() {
		isMessageDisplayed(“Tasa De Interés es obligatorio.”);

	}
}
```

3.9.2.1 Ejecución Primer Escenario

Se selecciona la clase, y se ejecuta utilizando la funcionalidad Junit,

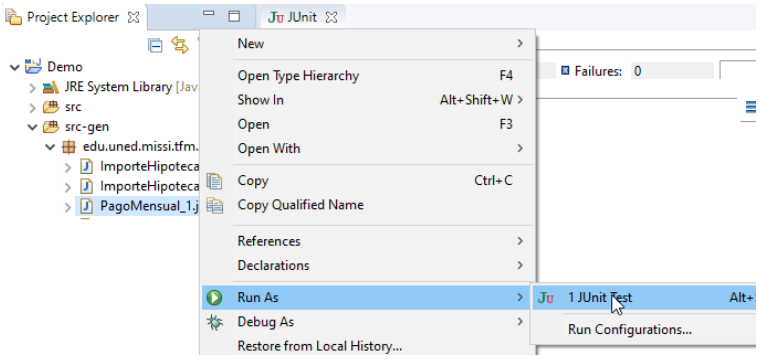


Figura 27. Ejecución del Primer Escenario

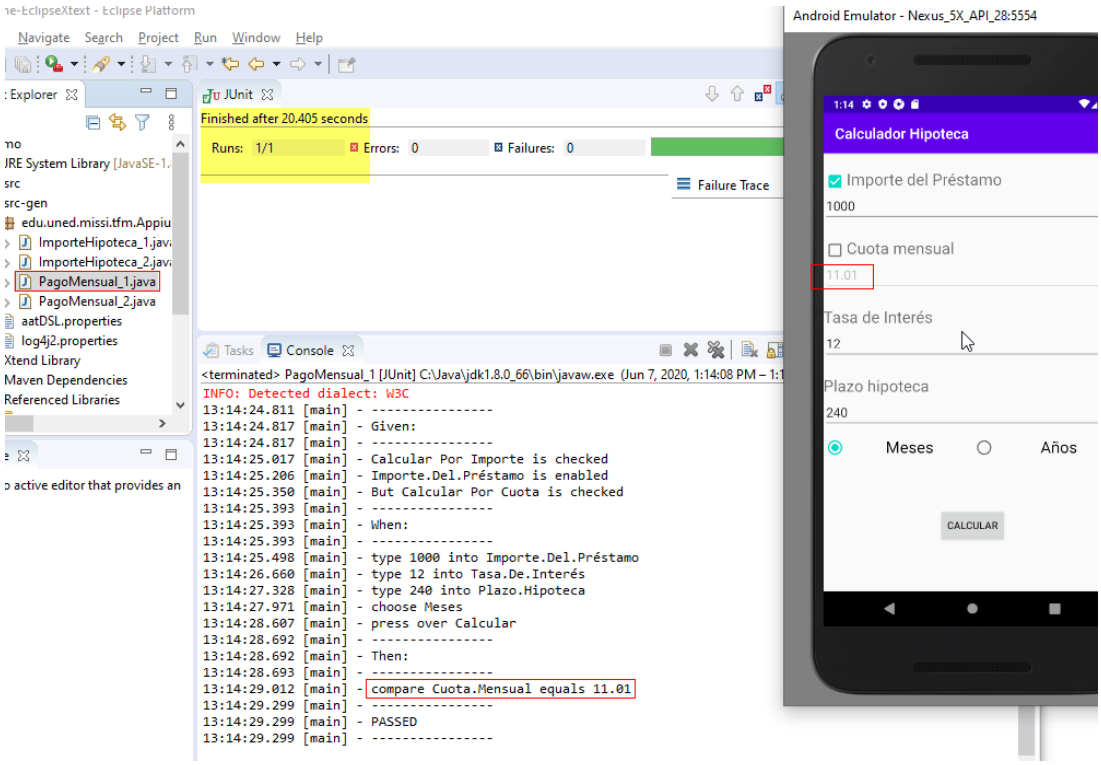


Figura 28. Resultados de ejecución del Primer Escenario

3.9.2.2 Ejecución Segundo Escenario

Se puede observar que la ejecución fue correcta, y cumplió con las instrucciones descritas en el escenario aatDSL.

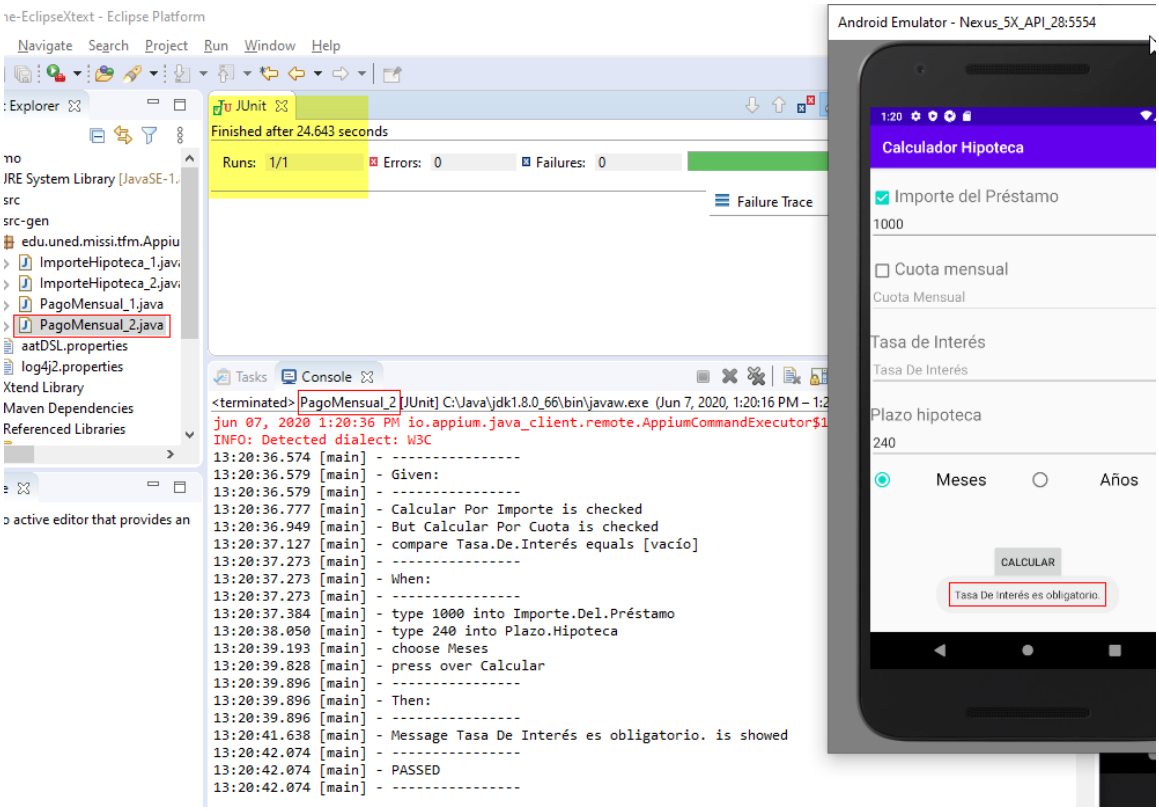


Figura 29. Resultados de ejecución del Segundo Escenario

4 Utilización DSL

Una vez creado el DSL, en este capítulo se desea mostrar el uso del mismo con dos diferentes enfoques, distintos al usado en la aplicación ejemplo del capítulo anterior. En primera instancia vamos a utilizar el DSL para realizar pruebas sobre una aplicación existente, y en segundo lugar usaremos para mostrar su utilidad en el contexto de desarrollo de aplicaciones orientadas por el comportamiento BDD.

4.1 Aplicación Existente

En este ejemplo vamos a usar la aplicación "Ingresos vs Gastos" [38]. Esta aplicación permite llevar registros de los gastos e ingresos de una persona, y poder evaluar la evolución en el tiempo de las finanzas personales. Vamos a probar una de las características que es la creación de un Gasto y dentro de esta dos posible escenarios. El primer escenario a probar consiste en agregar un Gastos dado un nombre y un valor, y el segundo escenario se usará para verificar la validación de "datos incompletos" cuando el usuario trate de crear un Gasto. Un punto importante anotar, es que al ser una aplicación existente se requiere de un emulador que permita su instalación y uso, para efectos demostrativos se usó el emulador BlueStacks.

A continuación, se muestra las pantallas de la aplicación como referencia:

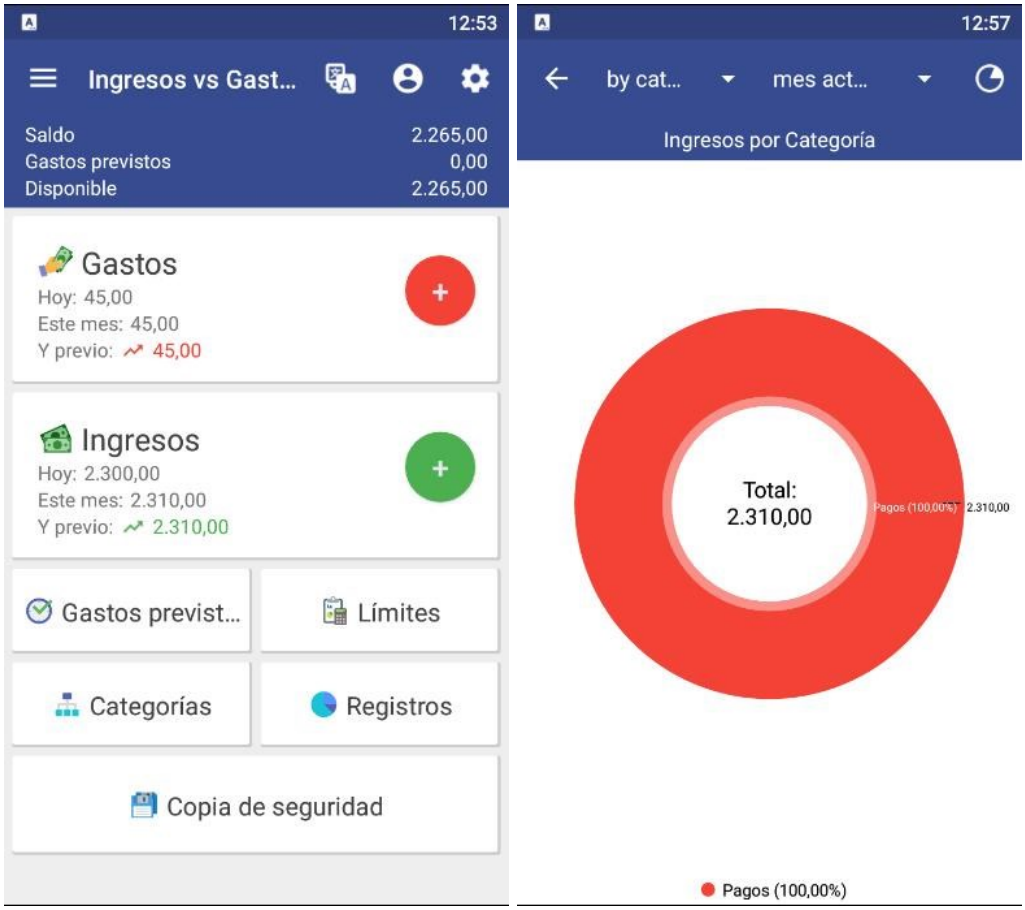


Figura 30. Aplicación "Ingresos vs Gastos"

4.1.1 Identificación de los elementos

Para poder escribir el archivo feature es necesario contar con las referencias o identificador de los componentes de interfaz a ser utilizado. En este caso, al ser una aplicación existente, se requiere realizar una introspección, para extraer los identificadores de dichos componentes. Para hacer esto, se puede optar: a) por aplicar ingeniería inversa para obtener el código fuente, o b) usar "Appium Inspector Studio" para extraer esta información.

4.1.2 Elaboración de la solución

El primer paso es crear un nuevo proyecto, en la interfaz Eclipse IDE aatDSL, y ajustar el archivo aatDSL.properties indicando el nombre del Paquete y Actividad inicial de la aplicación a usar.

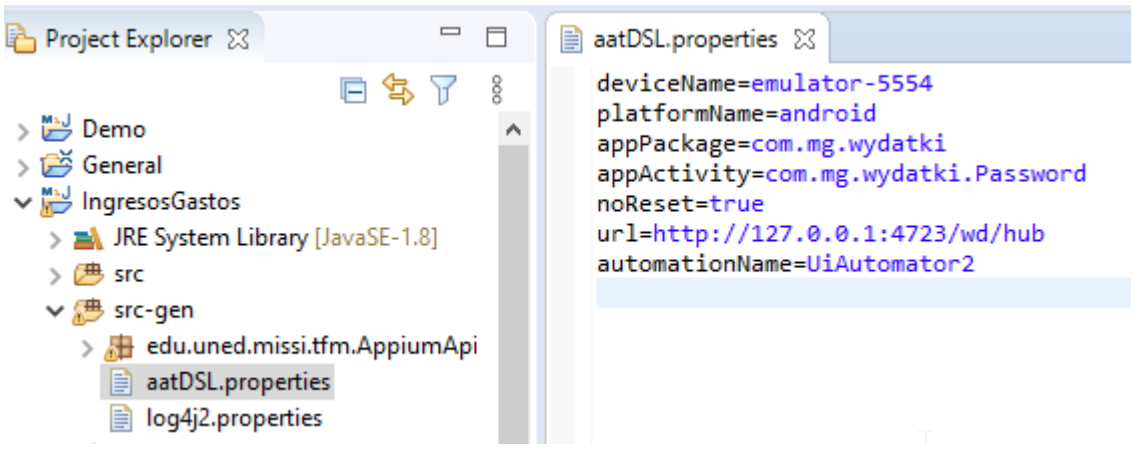


Figura 31. Aplicación Existente, proyecto Eclipse IDE

A continuación, se escribe el archivo feature, en el que se colocan los 2 escenarios a probar.

```
Feature:
       “Agregar un Gasto”
//
Scenario: 1
       “El usuario ingresa los datos completos para agregar un nuevo Gasto”
Given:
// Verificar que el botón agregar Gasto se encuentra presente
   bAddw is visible
When:
// Hacer clic sobre el botón
       I press over bAddw
// Ingresar el Costo, y el tipo de Gasto
       I type \”100.12” into etWKwota
       I input \”Queso” into etWOpis
// Presionar ok
       I press over ok
Then:
// Comprobar que el nuevo gasto aparezca en la lista
       Content text1wyd contains \”Queso”

//
Scenario: 2
       “El usuario ingresa agregar Gastos, pero no ingresa la cantidad.”
Given:
// Verificar que el botón agregar Gasto se encuentra presente
   bAddw is visible
When:
// Hacer clic sobre el botón
       I press over bAddw
// Ingresar el Costo, y el tipo de Gasto
```

```
        I input \"Queso" into etWOpis
// Presionar ok
        I press over ok
Then:
// Muestra mensaje, Datos faltantes
        Message \"Faltan datos por rellenar!" is showed
```

4.1.3 Ejecución del primer escenario

Una vez identificados los elementos de interfaz y escritos los correspondientes escenarios, se procede a su ejecución. A continuación, se muestra la ejecución del primer escenario:

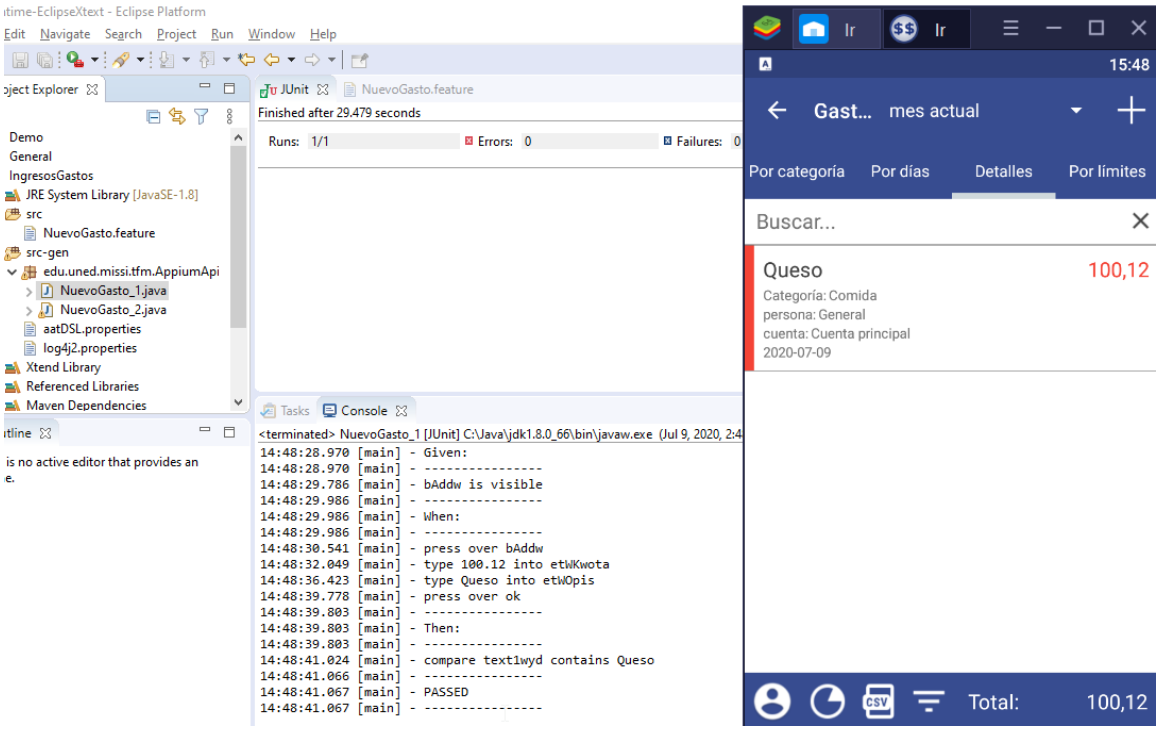


Figura 32. Ejecución del primer escenario, aplicación existente

Se puede observar en la ventana "Console" la ejecución paso a paso de las instrucciones que corresponden al script de prueba, y el resultado esperado "PASSED".

4.1.4 Ejecución del segundo escenario

Procedemos con la ejecución del segundo escenario, el cual como resultado esperado es la obtención de un mensaje al usuario indicando que la información está incompleta.

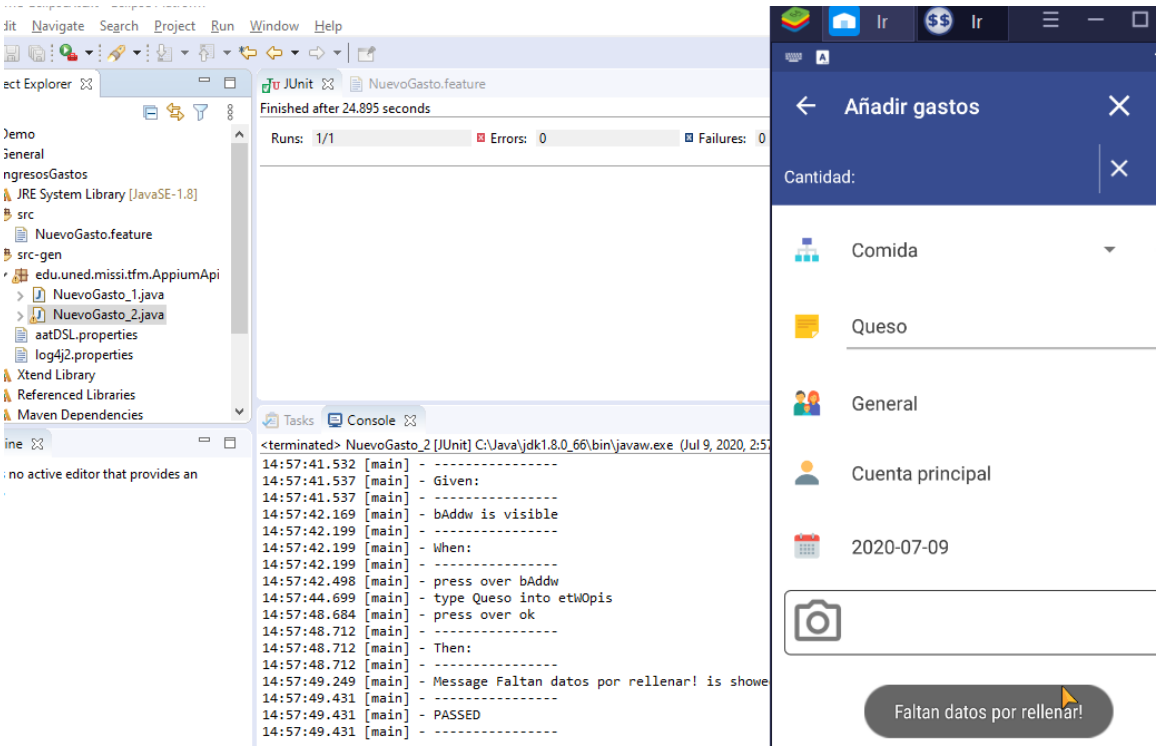


Figura 33. Ejecución del segundo escenario, aplicación existente

El resultado de la ejecución también es exitoso, la aplicación despliega una notificación que coincide con el texto indicado en el script feature.

4.2 Usando características BDD

En este ejemplo usaremos el DSL con la intención de generar una aplicación a partir de los escenarios de pruebas es decir con un enfoque BDD. La aplicación consiste en una lista de mercado, en la que se puede tener una o más listas, cada lista está conformado por ítems, los cuales a su vez contienen una descripción y una cantidad.

Para iniciar la elaboración de la aplicación se comienza realizando un bosquejo visual que muestre los componentes y elementos de interfaz esperados.

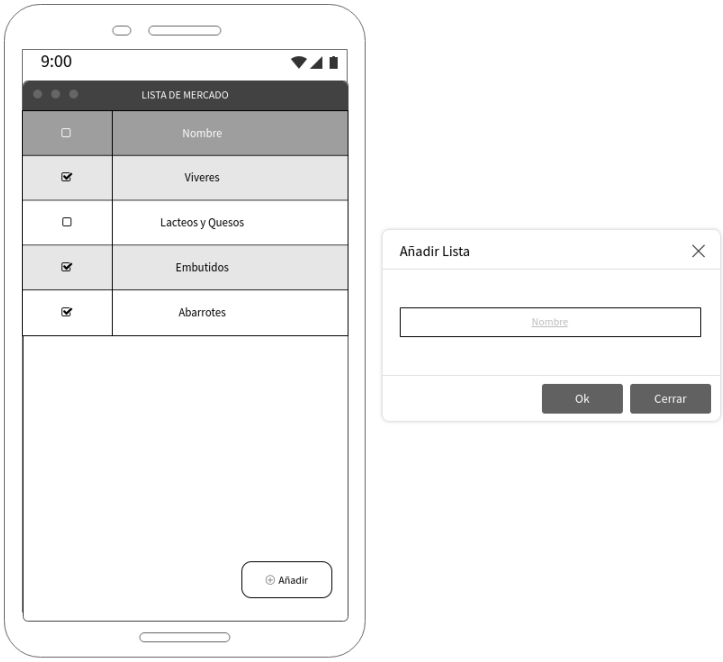


Figura 34. Diseño de la aplicación Lista Mercado – Añadir Lista

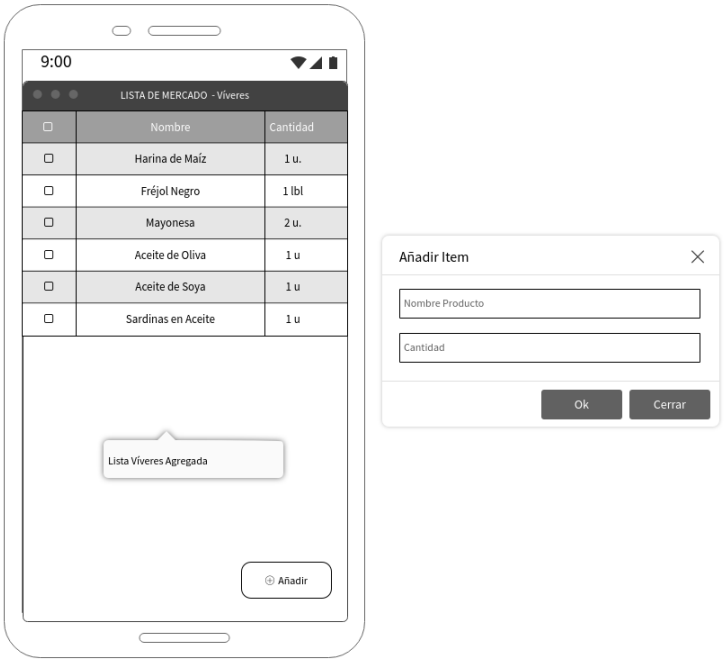


Figura 35. Diseño de la aplicación Lista Mercado – Añadir Ítem

A continuación, se resume la característica y los escenarios a implementar, el propósito es mostrar la utilidad de un DSL en el esquema BDD, por lo que la implementación funcional pretender ser lo más sencillas posible.

Característica:

Agregar una lista de mercado

Escenarios:

1. Con todos los datos requeridos, el sistema permite agregar una nueva lista.
2. Si el nombre de la lista no es ingresado, el sistema despliega un mensaje indicando que el nombre es obligatorio.

Característica:

Agregar un ítem a una lista de mercado

Escenarios:

1. Con todos los datos requeridos, el sistema permite agregar un nuevo ítem a una lista.

4.2.1 Proyectos Requeridos

Para implementar la solución requerimos la creación de 2 proyectos separados, el primer proyecto corresponde al creado sobre el Eclipse IDE aatDSL que corresponde a la interfaz en la que el experto de negocio escribe los scripts de aceptación funcionales. Al crear el proyecto se debe agregar el archivo aatDSL.properties en donde se colocan principalmente el nombre del paquete y actividad principal de nuestra nueva aplicación.

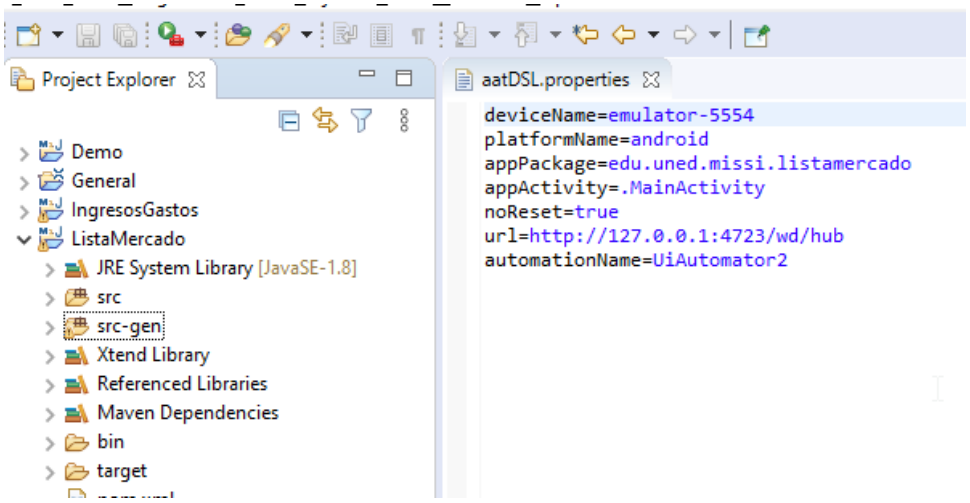


Figura 36. Nuevo proyecto eclipse IDE aatDSL IDE

El segundo proyecto, corresponde a la aplicación Android, se crea la primera versión de la aplicación, usando la plantilla "Empty Activity", tomando como partida el nombre del paquete y Actividad principal, descritos en el proyecto aatDSL.

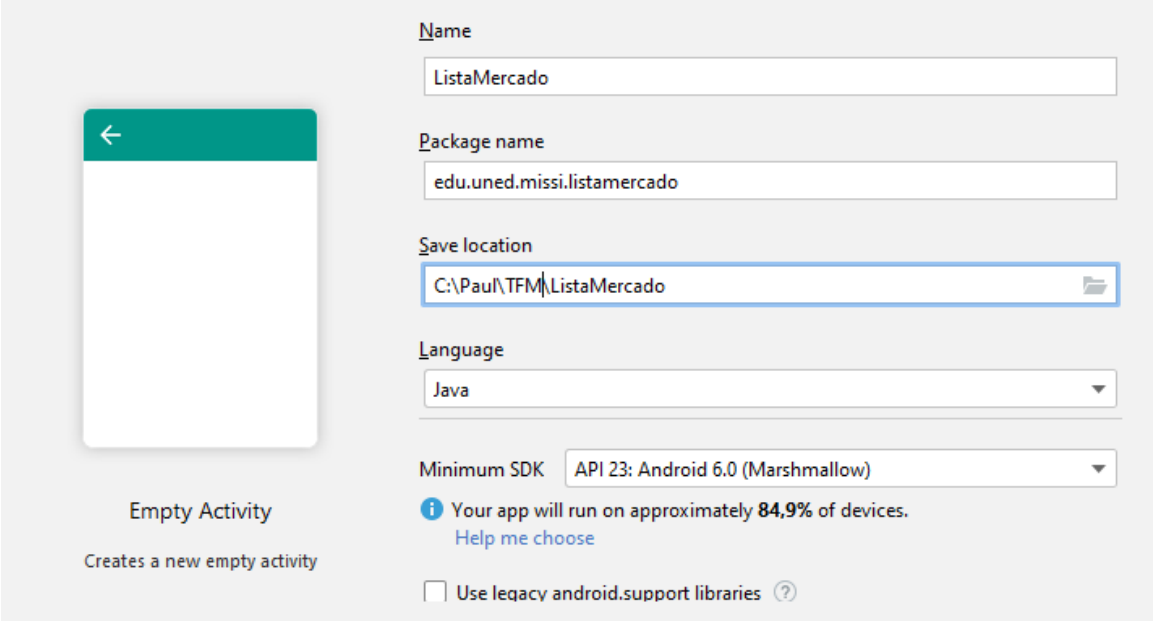


Figura 37. Nuevo proyecto Android Studio, Lista Mercado

4.2.2 Primera Característica

4.2.2.1 Implementar Primer Escenario

Para iniciar generamos el script feature con el que mostramos la funcionalidad esperada por la aplicación para cubrir el primer escenario,

```
Feature:
       “Crear una lista”

Scenario: 1
       “El usuario presiona el botón Agregar, e ingresa el nombre de la lista”
Given:
// Verificar que el botón agregar Gasto se encuentra presente
   Agregar.Lista is visible
When:
// Hacer clic sobre el botón agregar
       I press over Agregar.Lista
// Ingresar el Nombre de la Lista
       I type \”Víveres” into Nombre.Lista
// Presionar botón Aceptar
       I press over Aceptar
Then:
// Se despliega el mensaje, Lista Vivires agregada
       Message \”Lista Víveres agregada!” is showed
```

Procedemos a ejecutar el escenario, esperando que falla totalmente, ya que a esta altura aún no se ha implementado ninguna de la funcionalidad esperada.

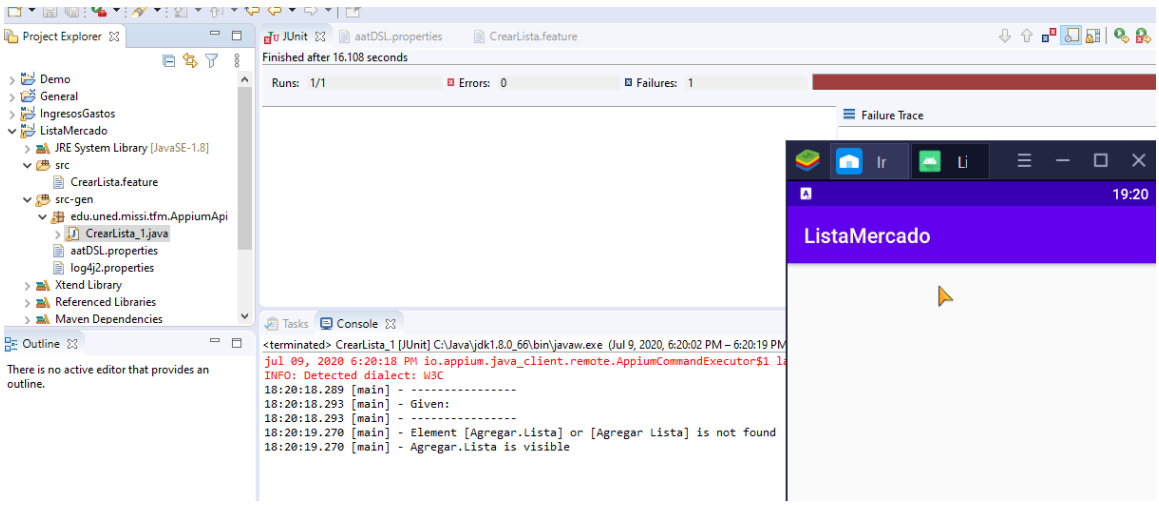


Figura 38. Primera ejecución lista de mercado – Agregar Lista.

Como se observa, el resultado es que la prueba falla totalmente, ya que ni siquiera el componente referenciado existen en la interfaz. Entonces, para proceder con una siguiente ejecución, se espera a que el desarrollador implemente las funcionalidades descritas en el script feature.

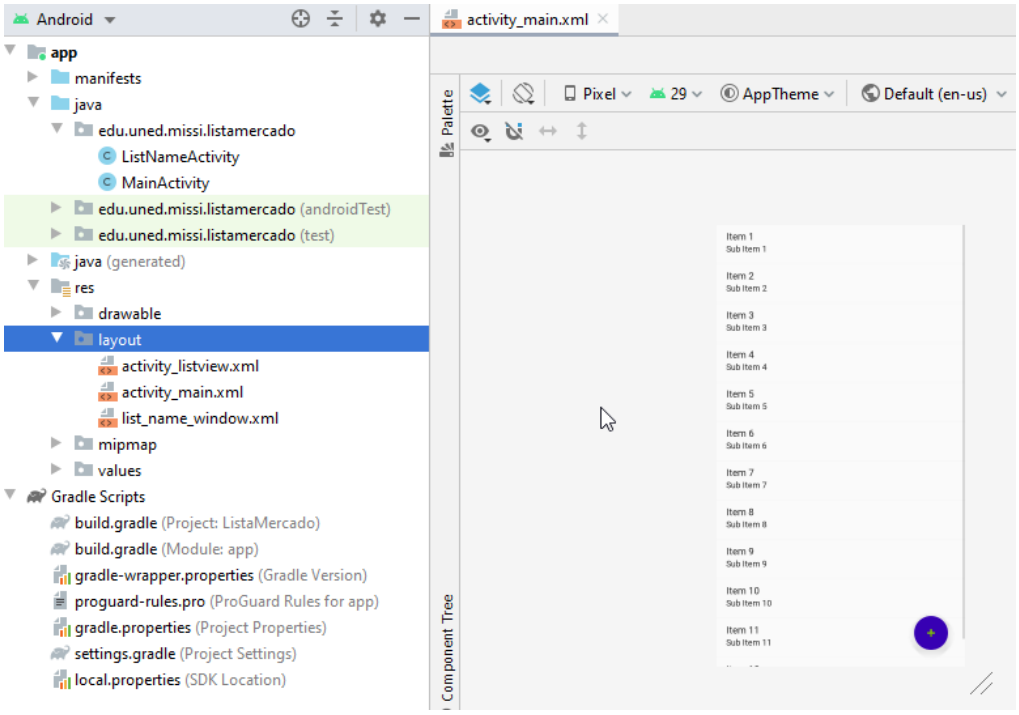


Figura 39. Implementación Escenario – Agregar Lista

Luego que la lógica requerida ha sido implementada, se procede a ejecutar nuevamente el caso de prueba,

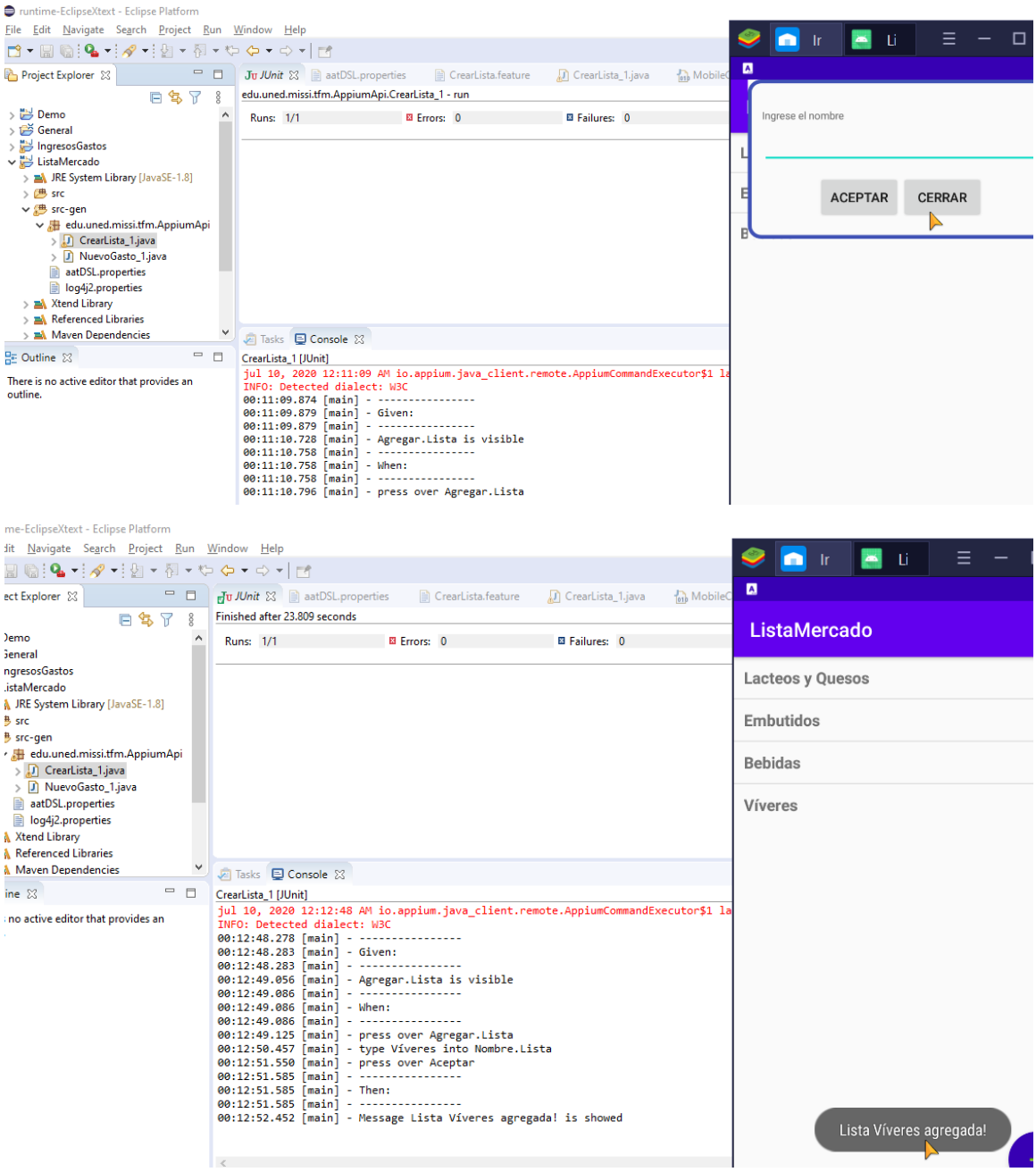


Figura 40. Ejecución del primer escenario – Agregar Lista

La intención de las 2 figuras anteriores es mostrar por un lado la interfaz de usuario en la que se ingresa el nombre de lista, y la segunda muestra el resultado de la ejecución.

4.2.2.2 Implementar Segundo Escenario

Para el segundo escenario, se usa la misma metodología, se implementa la lógica de acuerdo al requerimiento, escribiendo en primer lugar el escenario,

```
Scenario: 2
      “El usuario presiona el botón Agregar, y presiona el botón Aceptar”
Given:
// Verificar que el botón agregar Gasto se encuentra presente
   Agregar.Lista is visible
When:
// Hacer clic sobre el botón agregar
      I press over Agregar.Lista
// Presionar botón Aceptar
      I press over Aceptar
Then:
// Se despliega el mensaje, Lista Vivires agregada
      Message \”Nombre Lista es obligatorio!” is showed
```

Seguidamente se procede a ejecutar el escenario, y de manera similar falla, dado que la funcionalidad esperada aún no ha sido implementada.

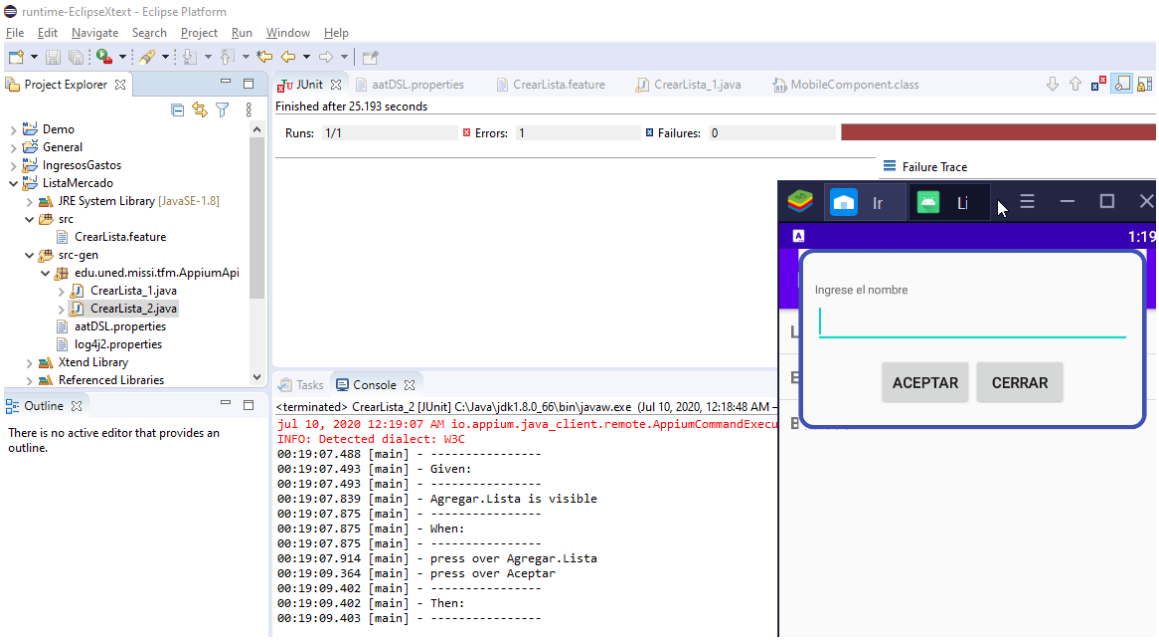


Figura 41. Primera ejecución del segundo escenario, lista de mercado.

Se procede a implementar la funcionalidad esperada,

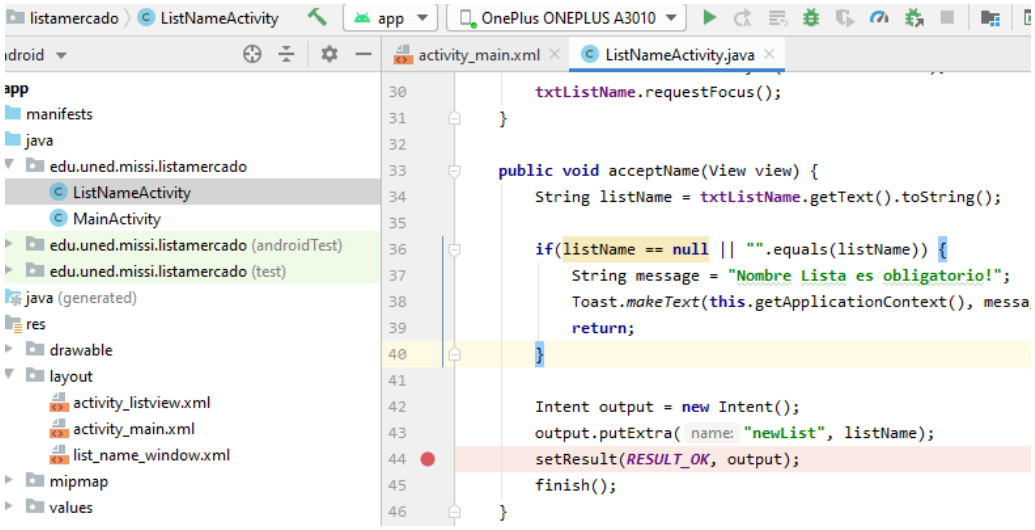


Figura 42. Implementación Escenario – Agregar Lista

y se vuelve a ejecutar el escenario de prueba.

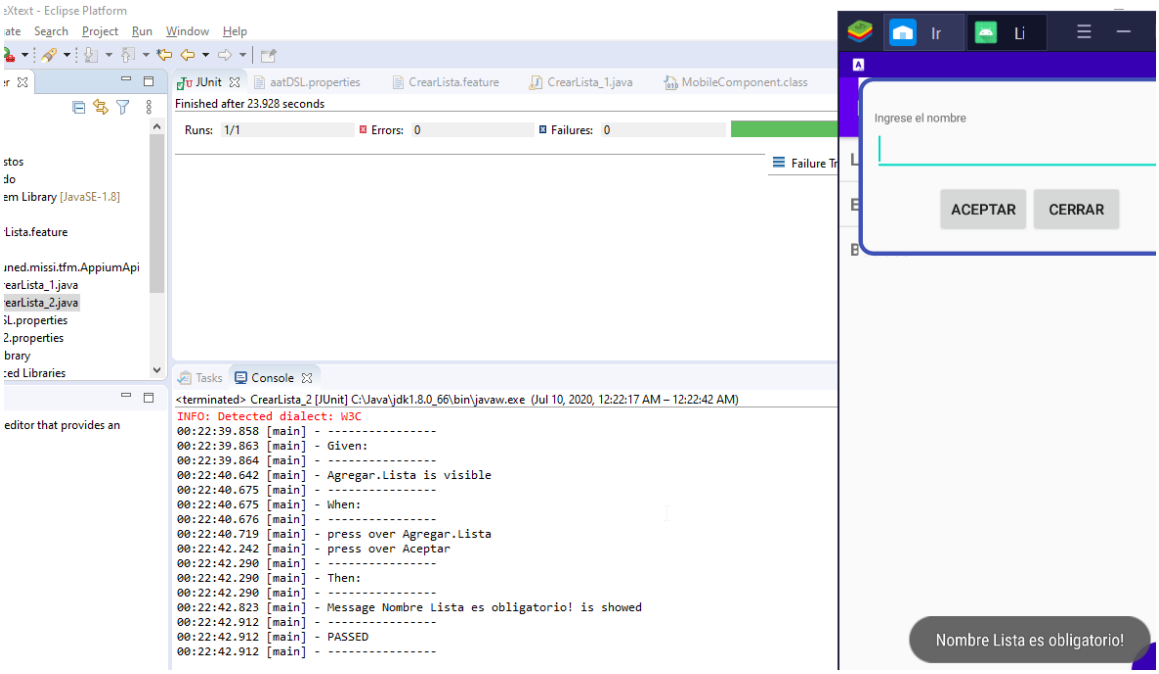


Figura 43. Ejecución del segundo escenario, lista de mercado.

Como resultado, una vez implementada la funcionalidad, el caso de prueba se marca como correcto, dado que el sistema muestra el mensaje de validación esperado.

4.2.3 Segunda Característica

4.2.3.1 Implementar Primer Escenario

Similar a la característica anterior, se procede primero a crear el archivo feature con el escenario que deseamos implementar.

```
Feature:
      “Agregar un Ítem a la Lista”

Scenario: 1
      “El usuario selecciona una lista, y agrega un ítem”
Given:
// Verificar que Listas.Mercado es visible
   Listas.Mercado is visible
When:
    I select \”Bebidas” in Listas.Mercado
    I 97apo ver Agregar.Item
    I input \”Coca Cola 1Lt” into Nombre.Producto
    I input \”2 u.” into Cantidad
    I press over Ok
Then:
// Se despliega el mensaje, Coca Cola 1Lt agregado
      Message \”Coca Cola 1Lt agregado a la lista!” is showed
```

Procedemos a ejecutar el escenario, y falla

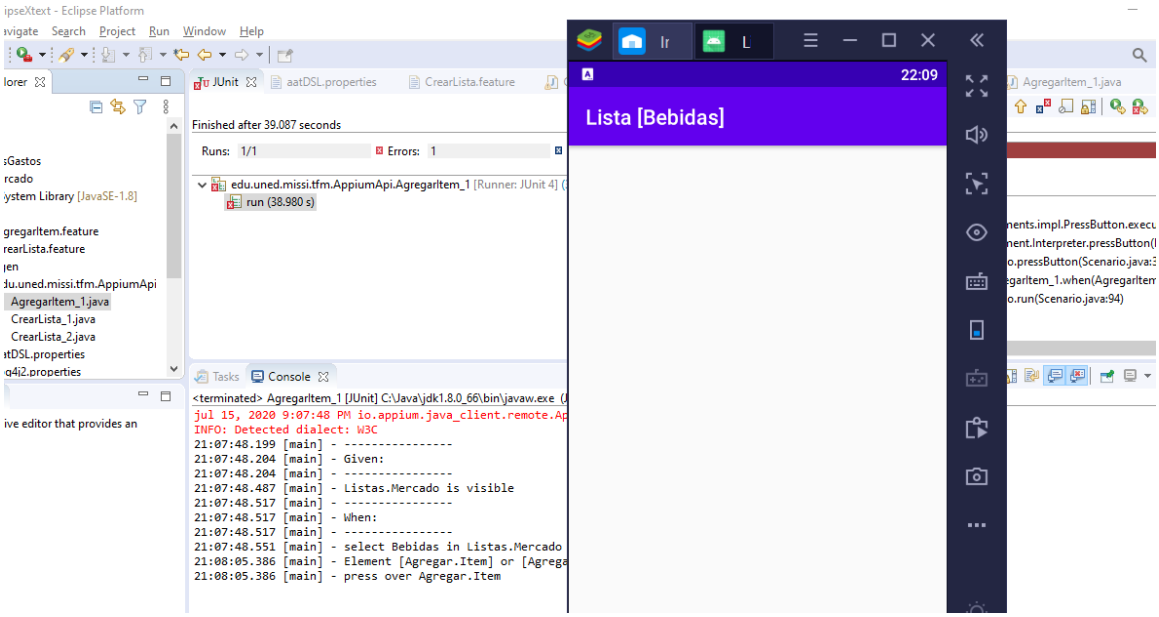


Figura 44. Primera ejecución lista de mercado – Agregar Ítem

Entonces se procede a implementar el código que debe cumplir con el escenario planteado,

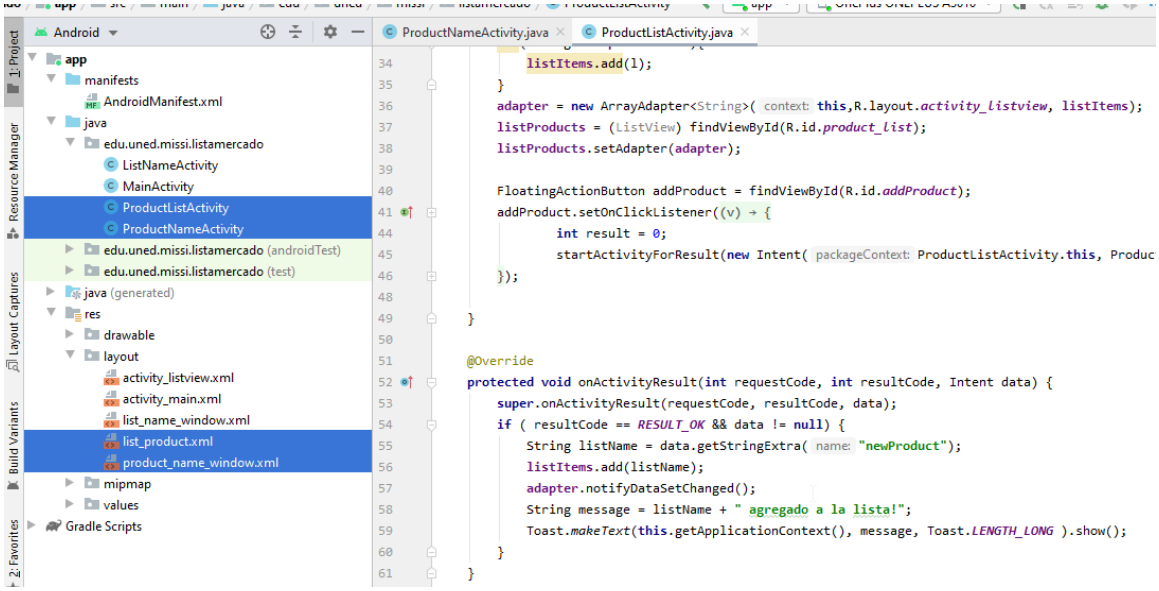
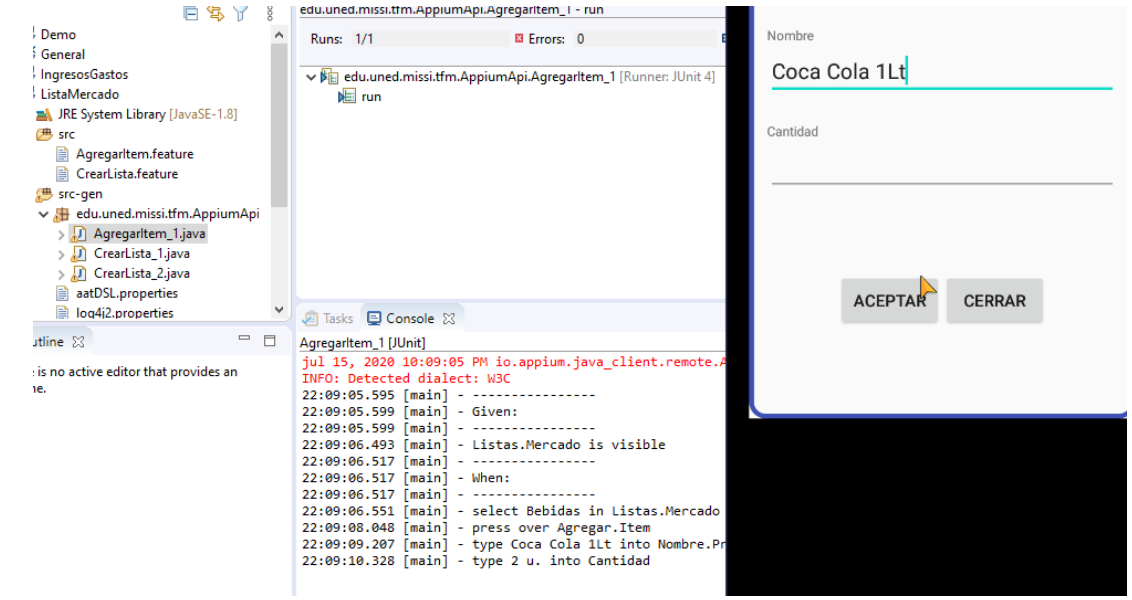


Figura 45. Primera ejecución lista de mercado – Agregar Ítem

Se procede a ejecutar nuevamente el escenario inicial, después de haber implementado la lógica que lo soporta:



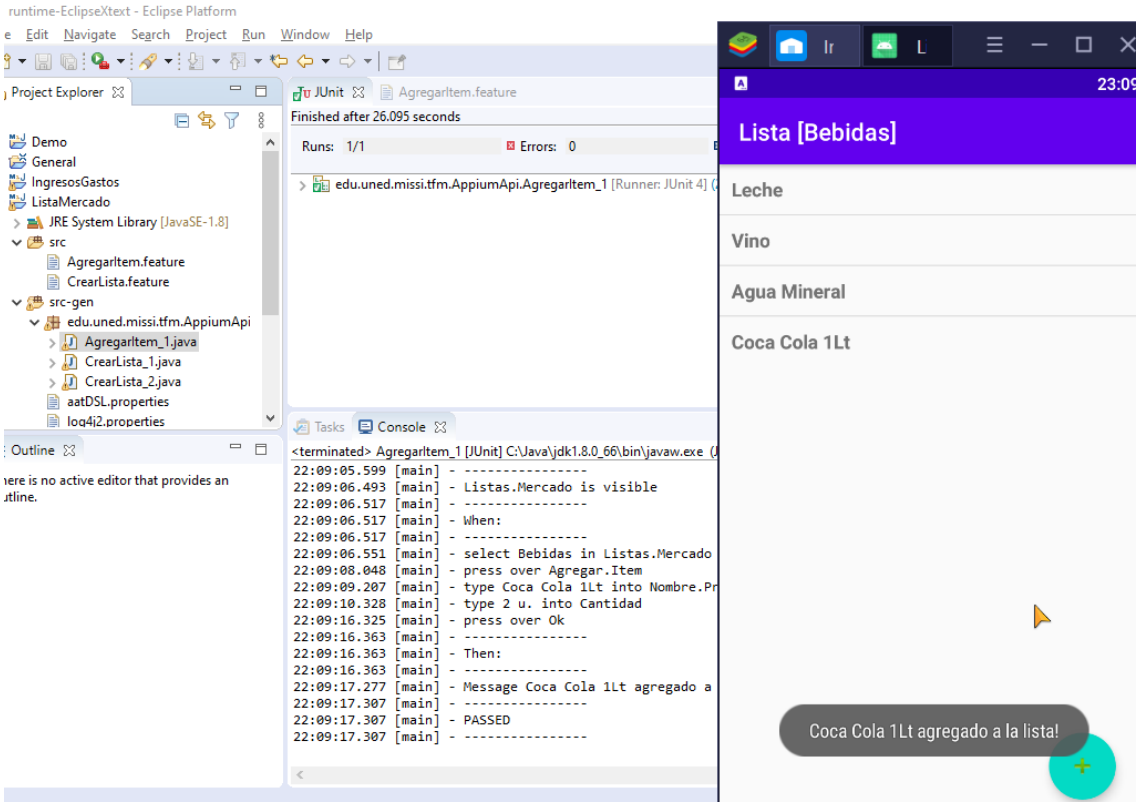


Figura 46. Ejecución del primer escenario – Agregar Ítem

Después de haber construido las partes de código que implementan la lógica esperada, se puede observar que el escenario de prueba resulta exitoso.

5 Conclusiones y Trabajos Futuros

Crear aplicaciones confiables, eficaces (que hagan lo que se esperan) en sistemas de negocio, se ha vuelto un factor crítico para la industria del software. Los principales usuarios de estas aplicaciones, son los clientes finales, por lo que proveer servicios a través de aplicaciones móviles es una necesidad intrínseca en cualquier tipo de industria. Como se ha mencionado para la construcción de estas aplicaciones es necesarios hacer uso de procedimientos ágiles, lo que claramente conlleva a la necesidad de encontrar mecanismos de validación de requerimientos versátiles y rápidos, con los cuales sea fácil encontrar posibles problemas y proveer las soluciones más adecuadas, antes de que las funcionalidades lleguen a los usuarios finales. Aquí entra en juego del DSL propuesto en esta disertación. Permite que el usuario de negocio describa de una manera ágil las expectativas funcionales que debe cumplir la aplicación; haciendo uso de un lenguaje similar al natural, en el que se describen los requisitos funcionales usando términos relacionados al dominio del problema (press, tap, show, check, etc)

Sobre DSL:
Si bien la intencionalidad del trabajo es mostrar la usabilidad de un DSL dentro del ámbito de la elaboración de pruebas en el proceso de construcción, también hemos evidenciados su usabilidad en la automatización de pruebas sobre aplicaciones existentes (para procesos de mantenimiento y soporte), y la factibilidad de poder usar el DSL para un proceso de construcción de aplicaciones sobre el enfoque BDD.
El usar un DSL reduce la brecha de comunicación que existe, por lo general, entre los usuarios de negocios y los desarrolladores. De un lado, trasladar (para el usuario de negocio) un escenario o caso de uso a código permite eliminar redundancia (pasos innecesarios o duplicados), también obliga a simplificar el uso de las funcionalidades (elementos de interfaz) para cumplir con los requerimientos, antes de que sean codificados. De otro lado, el proceso de trasladar (para desarrollador) un DSL a código final se simplifica, dado que el script le está describiendo explícitamente todo el contexto y ámbito

de automatización que se espera en la aplicación. El uso del DSL, conjuntamente con un adecuado entrenamiento, permite tanto al usuario de negocio como el desarrollador, hablar en términos comunes, evitando sobre todo las ambigüedades y malas interpretaciones. Implícitamente el DSL coloca límites a la interacción, encerrando en su gramática el conjunto de funcionalidades y elementos usables para cumplir con los requerimientos de negocio. Por ende, se anota una limitante al DSL en el contexto de llevar a cabo el proceso de pruebas en la construcción de aplicaciones (sí lo que se construyó no puede ser expresado en el DSL, entonces no podrá ser probado).

Alcance:

En principio, se debe indicar que en la propuesta desarrollada se ha tratado de mantener la simplicidad, no sólo en relación con la implementación, sino también el ámbito del alcance de la automatización lograda, usando un universo limitado de componentes y funcionalidades que pueden validarse (limitar el número de elementos de interfaz, y limitar el número de operaciones aplicables a dichos elementos). La intención, del trabajo era evaluar la factibilidad y usabilidad de contar con un DSL para la elaboración y ejecución de pruebas funcionales. La aplicabilidad del DSL al universo de funcionalidades y componentes disponibles en Android esta fuera del alcance de este trabajo.

Mejoras:

Desde el punto de vista funcional, hay varias mejoras de detalle que se pueden realizar a la herramienta propuesta, para mejorar su usabilidad, como, por ejemplo:

- Permitir que la ejecución se haga directamente desde el archivo feature
- Añadir funcionalidad de nuevos elementos, como el manejo del Scroll, o identificación de controles de tipo imagen.
- El uso de EMF permite que se puedan agregar características nuevas a la interfaz del IDE, como asistentes de codificación, mensajes de validación en línea, etc.
- Extender la funcionalidad de la librería AppiumApi para usarlo con otras plataformas
- Permitir el uso de componentes del lenguaje DSL en otros idiomas, como el español

En el contexto de la elaboración del DSL, se podría pensar en mejorar la gramática, para por ejemplo hacer uso de otros tipos de herramientas (análisis gramático o máquinas de aprendizaje) que puedan trasladar sentencias, en el contexto de intencionalidad, a las instrucciones que deben ejecutarse o implementarse. De esta forma se podría usar un rango más amplio de lenguaje, pero haciendo que los mecanismos descritos puedan analizar y generar a partir de la intencionalidad, las instrucciones a ejecutar. Por ejemplo:

```
If I type 3 at textbox value then a message will show “value does not permitted”
```

¿Se podría añadir una capa de traslación, que, a partir del texto escrito por el usuario, se genere las instrucciones DSL a implementar?

En el ámbito de las pruebas también se debería poner atención en las funcionalidades requeridas con respecto al seguimiento del proceso de pruebas. Esto implica no sólo la administración de los escenarios de pruebas (archivos feature) sino su relación con las diferencias ejecuciones realizadas, y sus correspondientes resultados (reportes e informes de seguimiento).

En cuanto a las aplicaciones móviles, cuando se tratan en más detalle, las aplicaciones actuales requerirían la implementación de eventos o elementos más complicados con relación a la interfaz de usuario, como manejo de gestos o eventos relacionados con sensores (reconocimiento facial, lectores de huellas, etc.) Uno de los trabajos futuros podría tratar sobre la viabilidad de implementar o extender el DSL para aprovechar dichos elementos, entre ellos la información contextual, como parte de los escenarios. Sería incluso posible evaluar si Appium es o no el framework más adecuado para la automatización para pruebas que usen información de contexto.

Final:
Como se ha descrito a lo largo del trabajo, el realizar pruebas sobre aplicaciones móviles no es una tarea trivial, que pueda considerarse de baja importancia, o no fundamental en el

proceso de su construcción. Los aspectos relacionados son su éxito tales como: la eficiencia, efectividad, calidad, usabilidad; van a estar ligados, de manera directa no sólo con la productividad, sino también a términos como: la imagen, aceptación, y reputación; todos estos se verán reflejados finalmente en la adhesión de nuevos clientes, o la adquisición de nuevos servicios por parte de clientes existentes. Por esta razón, poseer herramientas o mecanismos de pruebas "ágiles" (que puedan llevarse a cabo a la par con el ciclo de construcción de la aplicación) es primordial; por ende, el proveer un DSL acorde a la experiencia funcional, facilitará en gran medida las tareas relacionadas al proceso de pruebas de aplicaciones.

6 Lista de Referencias

[1] J. Bo, L. Xiang y G. Xiaopeng, «MobileTest: A Tool Supporting Automatic Black Box Test for Software on Smart Mobile Devices,» de *Second International Workshop on Automation of Software Test* , Minneapolis Minnesota, 2007.

[2] D. Amalfitano, N. Amatucci, A. M. Memon, P. Tramontana y A. R. Fasolino, «A general framework for comparing automatic testing techniques of Android mobile apps,» *Journal of Systems and Software,* vol. 125, nº Elsevier, pp. 322-343, 2017.

[3] T. Leung, Beginning PowerApps: The Non-Developers Guide to Building Business Mobile Applications, Apress, 2017.

[4] A. S. a. R. T. A. Hammershøj, «Challenges for mobile application development,» de *14th International Conference on Intelligence in Next Generation Networks*, Berlin, 2010.

[5] A. Ahmad, K. Li, C. Feng, S. M. Asim y A. Y. a. S. Ge, «An Empirical Study of Investigating Mobile Applications Development Challenges,» *IEEE Access,* vol. 6, nº 2018, pp. 17711-17728, 2018.

[6] V. V. a. M. Krbecek, «Development methodologies of mobile applications,» de *International Conference on Interactive Mobile Communication, Technologies and Learning* , San Diego, CA, USA, 2016.

[7] L. Corral, A. Sillitti y G. Succi, «Agile Software Development Processes for Mobile,» Springer-Verlag Berlin Heidelberg, Bolzano-Bozen, Italy, 2013.

[8] S. G. Jeff McWherter, Professional Mobile Application Development, Wrox, 2012.

[9] K. Yaghmour, Embedded Android, O'Reilly Media, Inc., 2013.

[10] M. Nakamura y M. Gargenta, Learning Android, 2nd Edition, O'Reilly Media, Inc., 2014.

[11] Google inc, «Where's my Gphone?,» 05 11 2007. [En línea]. Available: https://googleblog.blogspot.com/2007/11/wheres-my-gphone.html. [Último acceso: 17 04 2020].

[12] C. Stewart, B. Phillips y K. Marsicano, Android Programming: The Big Nerd Ranch Guide, 4th Edition, Big Nerd Ranch Guides, 2019.

[13] Google Inc, «Android Platform API,» Google, [En línea]. Available: https://developer.android.com/reference/packages. [Último acceso: 01 04 2020].

[14] Medium, «Android Jetpack/ Android X,» 12 07 2018. [En línea]. Available: https://medium.com/ta-tonthongkam/android-jetpack-android-x-36c73abff1c5. [Último acceso: 17 04 2020].

[15] C. H. a. I. Neamtiu, «Automating GUI Testing for Android,» de *Proceedings of the 6th International Workshop on Automation of Software Test*, New York, NY, USA, 2011.

[16] IBM Corporation, «A mobile app development primer,» IBM, Nueva York, 2015.

[17] L. Zhifang, G. Xiaopeng y X. Long, «Adaptive random testing of mobile application,» de *2nd International Conference on Computer Engineering and Technology*, Chengdu, 2010.

[18] S. Vilkomir, K. Marszalkowski, C. Perry y S. Mahendrakar, «Effectiveness of Multi-device Testing Mobile Applications,» de *2nd ACM International Conference on Mobile Software Engineering and Systems*, Florence, Italy, 2015.

[19] The verge, «There are now 2.5 billion active Android devices,» 7 05 2019. [En línea]. Available: https://www.theverge.com/2019/5/7/18528297/google-io-2019-android-devices-play-store-total-number-statistic-keynote. [Último acceso: 20 04 2020].

[20] StatCounter, «Mobile Operating System Market Share Worldwide,» 03 2020. [En línea]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide. [Último acceso: 01 04 2020].

[21] T. Badgett, C. Sandler y G. J. Myers, The Art of Software Testing, 3rd Edition, Wiley, 2011.

[22] F. Viktor y G. Alex, Test-Driven Java Development, Birmingham B3 2PB, UK.: Packt Publishing Ltd., 2015.

[23] K. S. Arif y U. Ali, «Mobile Application testing tools and their challenges:A comparative study,» de *International Conference on Computing, Mathematics and Engineering Technologies*, Boston, 2019.

[24] D. T. Milano y P. Blundell, Learning Android Application Testing, Packt Publishing, 2015.

[25] K. Beck, «CampSamlltalk The mother of all unit testing frameworks,» SourceForge, 2003. [En línea]. Available: http://sunit.sourceforge.net/manual.htm. [Último acceso: 15 04 2020].

[26] robolectric.org, «AndroidX Test,» GitHub, 2017. [En línea]. Available: http://robolectric.org/androidx_test/. [Último acceso: 22 04 2020].

[27] D. Zelenchuk, «Getting Started with Espresso for Android,» de *Android Espresso Revealed: Writing Automated UI Tests*, Apress, 2019.

[28] D. Zelenchuk, Android Espresso Revealed: Writing Automated UI Tests, Apress, 2019.

[29] SmartBear Software, «Cucumber Documentation,» 2019. [En línea]. Available: https://cucumber.io/docs/cucumber/. [Último acceso: 22 04 2020].

[30] Bitbar, «Calabash Tutorial,» 20202. [En línea]. Available: https://bitbar.com/blog/calabash-tutorial-for-mobile-app-testing/. [Último acceso: 11 04 2020].

[31] GitHub Inc., «calabash_steps.rb,» 4 04 2016. [En línea]. Available: https://github.com/calabash/calabash-ios/blob/develop/calabash-cucumber/features/step_definitions/calabash_steps.rb#L1. [Último acceso: 21 06 2020].

[32] M. Hans, Appium Essentials, Packt Publishing, 2015.

[33] JS Foundation, «Automation for App,» 2020. [En línea]. Available: http://appium.io/docs/en/about-appium/appium-clients/. [Último acceso: 01 05 2020].

[34] D. Ghosh, DSLs in Action, Manning Publications, 2010.

[35] L. Ralf, V. Joost y S. Joao, «WebDSL: A Case Study in Domain-Specific Language Engineering,» de *Generative and Transformational Techniques in Software Engineer II*, Braga, Portugal, Springer, 2008, pp. 291-368.

[36] A. Rahien, DSLs in Boo: Domain-Specific Languages in .NET, Manning Publications, 2010.

[37] Eclipse Foundation, Inc, «LANGUAGE ENGINEERING FOR EVERYONE!,» [En línea]. Available: https://www.eclipse.org/Xtext/index.html. [Último acceso: 01 05 2020].

[38] mgoraldev@gmail.com, *Ingresos vs Gastos - finanzas bajo control,* [Aplicación Móvil], 2020.

[39] Agile Alliance, «Agile Alliance,» Agile Alliance, 2020. [En línea]. Available: https://www.agilealliance.org/agile101/. [Último acceso: 01 04 2020].

[40] I. Lake y R. Meier, Professional Android, 4th Edition, Wrox, 2018.

[41] M. Voelter, B. Sebastian, D. Christian, E. Birgit, H. Mats, K. Lennart, V. Eelco y W. Guido, Designing, Implementing and Using Domain-Specific Languages, http://dslbook.org, 2013.

[42] behat community, «Writing features - Gherkin language,» 2016. [En línea]. Available: https://docs.behat.org/en/latest/user_guide/gherkin.html. [Último acceso: 21 02 2020].

[43] DAN NORTH & ASSOCIATES, «introduction to BDD,» 03 2006. [En línea]. Available: https://dannorth.net/introducing-bdd/#translations. [Último acceso: 07 05 2020].

7 Apéndice

7.1 Lista de símbolos y abreviaciones

UI	Interfaz de usuario
SDK	Kit de desarrollo de software
TDD	Desarrollo orientado a pruebas
BDD	Desarrollo orientado a comportamiento
DSL	Lenguaje específico de dominio
IDE	Entorno de desarrollo integrado
Log4J2	Librearía Apache Common usada para realizar trazabilidad de código fuente Java.
EMF	Marco de Trabajo de Modelamiento sobre Eclipse
aatDSL	Android Application Testing DSL

7.2 Gramática DSL

```
grammar edu.uned.iss.tfm.AatDSL with org.eclipse.xtext.common.Terminals

generate aatDSL "http://www.uned.edu/iss/tfm/AatDSL"

Model:
	feature+=Feature
;

// Each Feature, has one or more Scenarios
Feature:
	'Feature:' description=STRING
	scenarios += Scenario*
;

// Each Scenario, has Given, When, Then
Scenario:
	'Scenario:' sequence=INT scenario=STRING
		('Given:' given=GivenStatements)?
		'When:' when=ActionsStatements
		'Then:' then=ThenStatements
;
```

```
//
ActivityStarted:
	"Activity_to_check" name=ANDROID_ID
;

GivenStatements:
	activity=ActivityStarted?
	validations+=ConditionalStatement+
;

//
ActionsStatements:
	statements+=Statement+;


//
ThenStatements:
	validations+=ConditionalStatement+;


/**********************
 * Action/Event Statements *
 **********************/
Statement:
	InputText |
	ButtonPress |
	OptionChoose |
	ValueSelect |
	{Statement} COMMENT
;

InputText:
	('I')? ('type'|'input') value=VALUE  'into' name=ANDROID_ID
;
ButtonPress:
	('I')? ('tap'|'press'|'click') 'over' name=ANDROID_ID
;
OptionChoose:
	('I')? 'choose' value=VALUE
;
ValueSelect:
	('I')? 'select' value=VALUE 'in' name=ANDROID_ID
;

/**********************
 * Validation Statements *
 **********************/
ConditionalStatement:
	IsEnabled |
	IsVisible |
    OptionIsChecked |
```

```
    ValueIsSelected |
    MessageIsDisplayed |
    TextContains |
     {ConditionalStatement} COMMENT
;


IsEnabled:
      op=('And'|'But')? name=ANDROID_ID 'is' 'enabled'
;
IsVisible:
      op=('And'|'But')? name=ANDROID_ID 'is' 'visible'
;
OptionIsChecked:
      op=('And'|'But')? 'Option' value=VALUE 'is'? 'checked'
;
ValueIsSelected:
      op=('And'|'But')? 'Value' value=VALUE 'is'? 'selected' 'at'?
name=ANDROID_ID
;
MessageIsDisplayed:
      op=('And'|'But')? 'Message' value=VALUE 'is'? 'showed'
;
TextContains:
      op=('And'|'But')? 'Content' name=ANDROID_ID expression=BooleanEvaluation
value=VALUE
;


//Boolean expression, to check Content
enum BooleanEvaluation:
    StartsWith = 'startsWith' |
    EndsWith = 'endsWith' |
    Contains = 'contains'  |
    Equals = 'equals'  |
    GreaterThan = 'greaterThan' |
    GreaterEqualsThan = 'greaterEqualsThan' |
    LessThan = 'lessThan' |
    LessEqualsThan = 'lessEqualsThan'
;

//Comment
terminal COMMENT : '#' !('\n'|'\r')* ('\r'? '\n')?;

//Android identifier
ANDROID_ID:
      ID(("."|"á"|"é"|"í"|"ó"|"ú"|"ñ") ID)*
;

VALUE:
      '\\'STRING
;
```

7.3 Generador Xtend

```
/*
 * generated by Xtext 2.21.0
 */
package edu.uned.iss.tfm.generator

import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.AbstractGenerator
import org.eclipse.xtext.generator.IFileSystemAccess2
import org.eclipse.xtext.generator.IGeneratorContext
import edu.uned.iss.tfm.aatDSL.Scenario
import edu.uned.iss.tfm.aatDSL.OptionIsChecked
import edu.uned.iss.tfm.aatDSL.IsEnabled
import edu.uned.iss.tfm.aatDSL.ValueIsSelected
import edu.uned.iss.tfm.aatDSL.MessageIsDisplayed
import edu.uned.iss.tfm.aatDSL.TextContains
import edu.uned.iss.tfm.aatDSL.InputText
import edu.uned.iss.tfm.aatDSL.ButtonPress
import edu.uned.iss.tfm.aatDSL.OptionChoose
import edu.uned.iss.tfm.aatDSL.ValueSelect
import java.util.Date
import edu.uned.iss.tfm.aatDSL.IsVisible

/**
 * Generates code from your model files on save.
 *
 * See
https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#code-
generation
 */
class AatDSLGenerator extends AbstractGenerator {

  final String PKG = "edu.uned.missi.tfm.AppiumApi"

  override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
IGeneratorContext context) {
    val fileName =
resource.URI.lastSegment.toString.replaceFirst("\\.feature","")
    val path = PKG.replace('.','\\')
    for (e : resource.allContents.toIterable.filter(Scenario)) {
      fsa.generateFile(
        path + "\\" + fileName + "_" + e.sequence + ".java",
        e.convertJava(fileName)
      )
    }

  }

  // Get Java Class
  def CharSequence convertJava(Scenario scenario, String fileName){


    '''
package «PKG»;

/*
```

```
 No: «scenario.sequence»
 Description:
 «scenario.scenario»
 «new Date()»
*/

import edu.uned.missi.tfm.appiumlib.model.Expression;
import edu.uned.missi.tfm.test.impl.Scenario;

public class «fileName + "_" + scenario.sequence» extends Scenario{

//
//Given
//
public void given(){
  «IF scenario.given !== null»
    «IF scenario.given.activity !== null»
      startActivity(«scenario.given.activity.name»);
    «ENDIF»
    «IF scenario.given.validations !== null»
      «FOR v : scenario.given.validations»
        «IF v instanceof IsEnabled»
              «isEnabledSTMT((v as IsEnabled))»
        «ENDIF»
        «IF v instanceof IsVisible»
              «isVisibleSTMT(v as IsVisible)»
        «ENDIF»
        «IF v instanceof OptionIsChecked»
              «isOptionIsCheckedSTMT((v as OptionIsChecked))»
        «ENDIF»
        «IF v instanceof ValueIsSelected»
              «ValueIsSelectedSTMT((v as ValueIsSelected))»
        «ENDIF»
        «IF v instanceof MessageIsDisplayed»
              «MessageIsDisplayedSTMT((v as MessageIsDisplayed))»
        «ENDIF»
        «IF v instanceof TextContains»
               «TextContainsSTMT((v as TextContains))»
        «ENDIF»
      «ENDFOR»

    «ENDIF»
  «ENDIF»
}

//
// When
//
public void when(){
  «IF scenario.when !== null»
    «IF scenario.when.statements !== null»
      «FOR s : scenario.when.statements»
        «IF s instanceof InputText»
              «inputTextSTMT((s as InputText))»
        «ENDIF»
        «IF s instanceof ButtonPress»
              «buttonPressSTMT((s as ButtonPress))»
        «ENDIF»
        «IF s instanceof OptionChoose»
              «optionChooseSTMT((s as OptionChoose))»
```

```
        «ENDIF»
        «IF s instanceof ValueSelect»
              «valueSelectSTMT((s as ValueSelect))»
        «ENDIF»
      «ENDFOR»

    «ENDIF»
  «ENDIF»
}

//
// Then
//
public void then(){
  «IF scenario.then !== null»
    «IF scenario.then.validations !== null»
      «FOR v : scenario.then.validations»
        «IF v instanceof IsEnabled»
              «isEnabledSTMT((v as IsEnabled))»
        «ENDIF»
        «IF v instanceof IsVisible»
              «isVisibleSTMT(v as IsVisible)»
        «ENDIF»
        «IF v instanceof OptionIsChecked»
              «isOptionIsCheckedSTMT((v as OptionIsChecked))»
        «ENDIF»
        «IF v instanceof ValueIsSelected»
              «ValueIsSelectedSTMT((v as ValueIsSelected))»
        «ENDIF»
        «IF v instanceof MessageIsDisplayed»
              «MessageIsDisplayedSTMT((v as MessageIsDisplayed))»
        «ENDIF»
        «IF v instanceof TextContains»
              «TextContainsSTMT((v as TextContains))»
        «ENDIF»
      «ENDFOR»

    «ENDIF»
  «ENDIF»
}

}
  '''
}


/* GIVEN */
def ValueIsSelectedSTMT(ValueIsSelected component) {
  '''
  «IF component.op == "But"»
  valueNotSelected("«component.name»",«convertValue(component.value)»);
  «ELSE»
  valueSelected("«component.name»",«convertValue(component.value)»);
  «ENDIF»
  '''
}

def isOptionIsCheckedSTMT(OptionIsChecked component) {
  '''
  «IF component.op == "But"»
```

```
  isNotChecked(«convertValue(component.value)»);
  «ELSE»
  isChecked(«convertValue(component.value)»);
  «ENDIF»
  '''
}

def isEnabledSTMT(IsEnabled component) {
  '''
  «IF component.op == "But"»
  isNotEnabled("«component.name»");
  «ELSE»
  isEnabled("«component.name»");
  «ENDIF»
  '''
}

def isVisibleSTMT(IsVisible component) {
  '''
  «IF component.op == "But"»
  isNotVisible("«component.name»");
  «ELSE»
  isVisible("«component.name»");
  «ENDIF»
  '''
}

def MessageIsDisplayedSTMT(MessageIsDisplayed component) {
  '''
  «IF component.op == "But"»
  isMessageNotDisplayed(«convertValue(component.value)»);
  «ELSE»
  isMessageDisplayed(«convertValue(component.value)»);
  «ENDIF»
  '''
}

def TextContainsSTMT(TextContains component) {
  '''
  «IF component.op == "But"»

notCompare("«component.name»","Expression.«component.expression»,«convertValue(
component.value)»);
  «ELSE»

compare("«component.name»",Expression.«component.expression»,«convertValue(comp
onent.value)»);
  «ENDIF»
  '''
}

/* WHEN */

def valueSelectSTMT(ValueSelect component) {
  '''
  selectValue("«component.name»",«convertValue(component.value)»);
  '''
}

def optionChooseSTMT(OptionChoose component) {
```

```
  '''
  chooseOption(«convertValue(component.value)»);
  '''
}

def buttonPressSTMT(ButtonPress component) {
  '''
  pressButton("«component.name»");
  '''
}

def inputTextSTMT(InputText component) {
  '''
  type("«component.name»",«convertValue(component.value)»);
  '''
}

// Utility

def String convertValue(String value) {
  return value.replace("\\","");
}

}
```