UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Desarrollo de un Framework de Automatización de Pruebas de Aplicación Móvil

INFORME DE ACTIVIDADES PROFESIONALES

Que para obtener el título de

Ingeniero en Computación

P R E S E N T A

Andrés Uriel Chavira Tapia

ASESORA DE INFORME

M.I. Tanya Itzel Arteaga Ricci

Ciudad Universitaria, Cd. Mx., 2022

Índice

Introducción 4

Objetivo 4

Capítulo 1. Antecedentes 5

1.1 Ciclo de vida de desarrollo de software 5
1.2 Pruebas de software 6
1.2.1 Pruebas estáticas 7
1.2.2 Pruebas funcionales 7
1.2.2.1 Pruebas de caja blanca 8
1.2.2.2 Pruebas de caja negra 8
1.2.3 Pruebas de rendimiento 10
1.3 Prácticas de pruebas de software 11
1.3.1 Pruebas de regresión 11
1.3.2 Pruebas smoke 11
1.3.3 Pruebas sanity 12
1.4 Entornos de pruebas 12
1.5 Automatización de pruebas basadas en el protocolo WebDriver 13
1.5.1 Protocolo WebDriver 13
1.5.2 Automatización en móviles 14
1.5.3 Robot Framework 15
1.6 DevOps 16
1.6.1 CI/CD 17
1.7 Sobre la importancia de realizar pruebas automatizadas 18

Capítulo 2. Contexto de la participación profesional 19

Capítulo 3. Contexto del proyecto 20

3.1 Definición del problema 20
3.2 Aplicación por automatizar 21

Capítulo 4. Análisis y metodología empleada 23

Capítulo 5. Participación profesional 24

5.1 Propuesta inicial 24
5.2 Desarrollo del proyecto 25
5.2.1 Fase 1 25
5.2.1.1 Análisis 25
5.2.1.2 Diseño 26
5.2.1.3 Entrega 26
5.2.2 Fase 2 26
5.2.2.1 Análisis 26
5.2.2.2 Diseño 27
5.2.2.3 Entrega 27
5.2.3 Fase 3 27

5.2.3.1 Análisis 27
5.2.3.2 Diseño 28
5.2.3.3 Entrega 29
5.2.4 Fase 4 29
5.2.4.1 Análisis 29
5.2.4.2 Diseño 29
5.2.4.3 Entrega 30
5.2.5 Fase 4.1 30
5.2.5.1 Análisis 30
5.2.5.2 Diseño 31
4.3.5.3 Entrega 31
5.2.6 Fase 4.2 31
5.2.6.1 Análisis 31
5.2.6.2 Diseño 31
5.2.6.3 Entrega 32
5.2.7 Fase 5 33
5.2.7.1 Análisis 33
5.2.7.2 Diseño 34
5.2.7.3 Entrega 34
5.2.8 Fase 6 35
5.2.8.1 Análisis 35
5.2.8.2 Diseño 35
5.2.8.3 Entrega 35
Capítulo 6. Resultados 37
6.1 Framework de automatización de aplicación móvil 37
6.2 Herramienta interna de depuración 38
6.3 Documentación y sesiones de entrenamiento 38
Capítulo 7. Conclusiones 39
Bibliografía 40

Introducción

La tecnología se abre paso poco a poco en la vida de las personas, desde ayudarlas a reducir tiempos de trabajo o espera, enviar u obtener información al instante, organizar y administrar tareas compartidas, entre otras tareas, hasta automatizar procesos y, en general, a realizar cualquier acción que pueda ser programada y puesta en un computador. Esto no sólo trae consigo una aceptación y gran interés de estos avances en la integración en ambientes domésticos o de industria, sino también una expectativa creciente de procesos cada vez más rápidos, eficientes y, a menudo, automáticos.

Hoy en día existen un sinfín de servicios computacionales que emplean algún tipo de proceso automático, y es, sin duda alguna, una de las direcciones hacia donde se dirigen los esfuerzos alrededor del desarrollo de software; estas incluyen el análisis de la calidad del código, la compilación automática, hasta la ejecución de pruebas automatizadas.

Dado que la demanda del desarrollo de software crece año con año, así también lo hace la necesidad de contar con elementos que garanticen la calidad de dicho software. En este sentido, las pruebas de software, generalmente ejecutadas de manera manual, son parte esencial de este proceso, y su respectiva automatización cobra importancia cuando se contempla que una sola prueba debe ejecutarse en distintos dispositivos y ambientes bajo distintas condiciones, y después de cada actualización, por lo que resulta ser un proceso largo y costoso para las empresas, que principalmente consiste en la repetición manual de tareas a nivel de usuario. Es por ello que la automatización de dichos procesos resulta de gran atractivo en la industria en la actualidad.

Objetivo

Desarrollar un sistema escalable y de fácil mantenimiento, que permita la ejecución automatizada de distintos casos de prueba de validación de un producto en un dispositivo móvil al simular la entrada a nivel de usuario. Dicho sistema debe basar su estructura en un acercamiento modular, con el fin de que las funciones que en este se implementen puedan ser reutilizadas bajo diversos escenarios. A su vez, debe permitir identificar fallas en dichas ejecuciones en tanto el resultado encontrado no sea el esperado, con la intención de disminuir la carga de las pruebas manuales, así como para diversificar los entornos en los que se realizan dichas pruebas, ya sean ambientes internos del servicio, versiones de sistema operativo o dispositivos propiamente, reduciendo igualmente el error humano. Por último, debe ser capaz de generar datos históricos del tiempo que toma realizar ciertos procedimientos en el sistema para poder investigar y actuar sobre posibles problemas de rendimiento entre versiones y ambientes.

Capítulo 1. Antecedentes

1.1 Ciclo de vida de desarrollo de software

A medida que ocurren avances tecnológicos, las expectativas generadas hacia los productos de software crecen a un ritmo similar, entre las cuales están las de concebir sistemas robustos, de rápida respuesta, libres de fallas, con actualizaciones continuas y alta disponibilidad, por mencionar algunas, desde el día del lanzamiento. Esto ha motivado la evolución de los acercamientos que se tienen con respecto al desarrollo de sistemas, y con ello, notables mejoras en las maneras en las que el Ciclo de Vida del Desarrollo de Software (CVDS) es planeado y ejecutado, llevando a generar cambios incluso alrededor de dicho proceso. Estos esfuerzos se llevan a cabo para poder satisfacer estas y otras expectativas que, con el paso del tiempo, se convierten en necesidades del cliente, así como la demanda derivada del crecimiento natural en la complejidad de tales proyectos.

Aunado a lo anterior, el desarrollo de un sistema de información requiere del compromiso de tiempo y recursos de una compañía. Sin embargo, este no es un problema reciente, sino uno para el que progresivamente se encuentran mejores maneras de trabajar. Existen así secuencias de actividades diseñadas para guiar a los desarrolladores por un camino que provee de software de calidad, a tiempo y dentro del presupuesto establecido, lo que también se conoce como una metodología, y de donde primeramente nace el CVDS clásico, conocido igualmente como desarrollo en cascada o metodología de cascada.

Las etapas esenciales de este proceso consistían en planeación, análisis, diseño, e implementación. La planeación trataba sobre establecer los alcances del nuevo sistema al definir el problema por resolver. El análisis buscaba entender el sistema actual al conducir un estudio con tal finalidad. El diseño consistía en definir los procesos y los datos que se usarían en el nuevo sistema. Por último, la implementación partía de la preparación del software, pasando por la creación de los archivos de datos y la construcción del hardware para llegar a adoptar el nuevo sistema, como se ilustra en la figura 1.1 [1].

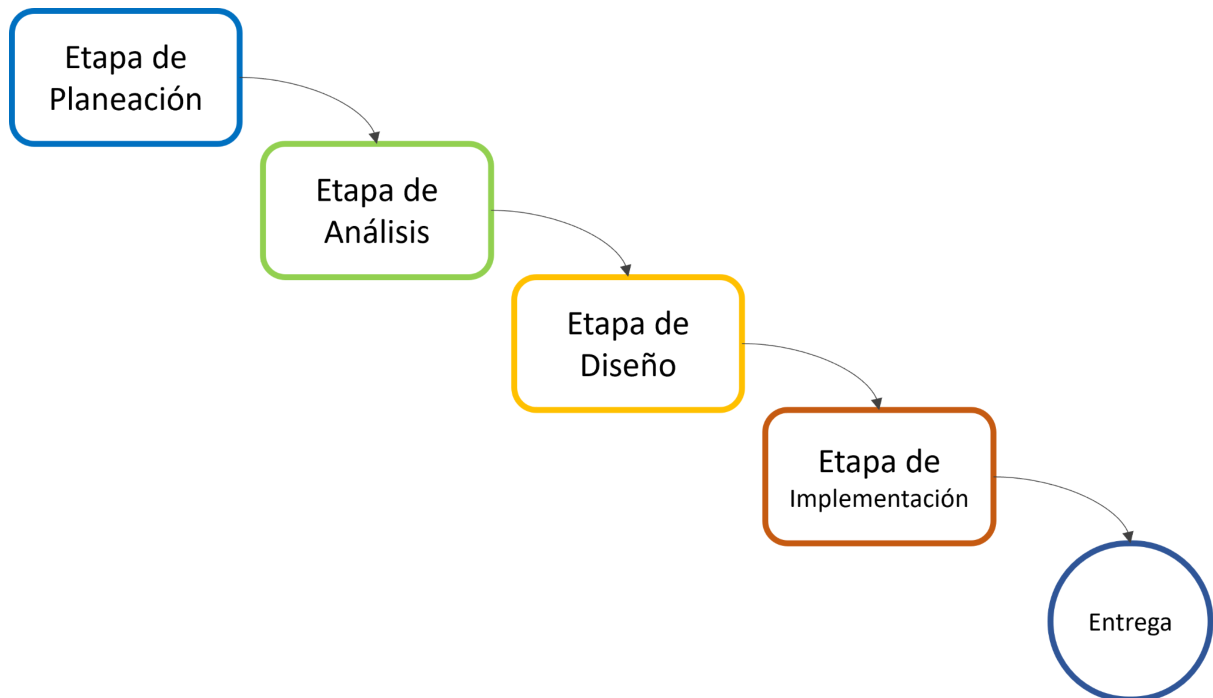

Figura 1.1 Ciclo de vida del desarrollo en cascada de Everett[1]

Dicha metodología asumía y confiaba en saber todos los detalles del proyecto desde la etapa de planeación, lo que indudablemente llevó a la concepción de una metodología que no sólo veía los cambios inesperados a mitad de alguna de las etapas subsecuentes como una excepción, sino como una regla, dada el nulo manejo que tenía esta metodología para estos casos.

En este sentido, las metodologías evolucionaron al punto en el que los cambios se esperaban comúnmente, y en muchos casos el desarrollo se fue fraccionando en desarrollos pequeños, iterativos e incrementales, los cuales se enfocan en proveer valor a los clientes de manera constante, obteniendo retroalimentación de este para cada una de estas entregas. De esta manera, las planeaciones a detalle pasaron a acortar sus plazos, reconociendo así que los cambios y complicaciones ocurrirían de una manera u otra en algún momento, dando paso a las metodologías ágiles.

1.2 Pruebas de software

Dado que estos cambios condujeron el desarrollo hacia un marco que contempla un análisis de resultados para cada una de las entregas, fue que las pruebas de software tomaron mayor relevancia, tanto para garantizar calidad en el producto entregado, como para evitar la propagación de defectos en cada una de estas iteraciones.

[1] Traducida del Inglés y recuperada de [1] p.14.

La importancia de las pruebas de software yace entonces en este proceso, puesto que ocupa un papel esencial en la aprobación del desarrollo mismo, y no sólo cumple su cometido, sino que evita costos y retrasos que, en una detección temprana de defectos, se solucionan sin mayor agravio. Según una encuesta realizada por el gobierno estadounidense en el año 2002, los errores de software le cuestan a su economía un total de 59.5 miles de millones de dólares por año, mientras que alrededor de 22.2 miles de millones de dólares en pérdidas podrían eliminarse con pruebas de software implementadas en todas las fases del desarrollo de software [4].

En la actualidad, existen equipos de aseguramiento de la calidad (QA, por sus siglas en Inglés) y son ellos quienes se encargan de promover distintas prácticas en el ciclo de desarrollo de software, sobre las cuales trabajan de manera activa, y extienden tal cultura hacia los demás equipos. Su principal tarea es la de asegurar que se cumplan los requisitos de calidad acordados en una organización [3].

Las pruebas, en este sentido, son esenciales en un proyecto, dado que evitan posibles futuros retrasos al ser atendidos a tiempo, lo que resulta en la prevención de costos mayores.

Existen diversas denominaciones en tipos de pruebas, según sea la literatura que se consulta, el fin de estas, sus medios, o sus alcances, entre otros factores. Se introducen así cuatro tipos generales de pruebas.

1.2.1 Pruebas estáticas

A pesar de que estas se ven involucradas en el desarrollo de un proyecto de software, estas pruebas no enfocan sus esfuerzos en el código del producto, sino en la reducción de defectos en su documentación, sobre la cual se desarrolla el producto. El desarrollo de software comienza, da paso y termina en la documentación [1]. Estas pruebas se realizan mediante pruebas de escritorio, inspecciones, tutoriales y presentaciones para garantizar lo completo y lo correcto de un documento.

1.2.2 Pruebas funcionales

El objetivo de estas pruebas es el de validar que el comportamiento del software corresponde al que fue documentado según la funcionalidad del negocio en los requerimientos y especificaciones del software [1].

Las pruebas de caja negra como las pruebas de caja blanca se consideran a su vez pruebas funcionales.

1.2.2.1 Pruebas de caja blanca

Dada la naturaleza del software, un producto llevado a producción no revela ni lleva consigo el código fuente con el que fue construido, sino una versión compilada o, si acaso, ofuscada de este, por lo que funciona como una caja negra.

Los desarrolladores, por su parte, tienen acceso a dicho código fuente y, en su caso, pueden realizar pruebas sobre cada línea de su código y validar que este funciona de la manera esperada, así como de que no “se rompe”. La combinación de estos dos acercamientos convierte este tipo de pruebas en el más efectivo en la búsqueda de defectos.

Su principal objetivo es el de verificar que las declaraciones, condiciones, bucles, rutas de código y flujos de datos sean correctos [1].

1.2.2.2 Pruebas de caja negra

El presente informe trabaja sobre este tipo de pruebas. Estas, como se describe en las pruebas de caja blanca, no revelan la lógica interna del producto, sino que funcionan como un sistema que únicamente recibe una o varias entradas y devuelve uno o varios resultados. Su objeto es el de verificar lo correcto del comportamiento del software que da soporte directo a la actividad diaria del negocio.

Para realizar estas pruebas, se requiere conocer el resultado esperado basado en una serie de entradas, así como tener la certeza de que dichas entradas fueron provistas de manera correcta. Si el resultado obtenido no coincide con el esperado, y la prueba se realizó de la manera especificada, entonces existe un error de implementación o un error de especificación, de manera respectiva para cada condición.

Un ejemplo de un caso de prueba se presenta en la tabla 1.3, la cual toma como base el caso de uso especificado en la tabla 1.2. En este se observan una serie de pasos para los cuales existe un resultado esperado en respuesta. Así, por ejemplo, para el paso número seis, tras completar la compra del “artículo azul”, el cliente recibe una confirmación de su compra, y la prueba puede continuar al paso número siete, en el supuesto de que todos los pasos anteriores cumplieron con sus respectivos resultados esperados.

Tabla 1.2 Ejemplo de un caso de uso para aplicación de compras en línea *(happy path)*.[2]

Actor	**Acción**	**Descripción**
Cliente	Iniciar sesión	Una pantalla de inicio configura la aplicación, cuenta el número de veces que el usuario abre la aplicación, y lleva al usuario al menú principal.
Cliente	Explorar catálogo	El cliente es capaz de explorar el catálogo de productos por categoría de producto, fabricante, y número de catálogo. Los productos de interés de la búsqueda pueden ser seleccionados para más información.
Cliente	Explorar detalle del producto	El cliente puede mostrar la descripción del producto, una imagen del producto, precio, y cantidad en stock de cada uno de los productos seleccionados.
Cliente	Actualizar el carrito de compras	El cliente puede agregar, quitar, o actualizar los productos a comprar del carrito de compras.
Cliente	Comprar los productos del carrito de compras	El cliente puede revisar los productos a ser comprados, indicar la información de envío de cliente, y proporcionar la información de pago del cliente.
Cliente	Completado de la compra	Se valida la capacidad del cliente para pagar por los productos, provee de una orden de compra al cliente, e inicia la entrega del producto.
Cliente	Cerrar sesión	Se valida el completado de todas las acciones del cliente durante la sesión antes de realizar la desconexión con este.

Tabla 1.3 Caso de prueba de una ruta de negocio completa de un caso de uso.[3]

Núm. de paso	**Paso**	**Resultado esperado**
1.	Iniciar sesión	Accede al menú principal
2.	Explorar catálogo	Encuentra un artículo azul
3.	Explorar detalle del producto	Muestra descripción del artículo azul
4.	Actualizar carrito de compras	Agrega el artículo azul

[2] Traducida del Inglés y recuperada de [1] p.101.
[3] Traducida del Inglés y recuperada de [1] p.101.

5.	Comprar los productos del carrito de compras	Compra el artículo azul
6.	Completado de la compra	Obtiene la confirmación de compra
7.	Cerrar sesión	Sale de la aplicación con éxito

1.2.3 Pruebas de rendimiento

Estas pruebas se realizan una vez que el software ha demostrado que funciona de manera correcta. Es posible que estas revelen defectos funcionales, mas no es su objetivo. Su principal enfoque es el de conocer los tiempos de respuesta y el rendimiento de un producto bajo ciertas condiciones; en otras palabras, el de validar que la “velocidad” del software cumple con la “velocidad” que se documentó en los requisitos del producto [1].

En la tabla 1.4 se observan distintos tipos de acciones en un sistema web, para los cuales se especifican distintos tipos de respuesta máximos requeridos para un flujo de usuarios concurrentes esperados, de acuerdo con los horarios en los que estos casos fueron registrados o previstos.

Tabla 1.4 Borrador de un plan de rendimiento de carga de trabajo.[4]

Grupo de transacciones	**Requerimiento de tiempo de respuesta**	**Pico de usuarios o clientes activos**	**Día y hora de pico de actividad**
Navegación del menú	Máx. 3 segundos	2,000	Lu-Vi 12-1 P.M.
Inicio/Cierre de sesión	Máx. 3 segundos	2,000	Lu-Vi 12-1 P.M.
Visualización de detalles de producto	Máx. 4 segundos	2,000	Lu-Vi 12-1 P.M.
Pasos de compra	Máx. 7 segundos	500	Sá 9-11 A.M.
Búsqueda en catálogo	Máx. 10 segundos	2,000	Lu-Vi 12-1 P.M.
Pago con tarjeta de crédito	Máx. 30 segundos	500	Sá 9-11 A.M.

[4] Traducida del Inglés y recuperada de [1] p.132.

1.3 Prácticas de pruebas de software

Existen diversos acercamientos para llevar a cabo pruebas de software. Para el propósito del presente informe, se presentan tres de ellos, aquellos cuyos papeles toman relevancia en proyectos donde frecuentemente se producen entregables, producto de metodologías ágiles de desarrollo.

La definición de estas prácticas no es siempre un reflejo exacto de las prácticas que se llevan a cabo en la industria, en el sentido de que el tipo o número de pruebas puede variar según la necesidad de la organización, el número de sus recursos, su tiempo, o la confiabilidad que tiene su software. Estas prácticas se adaptan de maneras distintas a distintos proyectos y organizaciones.

1.3.1 Pruebas de regresión

En esta práctica, se buscan correcciones o nuevas funciones de software que hacen menos estable a la versión actual al afectar otros segmentos de código que no están relacionados directamente a dichos cambios nuevos [1].

Para llevarlo a cabo, se ejecutan nuevamente todos los casos de prueba que anteriormente habían aprobado, con el fin de garantizar que ninguna funcionalidad anterior se ha “roto” con las nuevas adiciones de código. En caso de que las pruebas de regresión encuentren defectos, el estudio de estos se centra alrededor de las modificaciones recientes hechas en el código.

1.3.2 Pruebas smoke

Habiendo obtenido su nombre de las pruebas de hardware en las que al alimentar un nuevo equipo de hardware se buscaba que físicamente no hubiera humo, las pruebas smoke toman de estas el principio de que no son pruebas exhaustivas, sino una manera de validar que algo funciona hasta cierto punto.

En software, esta práctica se utiliza para validar que las funciones principales de un sistema funcionan de manera adecuada después de haber hecho cambios o incrementos en el código del producto [5]. De esta manera, será menor la cantidad de casos de pruebas o configuraciones que se ejecuten en comparación con las pruebas de regresión.

1.3.3 Pruebas sanity

A menudo confundidas con las pruebas smoke, las pruebas sanity también realizan un menor número de pruebas que las pruebas de regresión. Su objetivo es el de validar que las funciones y correcciones nuevas operen adecuadamente después de haber actualizado el producto [5].

1.4 Entornos de pruebas

Un entorno de pruebas le permite al *tester* tener un espacio de trabajo donde puede obtener resultados de ejecución como los que el usuario final verá en producción antes de que el software sea lanzado a producción. No deben confundirse con las simulaciones o los *benchmarks*, dado que no proveen una experiencia completa en lo que al producto final se refiere.

Este acercamiento se basa en crear un entorno de cómputo similar al de producción donde se ha de probar el nuevo software. Su objetivo es conseguir que el producto a probar devuelva un comportamiento real de producción mientras es analizado fuera del entorno de producción.

A menudo se recomienda contar con un mínimo de tres entornos: uno de desarrollo, para probar inmediatamente las nuevas funcionalidades que se desarrollan a medida que se codifican; uno de pruebas, utilizado específicamente para probar el trabajo realizado previamente en el entorno de desarrollo; y uno de producción, en el cual sólo deben realizarse cambios durante nuevos lanzamientos para proveer una experiencia consistente al usuario final [6]. Así, el ciclo de lanzamiento lleva esta secuencia, desarrollo, pruebas y producción, como se representa en la figura 1.5.

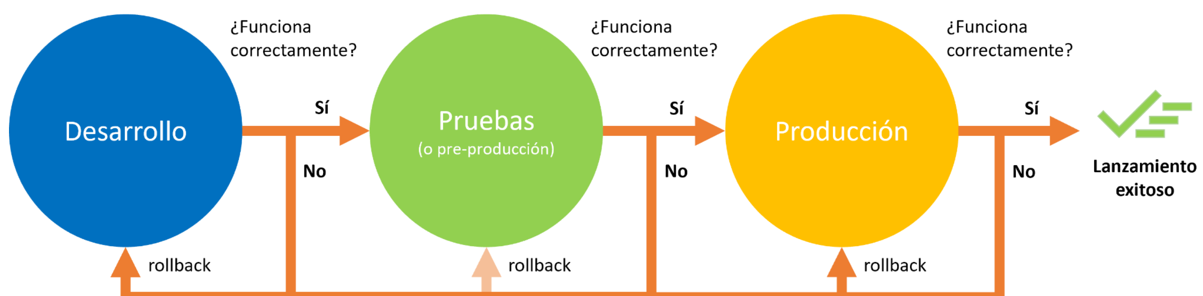

Figura 1.5 Ciclo de lanzamiento general con tres entornos.

Existen excepciones a esta regla, y la creación de más entornos dependerá de las necesidades, presupuesto y organización de cada empresa, para casos tales como entornos de demostración para ventas, casos especiales, para pruebas de integración, de rendimiento, y de seguridad, entre otras.

1.5 Automatización de pruebas basadas en el protocolo WebDriver

La automatización de pruebas funcionales de caja negra se puede llevar a cabo con herramientas de automatización basadas en el protocolo WebDriver. El uso de estas no se limita al de este tipo de pruebas, sin embargo, este es el enfoque que se le dará al presente informe. Esta práctica se emplea para pruebas de interfaz de usuario (IU), las cuales consisten en ejecutar casos de prueba sobre la capa final del producto que vería el usuario, para validar así, desde un acercamiento de caja negra, que el sistema opera correctamente.

1.5.1 Protocolo WebDriver

Un WebDriver es una interfaz de control remoto que provee de una plataforma y protocolo agnóstica de un lenguaje específico como una manera de permitir que programas ajenos envíen instrucciones que controlan el comportamiento de los navegadores web [7].

En la figura 1.6 se observa el funcionamiento de un WebDriver, donde la biblioteca del cliente (los scripts de automatización de tests) se comunica con un servidor intermedio, el cual traduce sus mensajes en comandos que el navegador es capaz de entender. Esto termina por emular las acciones de un usuario final en el navegador, operando directamente sobre el Document Object Model (DOM).

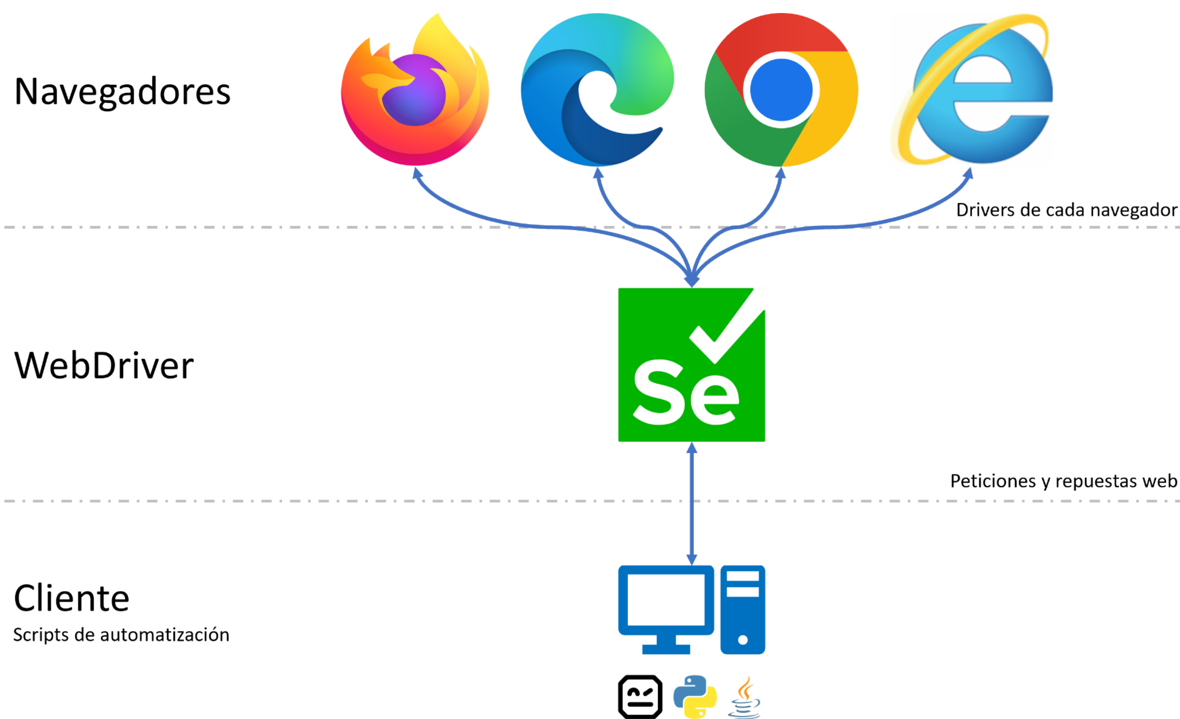

Figura 1.6 Arquitectura del Selenium WebDriver.[5]

1.5.2 Automatización en móviles

Para el caso de los móviles como Android, iOS y Windows Phone, el concepto de WebDriver se extiende de manera similar, siendo la mayor diferencia que el servidor intermedio no puede ser ejecutado en los mismos dispositivos móviles, como lo es posible en una computadora de escritorio [7].

Appium es uno de los servidores web que realizan esta conexión entre un cliente con scripts de automatización y traducen estas instrucciones a comandos que el dispositivo móvil entiende. Este servidor es similar al ChromeDriver de Google Chrome, en el sentido de que es capaz de comunicarse con la biblioteca de automatización de IU para sistemas Android e iOS. De manera interna, Appium sabe cómo comunicarse con la biblioteca UIAutomation de Android, la cual recibe comandos del servidor de Appium y los ejecuta en una aplicación, que eventualmente se traducen en acciones *set*, *get*, *click*, y demás acciones de usuario, como se observa en la figura 1.7. El presente informe se enfocará en su uso, dado que Appium es la solución empleada en desarrollo del mismo, además de tratarse de un proyecto de código abierto.

[5] Basada en la figura recuperada de [7] p.15.

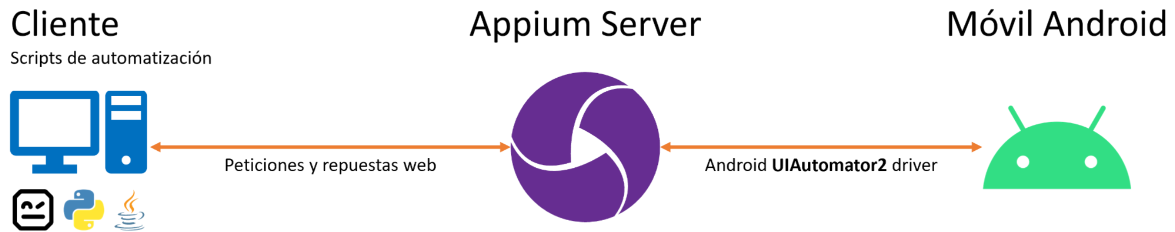

Figura 1.7 Arquitectura del Appium Server con Android.

1.5.3 Robot Framework

Originado en Nokia, Robot Framework (RF) es un framework genérico de automatización de código abierto utilizado para la automatización de pruebas y la automatización robótica de procesos (RPA, por sus siglas en Inglés) [14].

Su implementación está basada en Python, y puede ser integrado con cualquier otra herramienta para crear soluciones flexibles de automatización. Utiliza una sintaxis sencilla, basada en lenguaje natural, y sus funcionalidades pueden ser extendidas con bibliotecas de Python y otros lenguajes de programación.

Appium es sólo una de las bibliotecas que crean una interfaz entre RF y Android. En otras palabras, RF sirve como marco de base para crear y ejecutar la automatización, donde Appium es la herramienta que provee las instrucciones básicas de comunicación con Android.

Dentro de la filosofía de RF se encuentran tres puntos principales, los cuales indican que el "código" debe ser fácil de entender, fácil de mantener, y de rápida ejecución. Elaborando sobre los dos primeros puntos, los archivos creados en RF son fáciles de entender al seguir una convención de lenguaje natural al describir las funciones que se crean, con un nivel de abstracción adecuado, como se observa en la figura 1.8, mientras que para conseguir un sistema de fácil mantenimiento se requiere de una estructuración que permita la escalabilidad y segmentación de sus recursos, sobre la cual se hablará más adelante.

```
*** Test Cases ***
Valid login
    Open browser to login page
    Input user name    demo
    Input password    mode
    Submit credentials
    Welcome page should be open
```

Figura 1.8 Test case de Robot Framework, escrito en lenguaje natural.[6]

[6] Recuperada de [9] p.7.

1.6 DevOps

El término *DevOps* es una mezcla entre *development* (que representa a los desarrolladores de software y al personal de QA) y *operations* (representando a aquellos expertos que llevan el software a producción y administran dicha infraestructura, tales como administradores de bases de datos, de sistemas, y técnicos de redes). DevOps describe prácticas para coordinar el proceso de *delivery* o entrega de software, haciendo especial énfasis en obtener retroalimentación del lado de producción hacia el de desarrollo, y mejorar así el tiempo de cada ciclo; no sólo trata sobre entregar software de manera más rápida, sino también de producir software de calidad más alta [3].

Los aspectos que rigen el funcionamiento de DevOps son: la cultura, las personas por encima de los procesos y las herramientas; la automatización, esencial para obtener retroalimentación rápida; mediciones, la calidad y otros incentivos son críticos; y compartir, creando una cultura donde las personas comparten ideas, procesos y herramientas.

Derivado de su nombre, otro de los objetivos de DevOps es el de concebir acciones de sinergia entre los equipos de desarrollo y de operaciones, dado que tradicionalmente estos departamentos tienen diferencias y conflictos marcados, los cuales actúan como si fueran independientes entre sí (figura 1.9). Los conflictos entre estos dos se resumen en: la necesidad de realizar cambios, apoyada por los desarrolladores que los producen y quieren que se lleven a producción; y el temor al cambio, presentado en el equipo de operaciones, quienes quieren evitar hacer cambios al software para garantizar condiciones estables en los sistemas de producción.

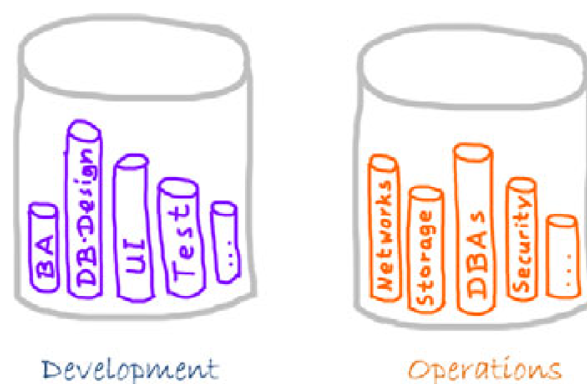

Figura 1.9 Desarrollo y operaciones son dos departamentos distintos. A menudo, estos actúan como silos porque son independientes el uno del otro.[7]

[7] Recuperada de [3] p.6.

Dentro de las prácticas de DevOps, se encuentra un ciclo tal que engloba al desarrollo como la integración del producto, y a las operaciones como la entrega de este, así como su posterior monitoreo y obtención de retroalimentación. Este ciclo se ilustra en la figura 1.10.

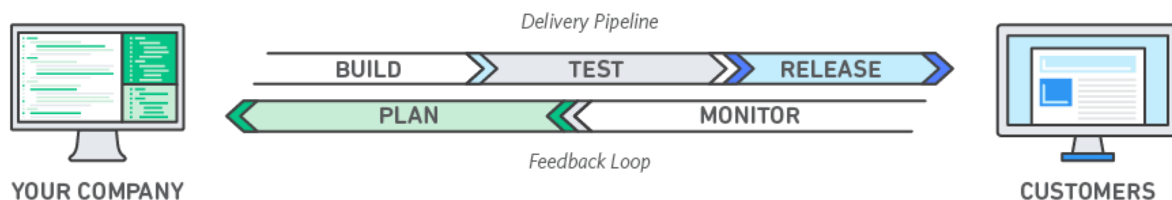

Figura 1.10 Modelo de DevOps.[8]

1.6.1 CI/CD

Como se observó anteriormente, existen dos líneas de acción en el ciclo de DevOps. En particular, la línea de integración y entrega del producto se plantea en dos conceptos, *continuous integration* (CI) y *continuous delivery* (CD). De allí que a estos acrónimos se les nombre juntos como CI/CD. Ambos cuentan con un fuerte fundamento en la automatización del proceso para lograr establecer estas acciones de manera *continua.*

CI hace referencia al ensamblaje de los componentes del producto en etapas incrementales, utilizando una estrategia y procedimientos definidos. En estos entornos, las aplicaciones se construyen y pasan por pruebas (generalmente unitarias y de integración) utilizando herramientas automatizadas cada vez que algún código es agregado al repositorio del proyecto [12].

CD se enfoca en minimizar el tiempo que transcurre entre el desarrollo del producto y el momento en que este es utilizado por los usuarios finales. Este es una extensión de la CI, puesto que de manera automática pone en marcha el nuevo código en un entorno de pruebas (o producción) tras terminar la propia CI [15].

En algunos casos, CD hace referencia a *continuous deployment*, y la diferencia con *continuous delivery* radica en que el proceso propio de *deployment* se encuentre automatizado dentro del ciclo; si está automatizado, se opta por llamarle *continuous deployment* (figura 1.11).

[8] Recuperada de [10].

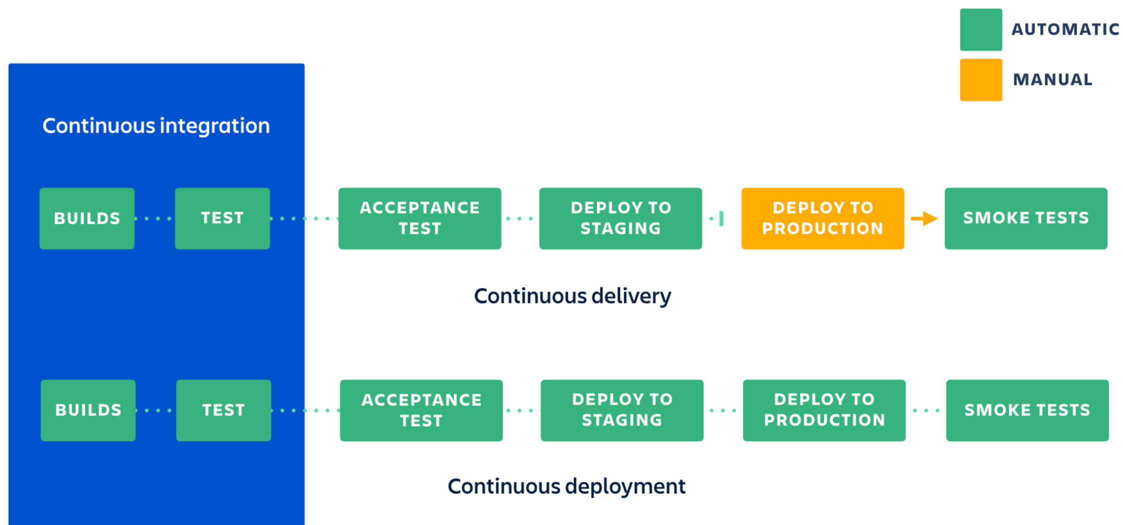

Figura 1.11 Diferencias entre integración continua, entrega continua e implementación continua.[9]

1.7 Sobre la importancia de realizar pruebas automatizadas

Como ya se observó, las pruebas forman parte esencial en los procesos expuestos. Dado que el objetivo de la gran mayoría de los productos que se encuentran en constante desarrollo es el de alcanzar una adopción de ciclos basados en DevOps y CI/CD, resulta relevante concebir procesos automatizados, y es allí donde la automatización de pruebas juega un papel fundamental, tanto en la etapa de CI después de la construcción o compilación del proyecto, así como después del lanzamiento del producto en pruebas smoke, sanity o de regresión, como se observa en la figura 1.11.

Estos ciclos toman importancia cuando se considera que deben realizarse frecuentemente, desde días hasta meses, replicándose igualmente por cada uno de los entornos de pruebas por los que se requiera obtener aprobación, puesto que ahorran tiempo, minimizan esfuerzos, y mejoran el rendimiento en una organización.

[9] Recuperada de [11].

Capítulo 2. Contexto de la participación profesional

El giro de la empresa en la que participé es el del transporte, dentro del ramo de la logística y administración de flotas de vehículos. Su objetivo práctico es el de proveer a sus clientes de un entorno integral que les permita administrar sus vehículos, conductores, y órdenes, entre otros elementos, a través de diversos portales web y servicios de API para el área de administración, y a través de aplicaciones móviles para los usuarios finales (conductores), para guiarlos en sus rutas, entregas, recolecciones, revisiones y otras tareas; esto mientras se cumplen los requerimientos legales de seguridad e impuestos de EE.UU. y Canadá.

Comencé mi carrera profesional al formar parte de esta empresa en agosto del 2020 como *QA Intern* en el área de automatización en el departamento de QA, en donde actualmente me desempeño como un *Associate QA Engineer*. Mi rol como *intern* consistió en el de aprender sobre tecnologías de automatización y conceptos acerca de pruebas para llevar a cabo la creación de un framework de automatización para una de las aplicaciones móviles de la empresa, con el fin de que este pudiera utilizarse como cimiento para la posterior ejecución de pruebas automatizadas, así como para dotar de herramientas a los técnicos de QA para la creación de pruebas automatizadas sobre el producto final. Las responsabilidades para este rol fueron evolucionando a lo largo del periodo que comprendió el proyecto que en este informe se describe, para el cual inicié fungiendo como contribuidor y que tiempo después terminé liderando. Formé parte del equipo asignado al desarrollo del producto sobre el que trabajé en automatizar, mas no me vi involucrado en su metodología de trabajo más allá de ser un espectador con el fin de mantenerme informado sobre nuevos cambios y conocimiento en general, pues llevé la metodología de mi proyecto de manera separada, al tener un objetivo distinto.

A mi llegada al puesto encontré un ambiente que me permitió aumentar mis conocimientos y habilidades interpersonales, así como uno donde aplicar lo que en asignaturas como Ingeniería de Software o Administración de Proyectos de Software aprendí para entender el funcionamiento interno de los proyectos. Por otro lado, la comunicación, la flexibilidad, el sentimiento de trabajo en equipo y la resolución de conflictos, formaron parte de las habilidades que, aunadas a la resolución de problemas, resultaron ser clave para ayudarme a progresar dentro de la empresa y como profesionista. Así, a mi ingreso, contaba yo con estas y otras habilidades, producto del trabajo de los proyectos propios en el curso de la carrera de Ingeniería en Computación, las cuales me permitieron promover e integrarme a un ambiente de comunicación efectiva, donde las propuestas fueron tomadas en cuenta, la ayuda existía en ambos sentidos, y donde se elaboraron acuerdos tomando en cuenta a todos los participantes.

Capítulo 3. Contexto del proyecto

3.1 Definición del problema

En la empresa descrita en el capítulo anterior, requerimos un framework de automatización diseñado para uno de sus productos para móviles. Este debía cubrir los casos básicos de manera modular para permitir crear casos de prueba automatizados más avanzados a partir del armado de estos. Dado que en esta compañía los lanzamientos de nuevas versiones se dan aproximadamente diez veces por año, sin contar aquellas de corrección de errores conocidas como *hotfix*, la automatización de las pruebas para este producto tiene sentido.

Considerando que esta aplicación móvil a su vez trabaja en conjunto con otras aplicaciones móviles, con plataformas web, equipos de telemática y servicios de API, el objetivo general a futuro constaba de crear esta automatización móvil y partir de allí para realizar pruebas en conjunto con estos otros servicios (pruebas de integración), los cuales igualmente se automatizaron en otros proyectos en paralelo para llegar así a tener una automatización integrada. El producto sobre el cual trabajé se denota en la figura 3.1, donde las conexiones entre los servicios se representan en un alto nivel sin considerar detalles de arquitectura.

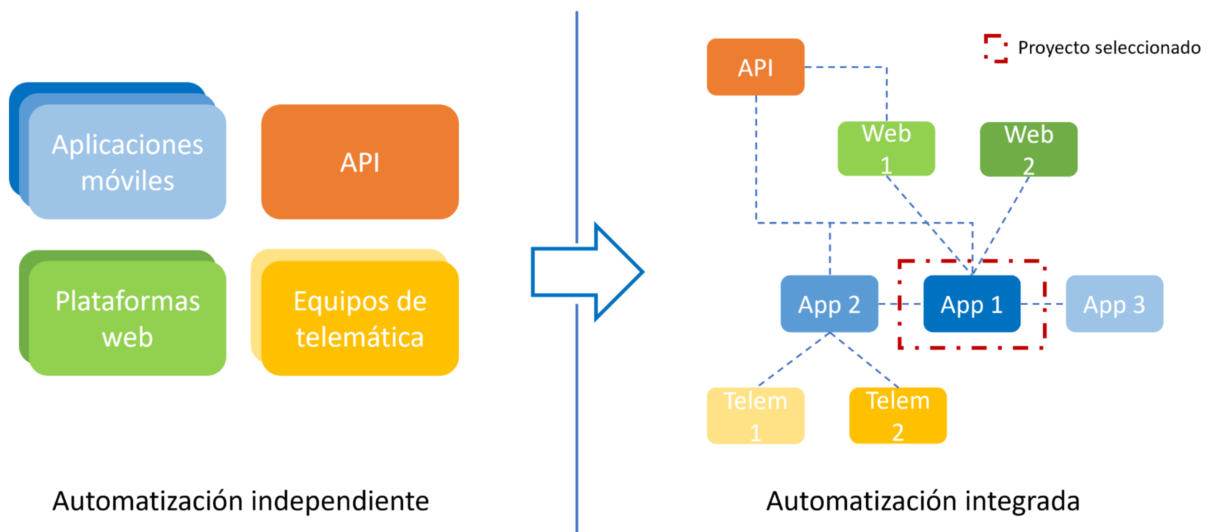

Figura 3.1 Transición de una automatización para pruebas independientes a una que permite la realización de pruebas integrales.

Dado que la participación en mi proyecto se delimitó a trabajar en la automatización de una aplicación móvil, en lo siguiente se tratará esta de manera aislada.

3.2 Aplicación por automatizar

En la aplicación por automatizar el objetivo es el de proveer un sistema integral que permita a los conductores de tales flotas conocer sus rutas del día, desde las paradas que deben realizar, las actividades que deben hacer en cada una de ellas, hasta el cumplimiento de tareas adicionales en relación con normas oficiales requeridas por el gobierno estadounidense y canadiense, así como cualquier otra documentación o tarea requerida particularmente por cada cliente. En este sentido, la aplicación funge como una guía secuencial de las tareas que debe realizar un conductor de carga en su día a día.

Puesto que las tareas que debe realizar una empresa dedicada al transporte de líquidos, como primer ejemplo, no son las mismas que las que lleva a cabo otra empresa dedicada a la mensajería, existe una alta gama de configuraciones en la selección, cantidad, obligatoriedad, entre otros factores, de las tareas que lleva a cabo cada cliente, sin la necesidad de generar aplicaciones distintas para cada caso. De esta manera, se cuenta con una aplicación que tiene múltiples propósitos según sean las preferencias y necesidades operacionales y legales del cliente.

El plan de acción en este contexto lo definimos entonces como el de llevar a cabo la automatización modular de todas las distintas tareas en sus distintas configuraciones, para permitir que posteriormente estas “piezas” sean “armadas” en el orden en el que sea conveniente realizar pruebas, según sea el caso.

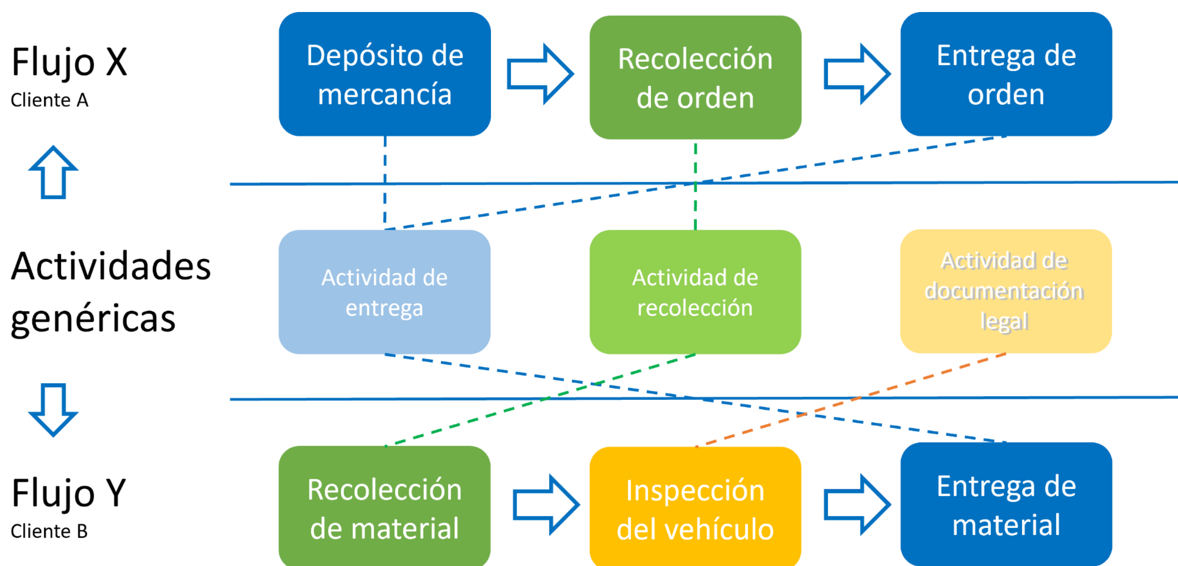

Figura 3.2 Creación de distintos flujos partiendo de la definición de actividades genéricas.

Un ejemplo se ilustra en la figura 3.2, donde se observa una secuencia arbitraria Flujo X que podría utilizar un cliente como su flujo de trabajo; debajo, se observan las distintas tareas de manera genérica y modular; por último, se presentan estas

tareas en un contexto distinto, probando así la reusabilidad de este acercamiento, en distinto orden y con distintos fines.

Aunado a lo ya mencionado, también deben automatizarse acciones tales como el inicio y cierre de sesión, la administración de rutas, las acciones de IU que inician las distintas tareas y sus opciones, así como las validaciones de textos o elementos que se esperen realizar en los distintos contextos.

Capítulo 4. Análisis y metodología empleada

La metodología de desarrollo empleada se fundamentó en Kanban. En nuestro día a día, contábamos con reuniones de *daily standup* para tratar avances, trabajo del día y posibles *blockers*. Dada la naturaleza del proyecto, nuestro "cliente" fue la misma empresa, para quien desarrollamos el framework de automatización y, asimismo, se contaba con un *product backlog* que ya contemplaba tareas que abarcaban todas las funcionalidades de la aplicación, aunadas a aquellas que fueron surgiendo y descubriéndose; de cierta manera, estas tareas estaban desfasadas con el proyecto principal de desarrollo dado que este proyecto de automatización se propuso un par de años después del comienzo del otro.

De manera propia de Kanban, contamos con un *Kanban board* para llevar registro y seguimiento de los avances y las tareas que asigné, y, al no contar con un periodo definido como los *sprints* de Scrum, las entregas y demostraciones las realicé cada vez que cumplimos con un objetivo significativo, a discreción propia o de mi superior.

Como se describió en el capítulo 2, esta metodología se llevó a cabo de manera paralela a aquella utilizada para el proyecto de desarrollo de este producto, dado que, aunque trataran sobre el mismo producto, el objetivo fue distinto. Esta diferencia se ilustra en la figura 4.1.

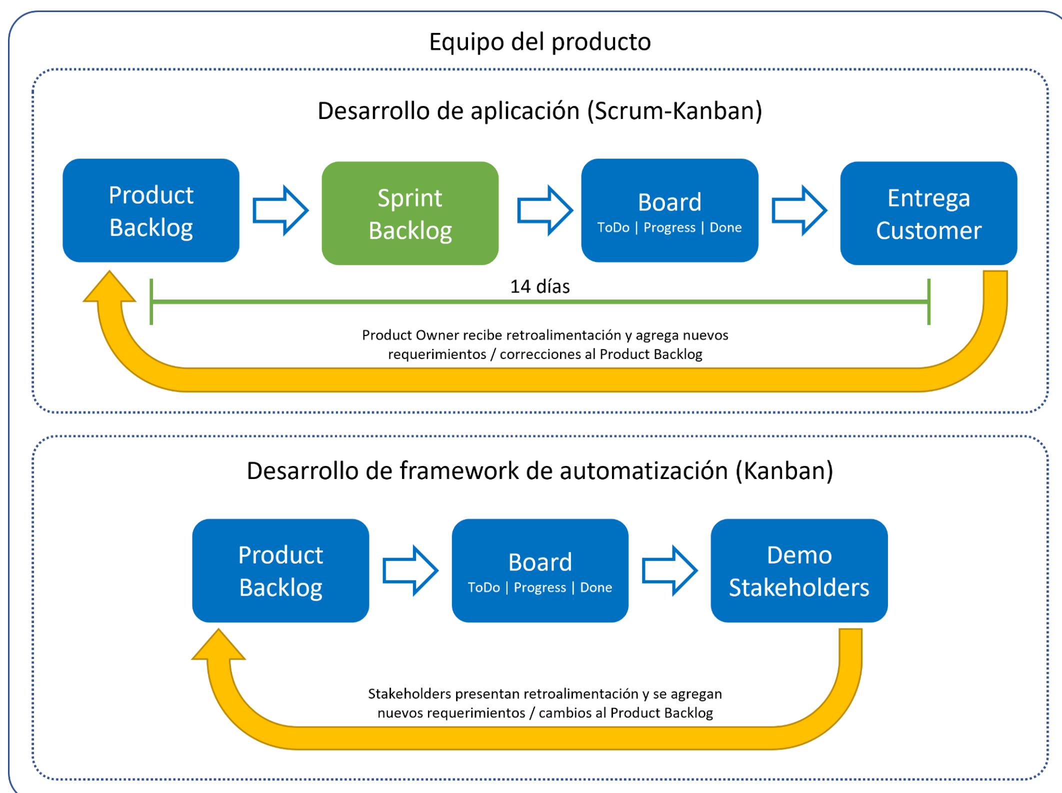

Figura 4.1 Vista de alto nivel de las metodologías involucradas en el flujo de trabajo del equipo del producto asignado.

Capítulo 5. Participación profesional

5.1 Propuesta inicial

Para desarrollar un proyecto con las características antes presentadas, requerimos de una administración de archivos que permitiera la escalabilidad, tanto para integrarlo con otros proyectos a futuro, así como para cualquier nueva funcionalidad que se implementara en la aplicación misma. De igual manera, esta estructuración debía ser de fácil mantenimiento, en caso de que se realizaran actualizaciones sustanciales a la IU, sobre la cual estuviera basado este framework de automatización.

Sabiendo que Robot Framework permite la modularización de los componentes que en él se desarrollan, partimos de esta idea para separar los esfuerzos y modularizar lo que conocíamos al momento, que existía una serie de pantallas de la aplicación que debían contar con sus *page objects* para partir de lo específico y de bajo nivel hacia lo abstracto y de alto nivel. De esta manera, el trabajo podía dividirse fácilmente entre los integrantes del proyecto.

Los *page objects* hacen referencia a las páginas o pantallas de la IU, como lo sería la pantalla de inicio de sesión. Estos objetos almacenan en sí los localizadores de los elementos de dicha página en el DOM, tales como el campo de texto del usuario, y el de la contraseña, así como el botón mismo de inicio de sesión, siguiendo el mismo ejemplo. Igualmente, dentro de este, se definen acciones básicas como ingresar texto en el campo del usuario, ingresar una contraseña en el campo de la contraseña, y de dar clic en el botón de inicio de sesión. Estas acciones, por definición, son llamadas *keywords*. Así, la dirección del esfuerzo se destinó a la creación de *keywords* para cada una de las pantallas, como se ilustra en la figura 5.1, donde para cada una de estas pantallas corresponde un archivo y, partiendo de estas, se pueden generar casos de prueba al armar estas *keywords*. En algunos casos específicos, es posible crear más de un archivo por pantalla, si la complejidad de esta hace conveniente dicha modularidad.

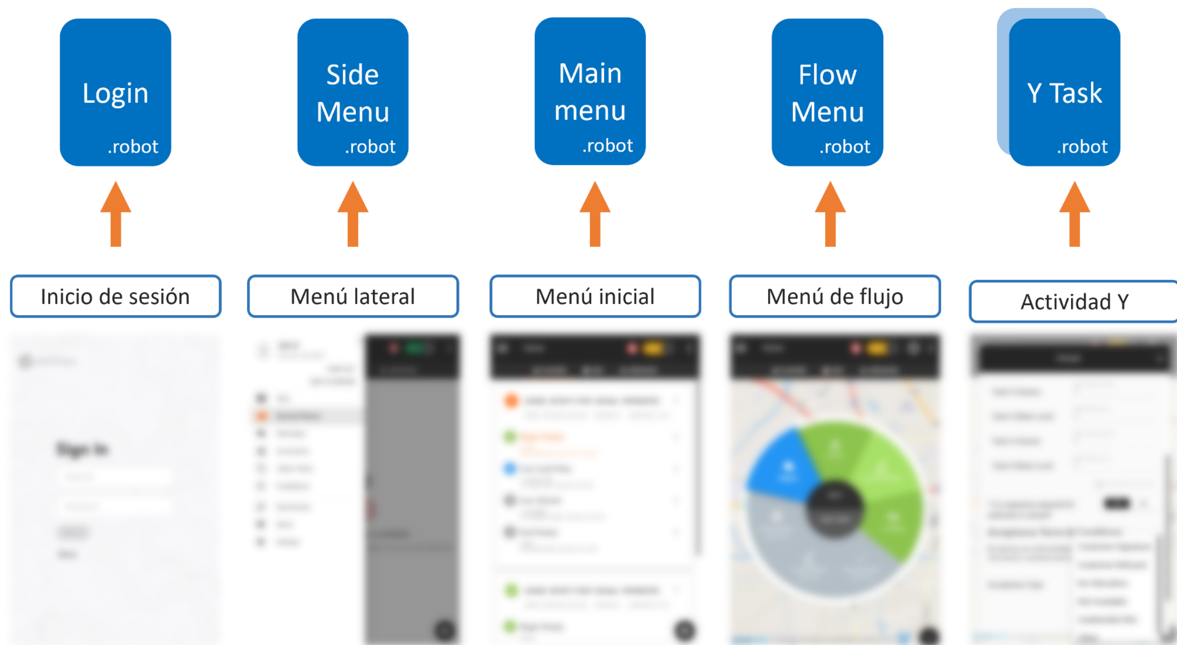

Figura 5.1 Relación entre pantallas y archivos *page object*.

5.2 Desarrollo del proyecto

Habiendo hablado ya sobre la propuesta inicial y los detalles técnicos esenciales del proyecto, a continuación se presentan las distintas fases que atravesé durante el desarrollo de este, sugiriendo cambios de diseño, y creando propuestas y herramientas que permitieron un avance continuo durante este periodo. Estas fases, desiguales en duración, comprenden el trabajo total de seis meses, y presento los detalles de sus etapas de análisis, diseño y entrega.

5.2.1 Fase 1

Durante esta fase, el equipo dedicado a la creación del framework de automatización estuvo comprendido por seis personas.

5.2.1.1 Análisis

Al ser la etapa inicial, y considerando que el *product backlog* ya existe, contamos con bastantes tareas por resolver, principalmente aquellas relacionadas a la creación de *page objects*. Dada la naturaleza del framework, podíamos dividir el trabajo y trabajar de manera independiente hasta tenerlas listas todas. Habría varias iteraciones sobre esta dirección.

5.2.1.2 Diseño

Para organizar estos esfuerzos, propusimos seguir el esquema inicial de estructura de Robot Framework, mismo que divide los archivos de recursos (los *page objects*) y los archivos de pruebas, con el fin de que estos últimos importen los primeros como recursos y de allí construyan las pruebas mismas. Esta estructura se ilustra en la figura 5.2.

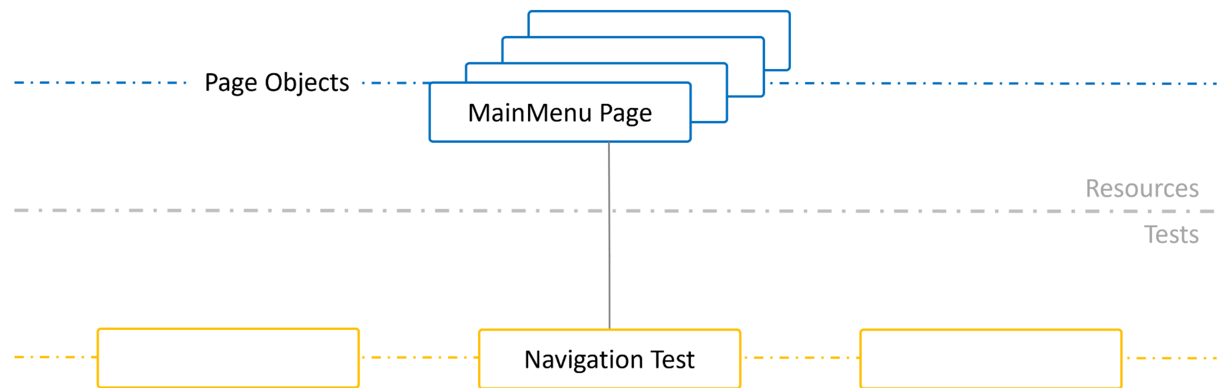

Figura 5.2 Estructura modular de archivos en la Fase 1.

5.2.1.3 Entrega

Como parte de los esfuerzos del equipo, implementé dos *page objects* de tareas del flujo de la aplicación, sin la creación de pruebas reales, sino "pruebas de prueba", para validar que el comportamiento automatizado era el esperado.

El entregable se vio incompleto durante esta etapa, dado que cuatro de los seis recursos del equipo se vieron reubicados a un proyecto de alta urgencia y prioridad, llevándose consigo además al *project manager* del equipo, quedando en este solamente dos personas por el resto de las siguientes etapas.

5.2.2 Fase 2

Dados los cambios al final de la fase anterior, decidí tomar la dirección del proyecto y reorganizar los esfuerzos aprovechando la oportunidad de ser sólo dos elementos trabajando sobre él, además de realizar las revisiones de código y resoluciones de conflictos en git. Mi superior se volvió un supervisor y soporte para el desarrollo del proyecto, y continuamos con la metodología Kanban antes especificada.

5.2.2.1 Análisis

Inicialmente, propuse y llegamos a los siguientes acuerdos,

- Revisar y reutilizar cualquier trabajo que fuera útil dentro del trabajo que quedó incompleto de la fase anterior.

- Dividir el trabajo basándose en la identificación de pantallas que utilizan lógica similar (tipos de elementos y su interacción).
- Mantener un canal de comunicación abierto para problemas y soluciones a estos, para evitar un retrabajo que el punto similar no haya prevenido.

Dado que el trabajo inicial restante era vasto, esta etapa tuvo una larga duración, y resolvimos comunicarlo al resto de los interesados.

5.2.2.2 Diseño

Conduje un *exploratory testing* en conjunto para identificar las distintas funcionalidades y estructuras de elementos en las distintas pantallas restantes por automatizar, para poder cumplir con el segundo acuerdo de la etapa de análisis. De estas pruebas exploratorias, segmenté las pantallas en aquellas que contaran con elementos puramente estáticos y en aquellas que tuvieran una combinación de dinámicos y estáticos; dentro de estos últimos, por el contexto de la pantalla, que sería la de las actividades de flujo, el menú de flujo, o el menú inicial, para dividir así los esfuerzos en módulos similares y evitar mayor retrabajo. La sección con mayor trabajo fue la de actividades flujo, pues se cubren alrededor de veinte pantallas para las distintas actividades que requiera cada cliente.

5.2.2.3 Entrega

Durante la creación de los *page objects* para las pantallas descritas, creé más “pruebas de prueba”, lo que terminó por convertirse en una prueba más grande, cuyo propósito fue el de completar un *happy path* que contuvo todas las pantallas antes vistas y validar su funcionamiento básico en secuencia. Como entregable, realicé una *demo* al equipo de automatización y al director de ingeniería para mostrar el avance y potencial del desarrollo del framework.

5.2.3 Fase 3

5.2.3.1 Análisis

Dado el gran crecimiento horizontal en la estructura de los *page objects*, se contaba con código similar o repetido en diversas ubicaciones del proyecto, y el objetivo hacia un proyecto escalable y sostenible comenzó a verse comprometido.

En este punto sugerí una capa adicional que contrarrestó el crecimiento de los archivos individuales y proveyó de una mejor administración para futuros cambios. En otras palabras, la idea se basó en separar la lógica de negocio del bajo nivel de los *page objects*, como se ejemplifica en la figura 5.3. De esta manera, y siguiendo el ejemplo presentado, si la manera de validar que una ruta está disponible cambia con una actualización, se modifica la “MainMenu Page”, al ser el contacto directo

con la IU, sin embargo, si los requerimientos del negocio para activar una ruta cambian, se modifica el documento “MainMenu Keywords”, sin alterar la implementación de bajo nivel, sino únicamente la *lógica del negocio*, o incluso la de las pruebas mismas.

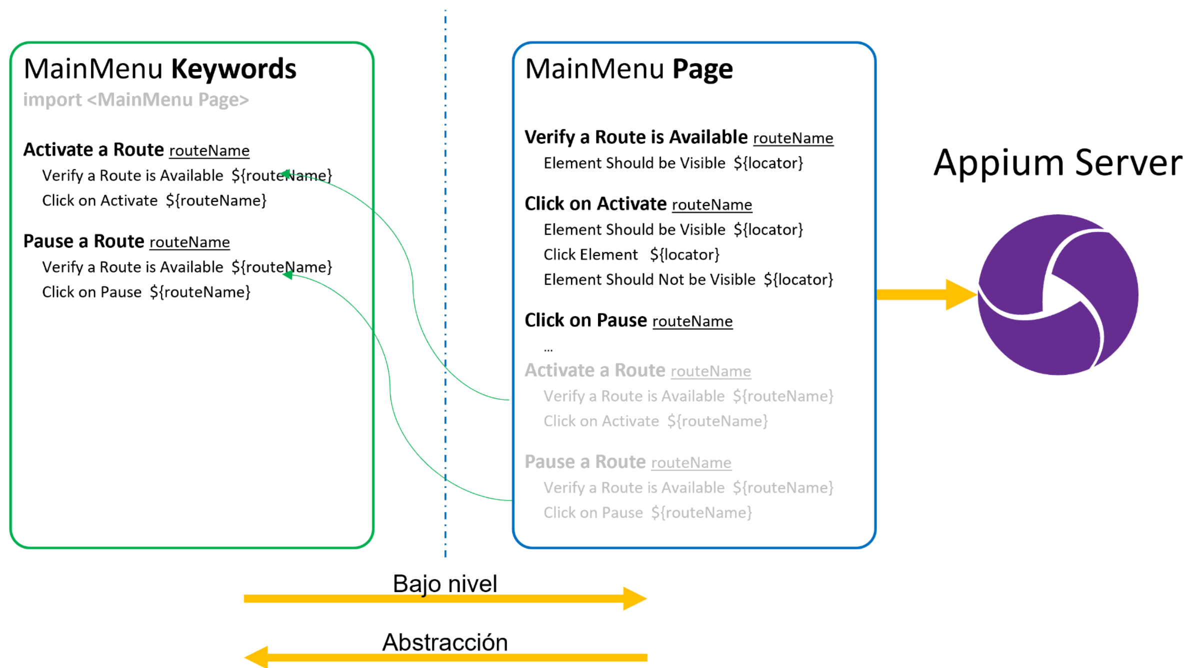

Figura 5.3 Separación de lógica de bajo nivel de la *lógica del negocio*.

Asimismo, encontré que existían diversos elementos gráficos comunes en todas las pantallas (o en grupos de estas), tales como el menú superior, el acceso al menú lateral, así como algunos elementos emergentes, y convenía evitar la duplicación de código al importar el uso de estos a una especie de biblioteca de utilidades, que todos los documentos que los usaran pudieran importar. De esta manera, el mantenimiento de ciertas funciones aplicaría para más de una pantalla, y el retrabajo sería menor a corto y largo plazo.

5.2.3.2 Diseño

Partiendo de lo anterior, realicé una propuesta sobre crear archivos de mayor jerarquía como se observa en la figura 5.3, así como sobre crear archivos centrales de recursos comunes entre las pantallas, al cual llamé *UI_Common*. *UI_Common* contendría funciones tales como el acceso automatizado a los menús que aparecen en varias pantallas, para ser implementados una única vez, así como también para los avisos emergentes de alertas o de confirmación de cambios, entre otras funciones como el deslizamiento de la pantalla por coordenadas relativas de algún elemento. Para elementos comunes entre grupos no globales de pantallas, se crearía un archivo *común* para dicho grupo. De esta manera, la estructura del proyecto tomó la forma que se ilustra en la figura 5.4.

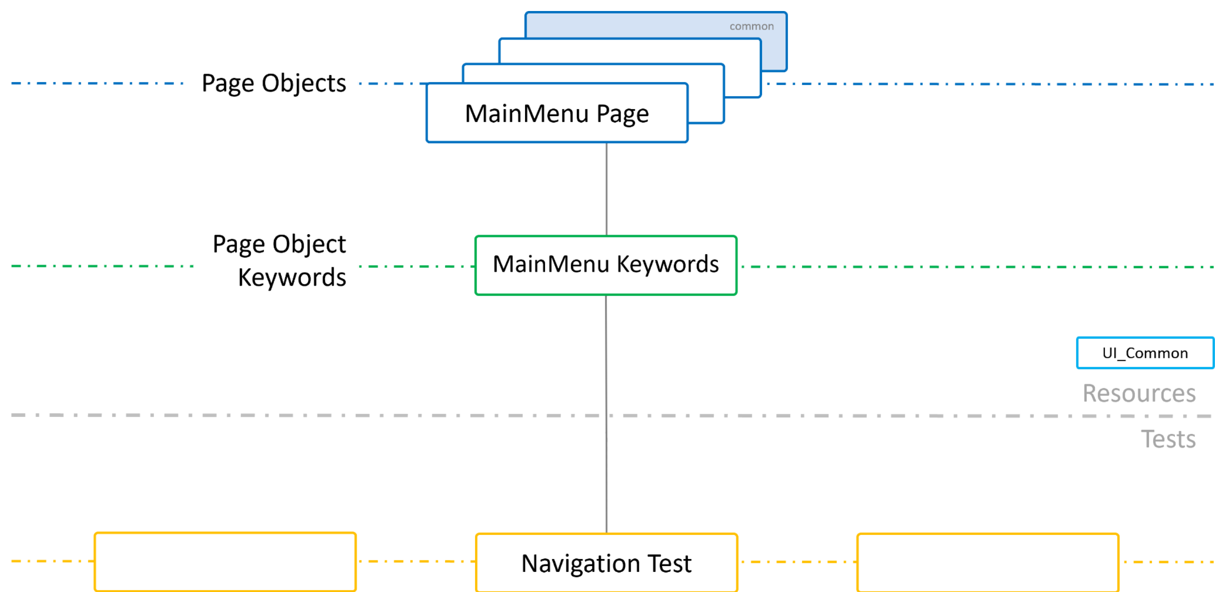

Figura 5.4 Estructura modular de archivos en la Fase 3.

5.2.3.3 Entrega

Coordiné y participé en el *refactor* de transición de las *keywords* ya existentes hacia archivos de mayor jerarquía con abstracción, para brindar una mejor estructura , escalabilidad y oportunidad de mantenimiento futuro a todo el proyecto. El entregable fueron dichas mejoras al framework y su respectiva documentación sobre los cambios y el nuevo acercamiento.

5.2.4 Fase 4

5.2.4.1 Análisis

El proyecto con el que se contaba estaba en una etapa medianamente funcional, puesto que se habían realizado esfuerzos para *mapear* todo lo posible de la aplicación y distintos escenarios en cada página, mas no se había enfocado aún hacia pruebas basadas en casos de uso, como se vio en el ejemplo de las tablas 1.2 y 1.3. El objetivo fue el de tomar una prueba y analizar qué *keywords* de los recursos ya creados darían la automatización de tal prueba; en otras palabras, el de poner en práctica el objetivo final del framework para presentar una demostración.

5.2.4.2 Diseño

Escogí una prueba que recorriera la secuencia de pasos que, para la aplicación, llevaría a cabo un conductor al completar una ruta, desde el inicio de sesión hasta dar por terminada la ruta y cerrar sesión (figura 5.5). Esto permitió probar al máximo el framework creado.

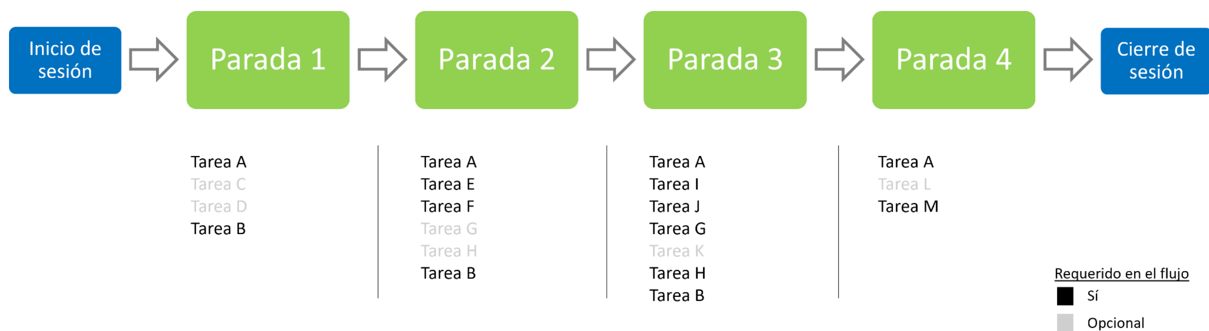

Figura 5.5 Secuencia del caso de prueba escogido.[10]

Para añadir soporte a futuros casos de prueba, con datos de entrada o de validación distintos, contemplé la creación de archivos de diccionario de datos, con lo cual se cortó la dependencia existente entre los archivos de pruebas y los datos. Lo anterior con el fin de evitar que, en un futuro, bajo el uso de distintos datos, no fuera necesario buscar los datos dentro del código, sino en un módulo específicamente empleado para ello.

5.2.4.3 Entrega

En el equipo generamos una prueba que cubrió toda la secuencia planeada, y noté que aún existía duplicidad de código, derivada de los nuevos archivos de mayor jerarquía, y acordamos analizar el tema en un futuro. Los datos de entrada se concentraron en un archivo para esa prueba y se utilizaron para validar la información mostrada, así como para ingresarla en la IU. La entrega que se realizó fue una demostración funcional de lo anterior con validación y entrada de datos.

5.2.5 Fase 4.1

5.2.5.1 Análisis

El desarrollo del proyecto se vió ralentizado por el alto tiempo de espera que existe entre cada cambio que se realiza y cada prueba del código, similar al tiempo de espera en que compila un programa y se verifica que funciona de manera esperada.

Investigué si existe alguna de las siguientes opciones para mejorar la productividad en nuestro equipo y perder el menor tiempo en dichas esperas:

- Cargar directamente la pantalla a probar (*activities* en Android).
- Saltar el proceso del inicio con una configuración de depuración en la aplicación.
- Utilizar *breakpoints* en Robot Framework, aunque no exista soporte oficial.

De lo anterior, encontré lo siguiente para cada punto, respectivamente:

[10] Los nombres reales de las paradas y tareas han sido modificados a nombres genéricos.

- Esta opción es realizable en aplicaciones nativas, mas no en aplicaciones híbridas, el cual era nuestro caso actual, una aplicación desarrollada en Apache Cordova de manera híbrida, además de tener implementación web. Descarté la idea.
- Al consultar con el equipo de desarrollo, entendí que este proceso no se puede saltar en una versión *release* de la aplicación, misma versión a la que estaba limitado el proyecto, dado su objetivo inicial. Descarté la idea.
- La inserción de *breakpoints* en el código era posible creando una biblioteca de Python que permitiera inyectar código por medio del depurador de Python (lenguaje sobre el cual se basa Robot Framework). Trabajé en esta idea.

5.2.5.2 Diseño

Propuse una idea basada en la inserción de código y el control de acceso a este mediante archivos *lock* que permitieran asimilar una función de *breakpoint* no oficial para Robot Framework. Se mandarían comandos basura al servidor de Appium para no dejar morir la sesión durante el tiempo en que no se utilizara.

4.3.5.3 Entrega

Generé una biblioteca en Python para activar la inyección de código de Appium en Robot Framework mediante el depurador de Python, y archivos *batch* para controlar el flujo con archivos *lock*. Se realizó una demostración interna sobre el programa y su potencial en aumentar la eficiencia de los equipos de automatización.

5.2.6 Fase 4.2

5.2.6.1 Análisis

La solución presentada en la fase 4.1 requiere de un entendimiento a profundidad del comportamiento interno de esta, de modo que para cumplir su objetivo final, opté por generar un programa con IU para controlar el flujo, la ejecución de los *batch*, así como para servir de ayuda en el formato de las instrucciones a inyectar; todo con el fin de abstraer el proceso y proveer de una herramienta útil para el futuro desarrollo del proyecto.

5.2.6.2 Diseño

Diseñe una solución con una IU con Python y Tkinter (figura 5.6) para correr las instrucciones de los archivos *lock*, identificar el directorio del proyecto a depurar, e insertar las instrucciones del usuario en el depurador en un proceso semiautomático.

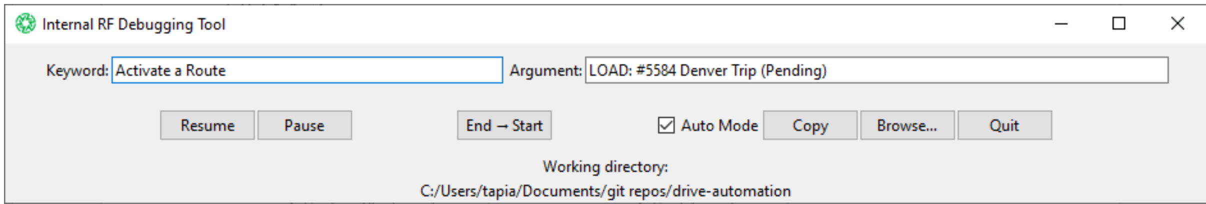

Figura 5.6 IU de la herramienta de depuración diseñada.

5.2.6.3 Entrega

Generé la solución de IU que realiza por detrás toda la lógica y la esconde del personal que realiza pruebas. Con estas soluciones, de fases 4.1 y 4.2, conseguí disminuir dramáticamente los tiempos de espera en hasta un 80%, como se ilustra en la figura 5.7, permitiendo así un desarrollo mucho más rápido del proyecto.

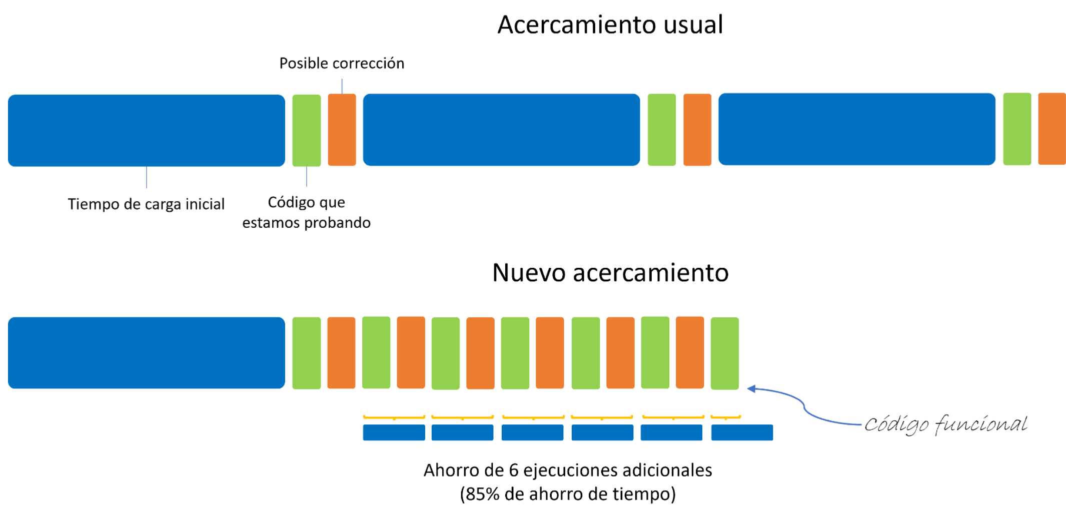

Figura 5.7 Comparativa de uso de tiempo ejemplificado en el nuevo acercamiento.

Anteriormente, se realizaban cambios en las automatizaciones y se probaban cumpliendo todos los prerrequisitos de las pruebas, lo que tomaba aproximadamente cuatro o cinco minutos, y si el cambio ingresado no funcionaba, la prueba terminaba y habría que ejecutar el mismo proceso, como se muestra a continuación en la figura 5.8.

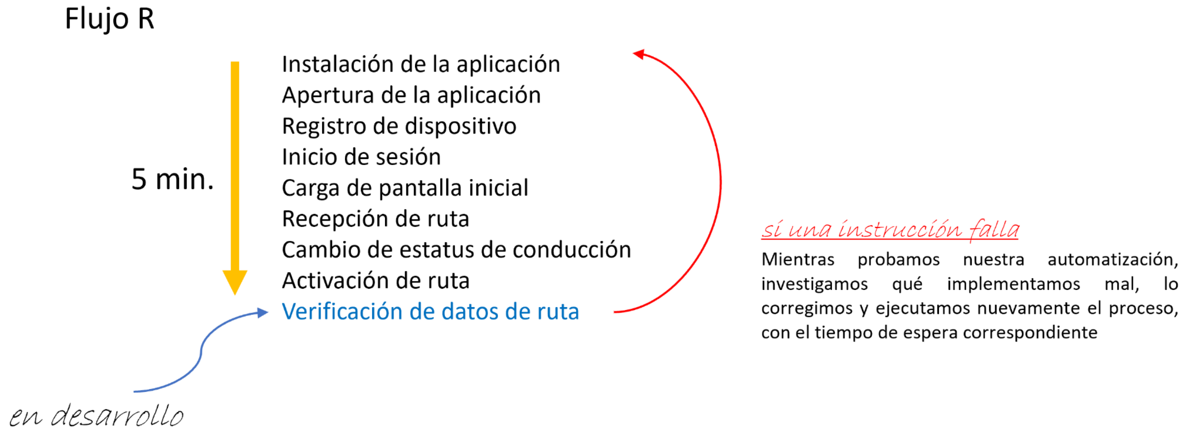

Figura 5.8 Desventajas de tiempo al ejecutar sin el uso de *breakpoints*.

5.2.7 Fase 5

5.2.7.1 Análisis

Con base en la prueba completa y en el rápido crecimiento provisto por la herramienta de la fase 4, notamos que la capa generada en la fase 3 no da todo el ancho necesario para cubrir las distintas configuraciones que, en relación con reusabilidad y modularidad, la construcción de pruebas automatizadas requiere. Por tanto, acordamos que hacía falta una tercera capa de abstracción (figura 5.9).

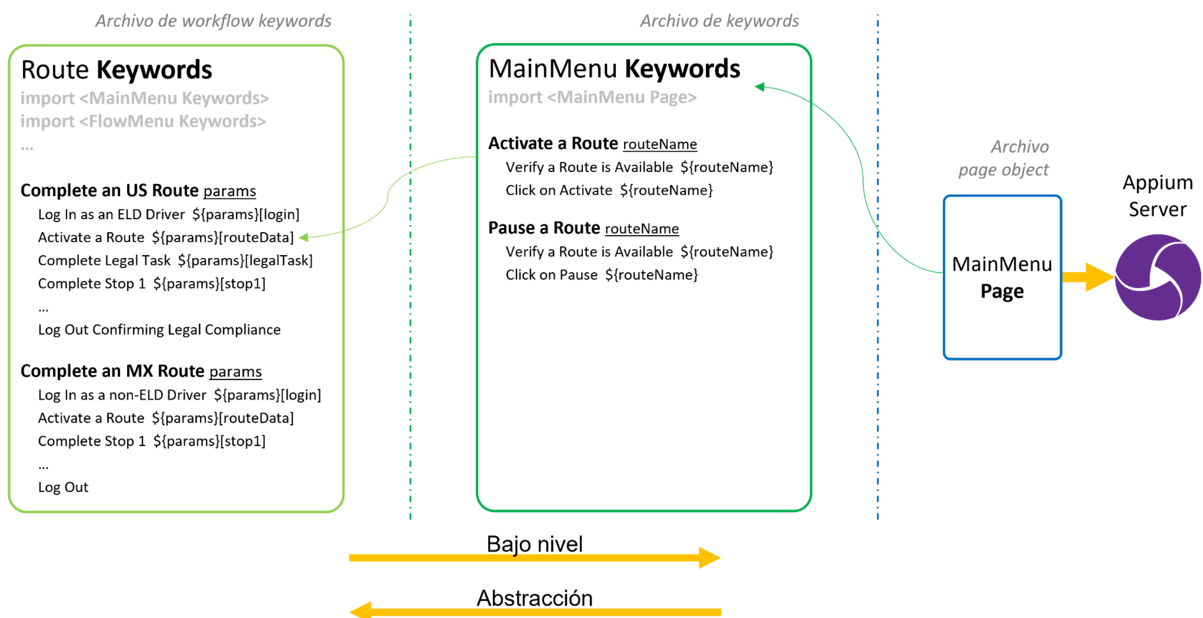

Figura 5.9 Separación de la lógica de medio y bajo nivel de la lógica de flujo.

5.2.7.2 Diseño

Una mejor manera de organizar los distintos acercamientos en el orden y uso de *keywords* se concibió al crear una tercera capa de abstracción, a la cual llamé *workflow keywords*, las cuales importan la segunda capa, y posiblemente extractos de la primera, en casos especiales. El objetivo de esta nueva capa es el de abstraer aún más el funcionamiento, pero principalmente el de proveer de distintas posibilidades en el "armado" de *keywords*, para mantener ya no sólo la reusabilidad y modularidad, sino también para preservar la legibilidad de las pruebas mismas. Los documentos de este tipo no estarían asociados a una página o pantalla de la aplicación, sino a un flujo dentro de esta. De esta manera, la estructura del proyecto tomó la forma que se ilustra en la figura 5.10.

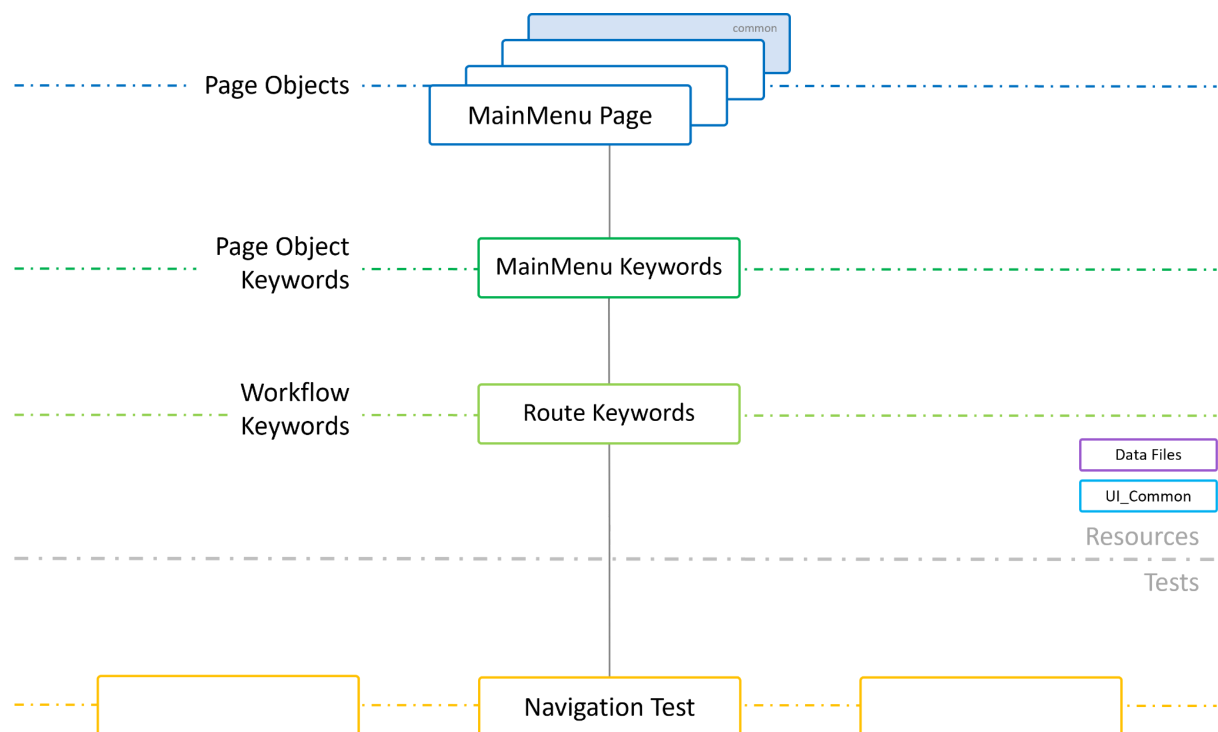

Figura 5.10 Estructura modular de archivos en la Fase 4.

5.2.7.3 Entrega

Trabajé en el *refactor* necesario para poder reescribir la prueba de la fase 4 con base en nuevas *workflow keywords* que cubren distintos subcasos de uso dentro de un solo flujo, de entre varios flujos. Esto permitió tener una mejor división entre la creación de pruebas automatizadas como scripts y entre la implementación de dicha automatización. El entregable para esta fase fueron dichas mejoras, así como su respectiva nueva documentación sobre la nueva estructura, y concluye el desarrollo del proyecto desde el punto de vista de automatización, restando únicamente la medición de tiempos.

5.2.8 Fase 6

5.2.8.1 Análisis

Dada la naturaleza del framework y de las tareas para las que se requiere medir el tiempo de carga, las instrucciones de medición debían hacerse explícitamente en el código, antes y después de cada tarea por medir. Estas mediciones debían arrojar archivos de log para su posterior análisis en Datadog.

5.2.8.2 Diseño

Para evitar un exceso de código, definí un acercamiento que únicamente invoca el cierre de la medición cuando es necesario (omitiendo espacios muertos que no nos interesaban medir), como se observa en la figura 5.11.

Figura 5.11 Estrategias de logging de tiempo.

5.2.8.3 Entrega

Generé una biblioteca de Python que recibía la acción actual y la anterior, así como *timestamps* y datos de cada fase, y con ella generaba las deltas de tiempos (de respuesta o de carga), y lo escribía hacia un archivo de logs, según fuera el caso de su invocación. El análisis de los datos se llevaría a cabo de una manera similar a la ilustrada en la figura 5.12. Como entrega de esta última fase, se realizó una demostración sobre esta funcionalidad del framework, así como una demostración general del framework a los equipos de automatización.

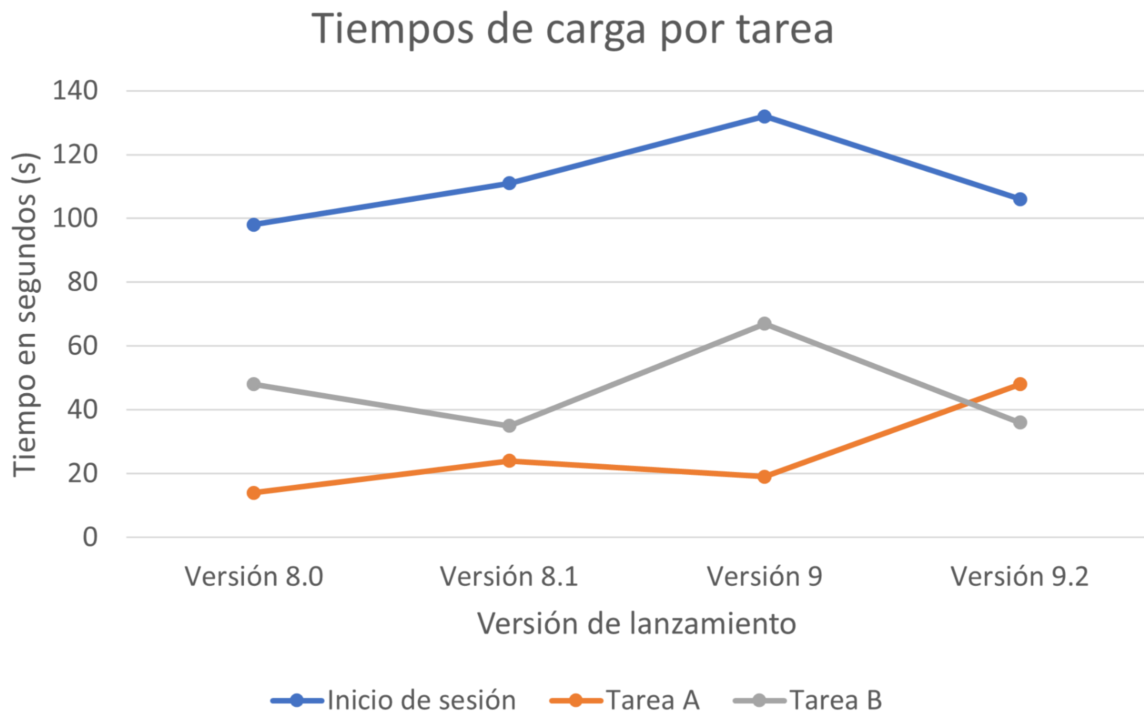

Figura 5.12 Gráfico para el análisis de los tiempos de carga por tarea.

Capítulo 6. Resultados

6.1 Framework de automatización de aplicación móvil

Al término del desarrollo del proyecto entregué un framework de automatización funcional para una de las aplicaciones de la empresa, el cual cubrió todas las bases para hacer de este una herramienta que posteriormente se podrá usar para generar pruebas funcionales de IU automatizadas (figura 6.1). Para este sistema cumplí con los alcances iniciales de escalabilidad, mantenimiento y reusabilidad basada en módulos al estructurar sus esfuerzos en distintas capas de abstracción, cada una con diferentes enfoques, en un desarrollo continuo e iterativo sobre el cual fui descubriendo y proponiendo mejoras, a la par de asumir responsabilidad del mismo. De igual manera, este proyecto es capaz de reportar mediciones históricas de tiempos de carga o de respuesta para su posterior tratamiento y análisis.

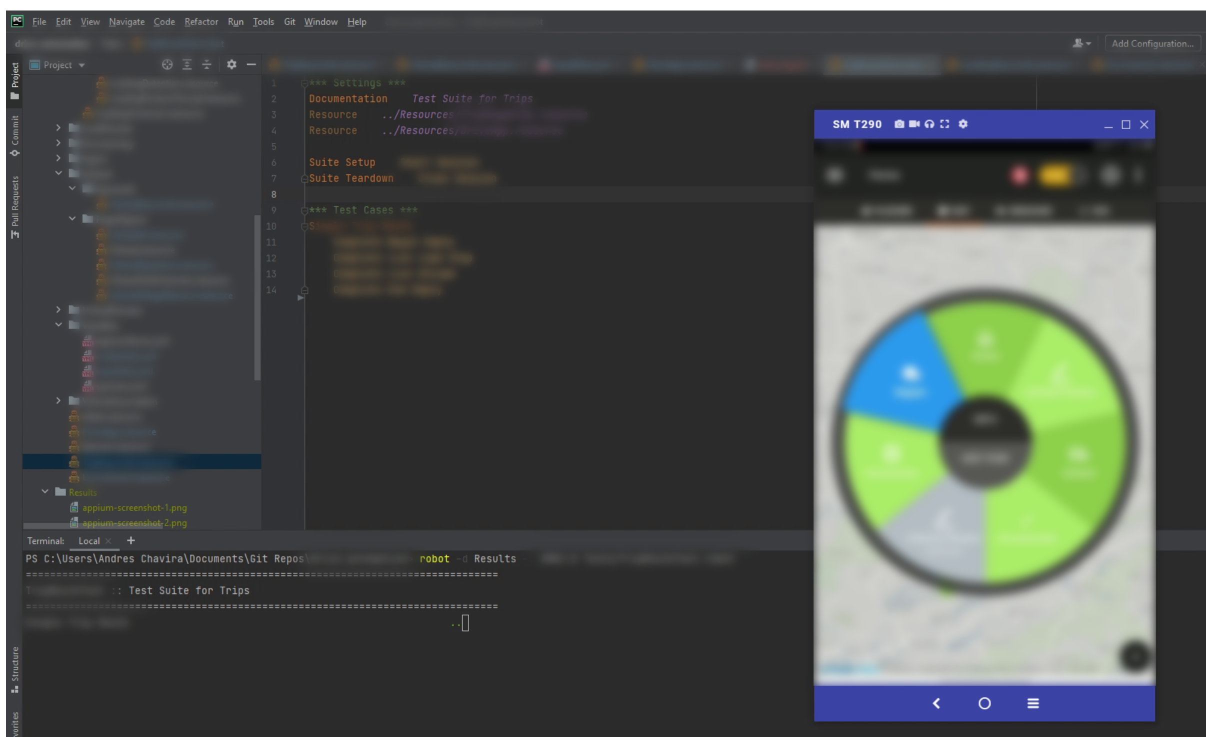

Figura 6.1 Ejecución de una prueba de completado de una ruta completa.

Este framework de automatización es capaz de ejecutar pruebas funcionales de distintas configuraciones, distintas entradas de datos, órdenes, casos especiales y en diferentes ambientes de desarrollo, con el fin de cubrir de manera integral cualquier prueba que se requiera automatizar en cualquier dispositivo Android versión 4.3 o superior. De esta manera, permite crear pruebas tan sencillas como complejas con el menor impacto en tiempo y mínima interacción con código, en virtualmente cualquier dispositivo.

6.2 Herramienta interna de depuración

Derivado del esfuerzo propio del desarrollo del framework de automatización, y del producto mismo de la mejora continua en mi manera de trabajar, también entregué una herramienta interna que facilita el desarrollo de automatización para Android, como explico en la sección 5.2.6.3, al reducir los tiempos de espera de manera dramática, permitiendo un flujo de trabajo fluido y evita pasar por uno similar al del desarrollo puro, donde por cada cambio existe un proceso de compilación. Dicha herramienta fue presentada y adoptada por otros equipos de automatización dentro de la empresa donde las limitaciones presentadas en la sección 5.2.5.1 también existen.

6.3 Documentación y sesiones de entrenamiento

Por último, elaboré y entregué documentación con respecto al proyecto, los acercamientos y desafíos técnicos para trabajarlo, así como guías de uso e instalación con el fin de instruir a futuros miembros del equipo. Por otra parte, elaboré y expuse presentaciones de entrenamiento a manera de inducción para otros equipos y nuevos elementos que se integraron al término del desarrollo del proyecto. Toda la documentación, las presentaciones y demostraciones antes mencionadas se condujeron en Inglés, y fueron publicadas o grabadas para su uso posterior.

Capítulo 7. Conclusiones

Al término del desarrollo del proyecto finalicé con conocimientos sobre pruebas y automatización, con dominio en Robot Framework y su fundación en Python, así como en Appium y en conceptos de escalabilidad, mantenimiento y diseño de sistemas. Dados los requerimientos del proyecto y el curso que este tomó, aumenté igualmente mis habilidades de comunicación, tanto para dialogar, escuchar y proponer ideas y resolver conflictos, como para llevar a cabo inducciones y demostraciones, lo que me permitió desempeñarme de mejor manera con personal de distintas áreas de la empresa. La constante búsqueda de mejora y cumplimiento de objetivos fomentó mi capacidad crítica, de análisis y resolución de problemas, al enfrentarme con desafíos técnicos y de diseño como se muestra en cada fase del presente informe, mientras que la dirección del proyecto me proveyó de una mejor habilidad para la toma de decisiones y la rendición de cuentas, así como una cultura de no imposibles.

De entre los desafíos encontrados en el equipo estuvieron presentes la disminución de los recursos dedicados en el desarrollo del proyecto, y la falta de una figura que realizara la dirección del mismo, lo cual ocasionó un proceso con menor velocidad, pero permitió mayor flexibilidad a los cambios que fui introduciendo en cada fase, y me permitieron a mí fungir como líder en un rol que de otra manera no habría podido obtener.

Al término del proyecto concretamos los alcances definidos inicialmente, de generar un sistema que permite realizar pruebas automatizadas, bajo distintos escenarios y con características de diseño de un sistema que admite cambios y adiciones constantes. Con este *framework* proveemos así de una manera intuitiva de generar pruebas automatizadas, aún sin un conocimiento a profundidad del mismo, para el uso dentro del área de QA en el equipo del producto.

Como trabajo futuro sobre el proyecto se encuentra la integración con otros proyectos, para llevar a cabo así pruebas automatizadas de integración sobre distintos productos interconectados, en donde la acción en uno de ellos se vea reflejado en alguno o algunos otros. Por otro lado, el objetivo final del proyecto dentro de la empresa es el de generar una suite de pruebas automatizadas para pruebas smoke y de regresión tras cada *deployment*, y su integración en una *pipeline* con ayuda de herramientas como Jenkins, para concebir un ciclo propio de DevOps al proveer de pruebas que se ejecuten en cada una de las iteraciones del ciclo.

Bibliografía

[1] Everett, G., & McLeod, R. D. (2006). Software Testing: Testing Across the Entire Software ExDevelopment Life Cycle. Wiley.

[2] Sommerville, I. (2016). *Software Engineering*. Pearson.

[3] Hüttermann, M. (2012). *DevOps for Developers*. Apress.

[4] NIST. (2002, junio 28). *Software Errors Cost U.S. Economy $59.5 Billion Annually*. NIST News Release. Recuperado el 15 de febrero, 2022, de https://web.archive.org/web/20200315132331/https://www.abeacha.com/NIST_press_release_bugs_cost.htm

[5] Hamilton, T. (2022, February 12). *Sanity Testing Vs Smoke Testing: Introduction and Differences*. Guru99. Recuperado el 15 de febrero, 2022, de https://www.guru99.com/smoke-sanity-testing.html

[6] Chamú, J. O., & González, A. (2020, November 16). *La importancia de separar los ambientes de trabajo en el desarrollo de los productos de software*. Portal TIC UNAM. Recuperado el 15 de febrero, 2022, de https://www.tic.unam.mx/2020/11/16/la-importancia-de-separar-los-ambientes-de-trabajo-en-el-desarrollo-de-los-productos-de-software/

[7] Macharla, P. (2017). *Android Continuous Integration: Build-Deploy-Test Automation for Android Mobile Apps*. Apress.

[8] *Introduction to the DOM - Web APIs | MDN*. (2021, September 14). MDN Web Docs. Recuperado el 15 de febrero, 2022, de https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

[9] Klärck, P. (n.d.). *Robot Framework Dos And Don'ts*. SlideShare. Retrieved February 15, 2022, from https://www.slideshare.net/pekkaklarck/robot-framework-dos-and-donts

[10] *¿Qué es DevOps? - Amazon Web Services (AWS)*. (n.d.). Amazon AWS. Recuperado el 15 de febrero, 2022, de https://aws.amazon.com/es/devops/what-is-devops/

[11] Pittet, S. (n.d.). *Comparación de integración continua, entrega continua e implementación continua*. Atlassian. Recuperado el 15 de febrero, 2022, de https://www.atlassian.com/es/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

[12] Dalton, J. (2018). *Great Big Agile: An OS for Agile Leaders*. Apress.

[13] *Introduction to Appium*. (2020). Appium. Recuperado el 15 de febrero, 2022, de https://appium.io/docs/en/about-appium/intro/

[14] *Introduction*. (n.d.). Robot Framework. Recuperado el 15 de febrero, 2022, de https://robotframework.org/

[15] Versluis, G. (2017). *Xamarin Continuous Integration and Delivery: Team Services, Test Cloud, and HockeyApp*. Apress.