UNIVERSIDAD NACIONAL DE COLOMBIA

Automatización de un Sistema de Pruebas de Software para la Optimización del Proceso de Calidad de DetectID™

Ángela Paola Duquino Sánchez

Universidad Nacional de Colombia
Facultad de Ingeniería
Área Curricular de Ingeniería Eléctrica y Electrónica
Bogotá, Colombia
2020

Automatización de un Sistema de Pruebas de Software para la Optimización del Proceso de Calidad de DetectID^{TM}

Ángela Paola Duquino Sánchez

Trabajo de grado presentado como requisito parcial para optar al título de:
Magister en Automatización de Industrial

Director:
Javier Rosero García, PhD

Facultad de Ingeniería
Área Curricular de Ingeniería Eléctrica y Electrónica
Bogotá, Colombia

2020

Resumen

El presente proyecto busca analizar las tecnologías implementadas actualmente para definir la arquitectura de integración continua que permita la optimización del proceso de calidad de software, y por tanto, mejorar la productividad en cuanto al desarrollo de software. Esta optimización permite que los usuarios finales reciban retroalimentación rápida y continua de los problemas de calidad y que se haga un manejo más efectivo de los errores (o denominados BUGS[1]) presentados en el ciclo de vida del software.

El proceso de calidad de software debe garantizar que el software cumpla con las funciones para las que fue diseñado y permita al usuario final un manejo efectivo del entorno desarrollado, por medio de la verificación y validación del software mediante el sistema de pruebas. Un sistema de pruebas se crea a partir de los requerimientos de usuario (objetivos y funcionalidades del software) y es implementado por medio de unos escenarios de prueba en donde se indican el paso a paso a realizar dentro del entorno desarrollado para garantizar el cumplimiento de los objetivos del mismo.

De esta manera, en el presente proyecto se implementa un sistema de automatización de pruebas, que permita mejorar el proceso de calidad y pruebas y adicionalmente el proceso de desarrollo de software. En primer lugar se estudia en la literatura las implementaciones llevadas a cabo actualmente, a continuación se analizan los procesos y objetivos del software de DetectID móvil y se procede a establecer las pautas, tecnologías y metodologías más adecuadas, que permitan que el sistema de automatización de pruebas tenga una respuesta efectiva, eficiente y óptima dentro del ciclo de vida del desarrollo de software, para garantizar su calidad, integración continua y manejo rápido de errores.

Finalmente, se implementan las metodologías y tecnologías definidas en el software de DetectID móvil, para realizar un análisis de resultados donde se muestra la importancia del sistema de automatización de pruebas y la integración continua dentro del ciclo de software.

Palabras clave:
Desarrollo de pruebas, aseguramiento de calidad de software, automatización de pruebas, integración continua.

[1]BUG: Problema o falla en un software que desencadena un resultado indeseado.

Contenido

Resumen v

1. Introducción 2

2. Marco Teórico 6

2.1. Antecedentes del desarrollo de software 6
2.2. Desarrollo de software 7
2.2.1. Ciclo de vida del desarrollo de software 8
2.2.2. Modelos de desarrollo de software 10
2.3. Objetivos de las pruebas 12
2.4. Proceso de pruebas 15
2.4.1. Planificación y estimación 15
2.4.2. Análisis y diseño 17
2.4.3. Implementación y ejecución 18
2.4.4. Evaluación de criterios de salida e informes 19
2.4.5. Seguimiento y control 19
2.4.6. Actividades de cierre 20
2.5. Enfoque de las pruebas 21
2.6. Técnicas de diseño de pruebas 21
2.6.1. Pruebas de Caja Blanca basadas en la estructura 22
2.6.2. Pruebas de Caja Negra basadas en la especificación 22
2.6.3. Pruebas basadas en la experiencia 23
2.7. Tipos de pruebas 23
2.7.1. Pruebas Estáticas 23
2.7.2. Pruebas Dinámicas 24
2.8. Niveles de pruebas 24
2.8.1. Pruebas de Componente 24
2.8.2. Pruebas de Integración 25
2.8.3. Pruebas de Sistema 25
2.8.4. Pruebas de Aceptación 26
2.9. Automatización de pruebas de software 27
2.10. Integración Continua 32

3. Estado del Arte **36**
3.1. Metodologías/Procesos de automatización de pruebas en la actualidad 36
3.1.1. Estrategias de pruebas 37
3.1.2. Pirámide de pruebas 39
3.1.3. Estrategias de automatización 40
3.1.4. Problemas en las pruebas automatizadas 42
3.1.5. Composición del sistema automatizado de pruebas 43
3.2. Software DetectID™ 45
3.2.1. Autenticación multifactorial 46
3.2.2. Factores de autenticación DetectID ™ 46
3.3. Metodología/proceso de desarrollo de software en DetectID ™ móvil 48
3.3.1. Planeación 48
3.3.2. Análisis y Diseño 48
3.3.3. Desarrollo 49
3.3.4. Pruebas 49
3.3.5. Entrega/Release 50
3.3.6. Integración Continua 50

4. Implementación: Automatización del Sistema de Pruebas de DetectID™ **52**
4.1. Metodología para la automatización de un sistema de pruebas 53
4.1.1. Flujo de la metodología de automatización de pruebas 54
4.1.2. Requerimientos para la condición exitosa 55
4.1.3. Ejecución de pruebas con Cucumber 55
4.1.4. Mock como estrategia para las pruebas 56
4.1.5. Generación de reportes de las pruebas automatizadas 56
4.1.6. Jenkins como estrategia de integración continua (CI) 57
4.2. Creación framework/proyecto de automatización de pruebas 57
4.2.1. Estructura del proyecto Android 58
4.2.2. Estructura del proyecto iOS 61
4.3. Implementación de escenarios a automatizar 64
4.4. Configuración del entorno de pruebas 66
4.4.1. Implementación con Cucumber - Android 66
4.4.2. Implementación con Cucumberish - iOS 68
4.5. Implementación de Mocks y creación de stubs 69
4.5.1. Implementación en Android 69
4.5.2. Implementación en iOS 70
4.6. Integración del sistema de pruebas automatizado con el SDK-DID 71
4.6.1. Implementación en Android 71
4.6.2. Implementación en iOS 72

4.7. Implementación de los pasos a ejecutar en cada escenario 74
4.7.1. Implementación en Android 74
4.7.2. Implementación en iOS 74
4.8. Ejecución de las pruebas 74
4.8.1. Ejecución en Android 75
4.8.2. Ejecución en iOS 75
4.9. Generación de informes con los resultados de las pruebas ejecutadas 75
4.9.1. Generación de informes en Android 75
4.9.2. Generación de informes en iOS 81
4.10. Implementación del sistema de pruebas automatizado en un entorno de integración continua 82
4.10.1. Implementación en Android 82
4.10.2. Implementación en iOS 83
4.10.3. Integración del SDK-DID y el sistema de pruebas automatizado dentro de un entorno de integración continua 84

5. Resultados 94
5.1. Metodología de desarrollo 95
5.2. Estrategia de pruebas 96
5.3. Ejecución y cobertura de las pruebas 97
5.4. Tiempos de ejecución de pruebas 98
5.5. Metodología de integración continua 99
5.6. Generación de reportes 100
5.7. Detección de errores dentro de la ejecución 101
5.8. Generación de notificaciones sobre el estado de la ejecución 101
5.9. Herramientas utilizadas 102
5.10. Plataformas y lenguajes soportados 103

6. Conclusiones y Recomendaciones 106
6.1. Conclusiones 106
6.2. Recomendaciones 108

A. Anexo: Implementación frameworks 110
A.1. Features 110
A.1.1. Feature: Register devices by code 110
A.1.2. Feature: Lifetime OTP 111
A.2. Configuración entorno de pruebas 111
A.2.1. Android 111
A.2.2. iOS 116
A.3. Implementación del Mock 119
A.3.1. Android 119

A.3.2. iOS 121
A.4. Integración con el SDK-DID 125
A.4.1. Android 125
A.5. Implementación de los pasos a ejecutar en cada escenario 128
A.5.1. Android 128
A.5.2. iOS 129
A.6. Ejecución de las pruebas 129
A.6.1. Android 129
A.6.2. iOS 132
A.7. Generación de reportes en iOS 133
A.8. Implementación sistema de pruebas automatizado en un entorno de integración continua 135
A.8.1. Android 135
A.8.2. iOS 136

B. Anexo: Implementación CI 138
B.1. Pipeline Administrador 138
B.2. Implementación JaCoCo 139
B.3. Implementación Sonarqube 140
B.3.1. Android 140
B.3.2. iOS 140

Referencias bibliográficas 142

1. Introducción

El desarrollo de software se ha convertido en un proceso fundamental en la calidad de vida de los seres humanos, dando paso a la implementación de nuevas tecnologías que permiten la ejecución de procesos de la vida cotidiana más eficientes, seguros, fáciles y accesibles. Sin embargo, a medida que se generan nuevos avances tecnológicos así mismo el software debe estar a la vanguardia, implementando mejoras en sus funcionalidades y ciclo de desarrollo. El ciclo de vida de desarrollo es un ciclo que se repite constantemente, conforme se generan nuevos criterios o necesidades para los usuarios, que exigen a los propietarios del software mejoras, desarrollo e implementaciones continuas.

El proceso de desarrollo requiere de una continua retroalimentación de sus procesos que les provea a los desarrolladores y probadores[1] la información suficiente y adecuada para hacer seguimiento de la calidad del software y la correcta gestión de un sistema de pruebas.

Los principales beneficios de la ejecución de pruebas de software son [3]:

- Reducir el riesgo de errores.
- Garantizar la calidad del software.
- Identificar defectos.
- Identificar cambios no intencionados.

Dada la robustez del ciclo de vida del software en el cual entregar una funcionalidad al cliente es un proceso repetitivo, complejo y demorado, es necesario proveer los procesos y herramientas tecnológicas adecuadas que permitan facilitar y optimizar el ciclo de vida de desarrollo de software, investigar los procesos y metodologías sugeridas por la literatura y análisis de datos, para implementar mejoras en el proceso de pruebas del desarrollo de software. Para dar solución a esta necesidad se plantea la siguiente pregunta:

[1] Probadores (testers): Personas que se encargan de probar el software para garantizar su calidad [3]

¿Cómo implementar un sistema automático de pruebas basado en las tecnologías de integración continua, que permita la ejecución de pruebas, análisis de resultados, certificación del cumplimiento de los criterios de usuario en diferentes plataformas, y por tanto optimización del ciclo de vida del software en DetectID^{TM} móvil?

Con el fin de dar respuesta a esta pregunta en el presente proyecto se realiza una introducción y antecedentes de los conceptos, procesos y metodologías que abarca el desarrollo de software y el proceso de pruebas. A continuación se caracteriza el software **DetectID^{TM}** móvil de la compañía **AppGate** para establecer la estructura y arquitectura para la automatización de un sistema de pruebas que permita preparar un entorno de pruebas, posteriormente se lleva a cabo ejecución de pruebas dentro del entorno configurado para gestionar en tiempo real el sistema de pruebas y a partir de los reportes de prueba generados, certificar que el software del producto cumpla con los requerimientos de usuario para los que fue diseñado. Por tanto, se investigan las metodologías de automatización de sistemas de pruebas implementadas actualmente para definir una metodología propia que se ajuste a las necesidades del producto.

Así mismo, se implementa el sistema de pruebas utilizando las tecnologías de integración continua que permitan integrar por completo dicho sistema de pruebas con el software en estudio, garantizando que el ciclo de desarrollo de software y la retroalimentación de la calidad del mismo sean más continuos, efectivos, rápidos, claros y provean la información suficiente para establecer las causas de los problemas y la visibilidad del cumplimiento de la funcionalidad del software a partir del sistemas de pruebas. Finalmente, se muestran al lector los resultados obtenidos de la implementación y se definen las conclusiones y recomendaciones del presente proyecto.

La metodología o proceso que se lleva a cabo para la ejecución del proyecto se muestra en la Figura **1-1**, y se compone específicamente de:

- **Marco Teórico** donde se realiza una introducción a:
  - Desarrollo de software y aseguramiento de calidad en el software, antecedentes, procesos, metodologías, tipos, niveles, técnicas, objetivos, entre otros.
  - Automatización de pruebas e integración continua.
- **Estado del Arte** donde e realiza una descripción de:
  - Las metodologías y procesos implementados actualmente en el desarrollo de software, automatización de pruebas e integración continua.
  - Software DetectID^{TM} móvil y sus procesos.

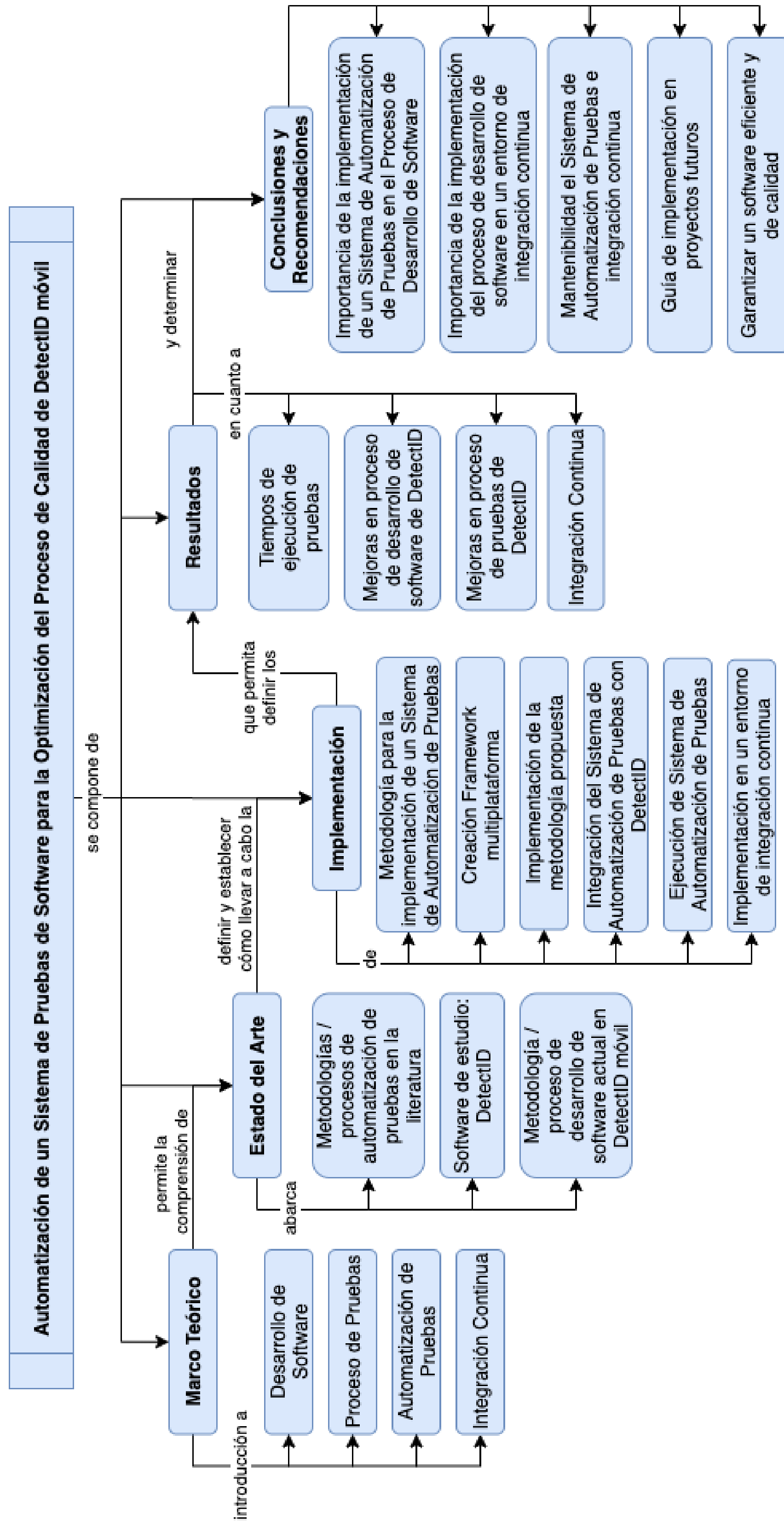


Figura 1-1.: Metodología para abordar el problema.

- **Implementación** donde se define la metodología de automatización de pruebas a ejecutar junto con la creación de frameworks multiplataforma (Android-iOS), ejecución del sistema de pruebas automatizado, integración con el DetectID^{TM} móvil dentro de un entorno de integración continua.

- **Resultados** de la implementación del sistema de pruebas automatizado en un entorno de integración continua, teniendo en cuenta tiempos de ejecución de pruebas, mejoras en el software DetectID^{TM} móvil, reporte o documentación del mismo, entre otros.

- **Conclusiones y recomendaciones** sobre la importancia de la implementación del sistemas de automatización de pruebas en de DetectID^{TM} móvil dentro de un entorno de integración continua, mantenibilidad de dicho sistema y generación de software eficiente y de calidad garantizada.

2. Marco Teórico

Para comprender la importancia del presente proyecto, es importante revisar los antecedentes e historia del Desarrollo de Software, los procesos que se han ido implementando con el desarrollo de nuevas tecnologías las cuales permiten una mejora continua en estos procesos. En el presente capítulo se realiza una introducción a:

- Desarrollo de Software: sus antecedentes, ciclo de vida, procesos, metodologías, y modelos.
- Aseguramiento de calidad en el software o Pruebas: sus objetivos, proceso, enfoque, técnicas, tipos, niveles, entre otros (Figura **2-3**).
- Automatización de pruebas: pruebas continuas (continuos testing), proceso de automatización de pruebas, sus objetivos, principios, entre otros (Figura **2-5**).
- Integración continua: sus objetivos, composición, proceso, cómo la integración continua permite la automatización completa de procesos dentro del ciclo de desarrollo de software, entre otros (Figura **2-7**).

2.1. Antecedentes del desarrollo de software

Los avances en conocimiento científico dan paso a la implementación de nuevas tecnologías en el hardware, redes, comunicaciones, información y han permitido la evolución del software, el cual ha cambiado con el tiempo con la implementación de procesos, metodologías, herramientas, lenguajes, entre otros, de tal manera que la interacción de los procesos de software ejecutados en un hardware sea efectiva, su respuesta sea lo más rápida posible, amena, consistente, clara y adecuada para el usuario final.

Anteriormente se requería de máquinas que permitieran el almacenamiento de grandes cantidades de información y el hardware era una de las principales preocupaciones para un correcto funcionamiento del software. Actualmente es posible llevarlo a cabo desde la nube sin tener que preocuparse por el hardware y las tecnologías de la información han adquirido un importante papel para la interacción de estas dos partes fundamentales: Software-Hardware (Figura **2-1**).

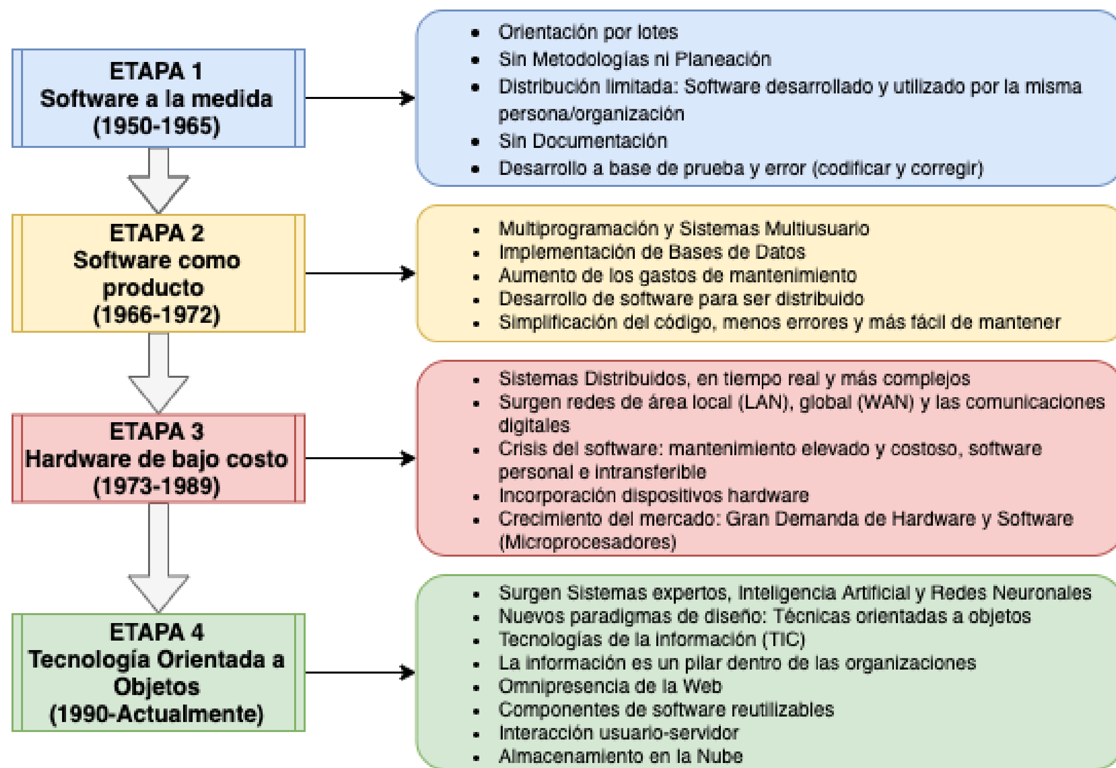


Figura 2-1.: Línea de tiempo de la evolución del Software.

Como se muestra en la Figura 2-1 con la evolución se han generado metodologías en el desarrollo de software que en un principio no tenían un proceso definido para su planeación, desarrollo, ejecución, mantenimiento y documentación. Así mismo, ha permitido que el software tenga menos errores en su desarrollo e implementación, que la respuesta ante cualquier problema y su corrección sea lo más rápida y efectiva posible y garantizar su calidad, mantenimiento, distribución y acceso a la información del mismo por medio de la administración de los procesos y metodologías del software.

2.2. Desarrollo de software

Con el desarrollo de software se pueden implementar sistemas informáticos con determinadas funcionalidades que permiten la interacción con un hardware específico. La implementación de estos sistemas informáticos o aplicaciones/funcionalidades tanto web como móviles está en auge dado que estas aplicaciones son más accesibles y permiten llevar a cabo actividades de la vida cotidiana de forma más rápida, fácil, segura y eficiente, ajustándose a las necesidades del usuario. El software ha dado paso a la implementación de funcionalidades que:

- Permiten la interacción/comunicación entre diferentes usuarios (redes sociales).

- Evitan a los usuarios desplazarse a determinados lugares para hacer todo tipo de transacciones y consultas (domicilios, servicio al cliente).
- Permiten al usuario desplazarse de un lugar a otro cuando lo requiera (uber).
- Protegen la información personal de los usuarios, dada su interacción con todo tipo de plataformas (autenticación segura).
- Permiten realizar transacciones desde un dispositivo/hardware (plataformas bancarias).
- Permiten realizar cálculos.
- Permiten llevar a cabo actividades de aprendizaje (en plataformas educativas).
- Almacenan determinado tipo de información (agendas, registro gastos, calendarios).
- Proveen de información en tiempo real (tráfico, clima).
- Permiten el diseño y creación de diferente contenido multimedia.
- Permiten al usuario interactuar con diferentes tipos de juegos.

La competitividad en el desarrollo de software ha aumentado, así como los diversos lenguajes de programación y herramientas de diseño, programación, implementación y ejecución de software. Para poder diseñar, implementar y entregar un software de calidad competitivo, que se ajuste a los requerimientos y necesidades del cliente es importante comprender las características, procesos, buenas prácticas, del desarrollo de software.

2.2.1. Ciclo de vida del desarrollo de software

El desarrollo de software requiere de unos procesos que le permitan su correcta ejecución y tener claridad del ciclo de vida del desarrollo de software. También conocido como SDLC es el proceso de planificación, desarrollo, prueba e implementación de software que sigue una secuencia de fases y cada fase utiliza el resultado de su fase anterior como se muestra en el diagrama de la Figura 2-2.

El ciclo de vida del desarrollo del software se compone de:

- **Análisis de Requerimientos**: Requisitos de las necesidades comerciales del proyecto, los cuales pueden ser internos (organización) o externos (cliente). El análisis implica definir el alcance de los requisitos, para establecer los objetivos y con la información recopilada, definir si la propuesta pretende mejorar el sistema o crear uno nuevo.

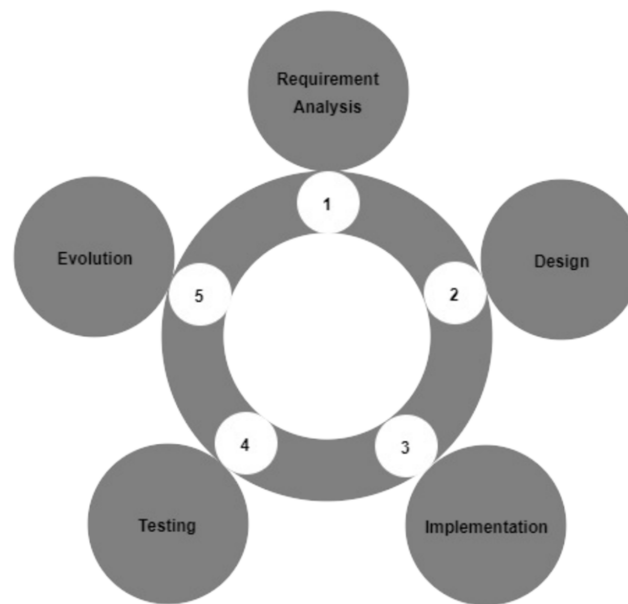


Figura 2-2.: Ciclo de vida del desarrollo de software [1].

- **Diseño**: Se formulan las características deseadas de la solución de software y se crea un plan de proyecto que incluye diagramas de proceso, interfaz general y parámetros diseño, junto con un amplio conjunto de documentación. Por tanto, implica realizar: análisis de sistemas, definición arquitectura de software, diseño de bases de datos y diseño de software.

- **Implementación**: Desarrollar el código en función de las tareas y objetivos definidos en la fase de diseño e implica adicionalmente ejecutar pruebas a nivel local para comprobar el funcionamiento previo del código implementado.

- **Pruebas**: Todas las características se prueban exhaustivamente para verificar la calidad, cumplimiento de requisitos y presencia de errores en el software. Si se presentan errores debido a defectos en la implementación, el equipo de desarrollo inmediatamente resuelve dichos problemas, para que finalmente el código probado sea implementado en el entorno de producción.

- **Evolución**: Se analizan los comentarios de los usuarios y se repite todo el ciclo de desarrollo, prueba y lanzamiento de las nuevas características y soluciones en forma de parches o actualizaciones e implica: mantenimiento, documentación y soporte.

Este ciclo de vida del desarrollo de software permite definir con mayor claridad las pautas, requerimientos, avances, proyecciones para todo tipo software, la implementación de tecnologías y metodologías para su ejecución, y garantiza la calidad del mismo, para una entrega al usuario final.

2.2.2. Modelos de desarrollo de software

El proceso de desarrollo adoptado para un proyecto dependerá de los objetivos y metas del proyecto ya que existen numerosos ciclos de vida de desarrollo implementados. Estos ciclos de vida van desde metodologías livianas y rápidas donde el tiempo de comercialización es esencial hasta metodologías totalmente controladas y documentadas donde la calidad y la confiabilidad son factores clave. Los modelos de Desarrollo de Software son los siguientes:

- **Modelo Cascada**
  El modelo de cascada es un proceso de desarrollo de software secuencial que se ejecutan en una dirección, en este modelo una vez se da inicia al desarrollo no se permite ninguna modificación en el diseño.

  Ventajas [1]

  - Los requisitos están bien documentados y arreglados.
  - La tecnología es fija y no dinámica.
  - No hay requisitos ambiguos, ni aparecen durante ninguna otra fase aparte de la fase de análisis de requisitos.

  Desventajas [1]

  - El software entregable para el usuario final se produce solo al final del ciclo de vida.
  - No es recomendable para proyectos donde la demanda de nuevas características es frecuente.
  - La integración se realiza únicamente cuando se completa toda la fase de desarrollo, por lo que los problemas de integración se encuentran en su etapa final y en grandes cantidades.
  - No hay trazabilidad hacia atrás.
  - Es difícil medir el progreso dentro de las etapas.

- **Modelo V**
  El Modelo V es un modelo secuencial que ilustra cómo las actividades de prueba (verificación y validación) pueden integrarse en cada fase del ciclo de vida. Dentro de este modelo las pruebas de validación se realizan especialmente durante las primeras etapas donde se revisan los requisitos del usuario, y al final del ciclo de vida durante la ejecución de las pruebas de aceptación. El modelo V se basa en cuatro niveles de prueba:
  - Pruebas de componentes.
  - Pruebas de integración.
  - Pruebas del sistema.
  - Pruebas de aceptación.

  Los niveles de prueba se pueden combinar o reorganizar según la naturaleza del proyecto o la arquitectura del sistema.

- **Modelos de desarrollo iterativo-incremental**
  Se basa en la ejecución de varias fases de ciclo de vida autónomas más pequeñas para el mismo proyecto. En este modelo las entregas se dividen en incrementos o compilaciones que agregan nuevas funcionalidades al producto. El incremento inicial contendrá la infraestructura requerida para soportar la funcionalidad de construcción inicial y los incrementos posteriores necesitarán pruebas para la nueva funcionalidad, pruebas de regresión de la funcionalidad existente y pruebas de integración de partes nuevas y existentes. Este ciclo de vida puede brindar una presencia temprana en el mercado con una funcionalidad crítica, es más simple de administrar porque la carga de trabajo se divide en partes más pequeñas y reduce la inversión inicial; sin embargo es más costoso a largo plazo. Los modelos iterativos son:

  - **Modelo desarrollo rápido de aplicaciones**: En este modelo las funcionalidades se desarrollan en paralelo basado en mini-proyectos, los desarrollos se estiman, se entregan y luego se ensamblan en un prototipo funcional. Esta metodología permite la validación temprana de los riesgos tecnológicos y una respuesta rápida a los requisitos cambiantes del cliente y fomenta la retroalimentación activa de los clientes.

    El cliente obtiene una visibilidad temprana del producto y puede proporcionar comentarios sobre el diseño y así mismo decidir de acuerdo a la funcionalidad existente, si continuar con el desarrollo, las funcionalidades a incluir en el próximo ciclo de entrega o incluso detener el proyecto si está no entregando el valor esperado.

- **Modelos de desarrollo ágil**: Extreme Programming (XP) es actualmente uno de los modelos de ciclo de vida de desarrollo ágil más conocidos y promueve:
  - La generación de historias de negocios para definir la funcionalidad.
  - Feedback continuo del cliente dado que exige a un cliente recibir comentarios continuos, definir y llevar a cabo pruebas de aceptación funcional.
  - La programación de pares y la propiedad de código compartido entre los desarrolladores.
  - Los scripts de prueba de componentes deben escribirse antes de que se escriba el código.
  - Automatización de pruebas.
  - La integración y las pruebas del código se realizarán varias veces al día.
  - Debe implementar la solución más simple para resolver los problemas.

  A medida que se realiza un cambio en el código, se prueba el componente y luego se integra con el código existente, que luego se prueba con el conjunto completo de casos de prueba y proporciona una integración continua dado que los cambios se incorporan continuamente en la compilación del software.

La ejecución de las **Pruebas de Software** es fundamental para garantizar la correcta funcionalidad e interacción con el usuario final y **asegurar la calidad del software**. En la Figura **2-3** se puede observar de forma resumida la importancia, objetivos, tipos, niveles, técnicas y procesos en el diseño, desarrollo e implementación de pruebas, y de igual manera con más detalle a continuación:

2.3. Objetivos de las pruebas

Los principales objetivos de la implementación de pruebas durante el ciclo de desarrollo de software son:

- **Reducir el riesgo** en cuanto a la generación de complicaciones durante la operación del software y garantizar que el software durante su operación tenga el comportamiento adecuado y no genere problemas que puedan afectar la interacción del usuario final con el mismo.

- **Garantizar la calidad del software** y tener la certeza de que el usuario estará a gusto con el producto final, dado su correcto funcionamiento, interacción, procesos y respuestas esperadas del usuario respecto al mismo, además, que se garantiza que el software esté cumpliendo con el objetivo para el que fue creado.

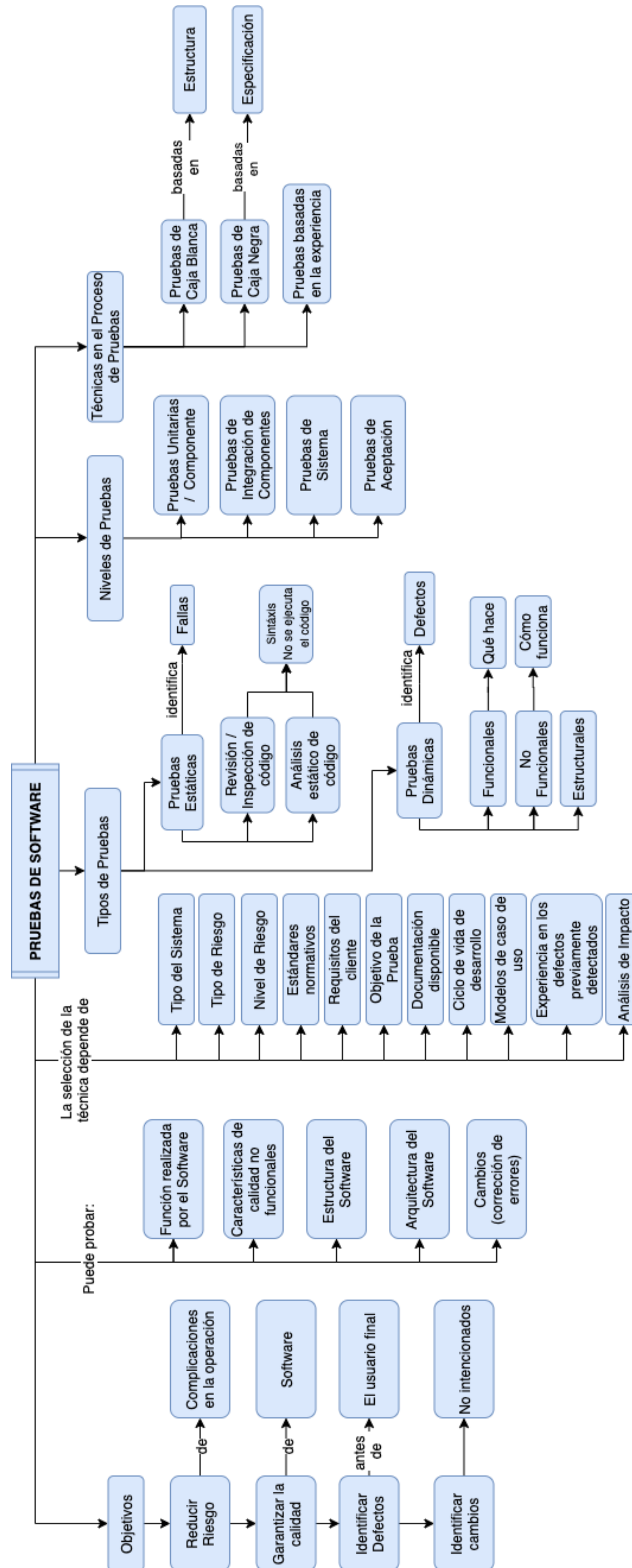


Figura 2-3.: Aseguramiento de la calidad del software: Pruebas en el desarrollo software.

- **Identificar defectos** presentados en el software durante su ciclo de desarrollo, antes de ser llevado al usuario final, dado el error humano, es necesario llevar a cabo un proceso que permita interactuar con el software antes de ser expuesto al cliente para realizar las verificaciones pertinentes en cuanto a su correcto comportamiento y cumplimiento de criterios.
- **Identificar cambios no intencionados**, que pueden presentarse al momento de implementar mejoras o ajustes dentro de un software ya creado y expuesto al usuario. Por el error humano pueden presentarse cambios que alteren el comportamiento adecuado del software, por tanto es necesario probar no solamente las nuevas funcionalidades del mismo sino garantizar que las previas funcionalidades sigan cumpliendo con sus objetivos, criterios y correcto comportamiento.

Por tanto, durante el proceso de pruebas, se puede probar:

- Función realizada por el Software.
- Características de calidad no funcionales.
- Estructura del software.
- Arquitectura del software.
- Cambios (corrección de errores).

De esta manera, dependiendo de los objetivos del software, a la hora de planear un proceso de pruebas se tienen en cuenta los siguientes objetivos [4]:

1. **Conformidad funcional básica:** Se debe:
   - Realizar casos de prueba donde se modele el uso típico y normal del sistema.
   - Validar las funcionalidades básicas, más importantes y críticas del software.
   - Validar la cobertura funcional de los casos normales en cuanto a las especificaciones.
   - Corregir los errores más importantes teniendo en cuenta los criterios de usuario.
2. **Estabilidad**: Se comprueba que el software mantenga su comportamiento frente a cambios de:
   - Contexto: Hardware, drivers, sistemas operativos, navegadores, versionamientos, protocolos de red, RAM, entre otros.
   - Configuración o parámetros.

- Forma de Ejecución, orden y duración.

3. **Robustez**: Validar el comportamiento del software frente un contexto hostil como:
    - Errores de datos de entrada.
    - Invocaciones incorrectas o fuera de contexto.
    - Recepción de códigos de error no previstos.
    - Recepción de respuestas erróneas.
    - No recepción de respuestas a invocaciones externas.
    - Loops.
    - Interrupción de transacciones.
    - Caída del software, sistema operativo, servidores, entre otros.
4. **Disponibilidad**: Validar que el software esté disponible cuando se requiere su uso.
5. **Rendimiento**: Validar la velocidad en cuanto a la respuestas del software y su interacción con el usuario.

2.4. Proceso de pruebas

Una ejecución correcta de pruebas, requiere de la elaboración de un proceso, que permita establecer los requerimientos, metodologías, pautas, condiciones, resultados e información esperada, entre otros, dicho proceso de pruebas de desarrollo de software se muestra en la Figura **2-4** y se compone de las siguientes etapas:

2.4.1. Planificación y estimación

La **planificación** consiste en definir los objetivos de las pruebas y especificar las actividades de las pruebas a realizar. Las principales actividades que se llevan a cabo en la planeación son [3]:

- Verificar la misión de las pruebas.
- Definir los objetivos de la pruebas.
- Determinar el alcance y los riesgos.
- Definir el enfoque global de las pruebas.
- Definir los niveles de las pruebas.
- Definir los criterios de entrada y salida.

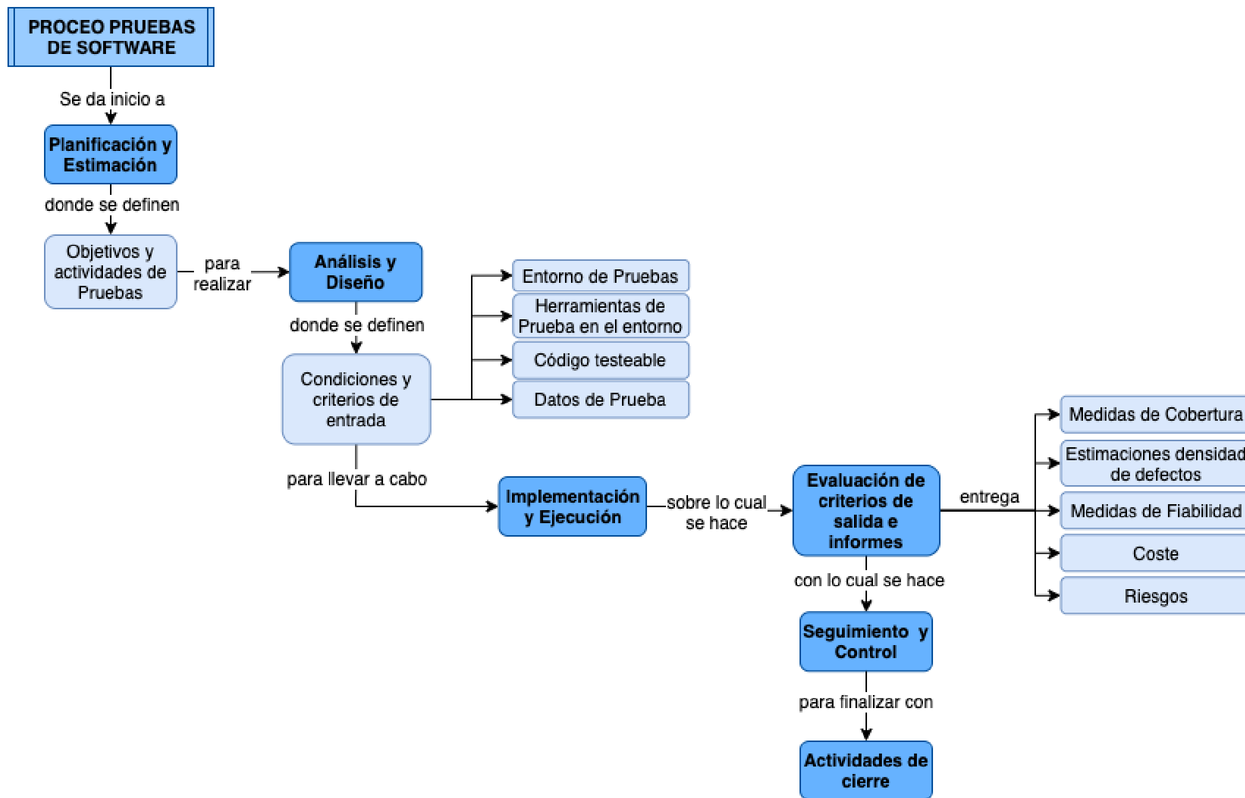


Figura 2-4.: Proceso de pruebas en el desarrollo de software.

- Integrar y coordinar las actividades de las pruebas dentro de las actividades del ciclo de vida del software (adquisición, suministro, desarrollo, operación y mantenimiento).
- Adoptar decisiones sobre: qué probar, quien llevará a cabo las actividades de pruebas. cómo se realizarán las actividades de pruebas y cómo se evaluarán los resultados.
- Programar las actividades de análisis y diseño de pruebas.
- Programar la implementación, ejecución y evaluación de las pruebas.
- Asignar los recursos para las actividades definidas.
- Definir la cantidades, el nivel de detalle, la estructura y plantillas para la documentación de las pruebas.
- Seleccionar las métricas para el seguimiento y control de la preparación y ejecución de las pruebas, la resolución de defectos y temas de riesgos.
- Establecer el nivel de detalle de los procedimientos de prueba para dar soporte a la preparación y ejecución de las mismas.

- Tener en cuenta las lecciones aprendidas.

Los **criterios de entrada**: Definen el inicio de las pruebas y deben cubrir lo siguiente:

- Disponibilidad y disposición del entorno de pruebas.
- Disposición de las herramientas de prueba en el entorno de pruebas.
- Disponibilidad de código testeable.
- Disponibilidad de datos de prueba

La **estimación** de las pruebas puede llevarse a cabo teniendo en cuenta dos enfoques [3]:

- Enfoque basado en métricas: La estimación se realiza a partir de métricas de proyectos anteriores o similares.
- Enfoque basado en expertos: La estimación se realiza en base a las realizadas por el propietario de las tareas, o se lleva a cabo por expertos.

El esfuerzo de la prueba puede depender de varios factores:

- Características del Producto: Calidad, especificación, tamaño, complejidad, requisitos de fiabilidad, seguridad y documentación.
- Características del Proceso de Desarrollo: Estabilidad del equipo, herramientas utilizadas, proceso de pruebas y habilidades del equipo.
- Resultado de pruebas: Número de defectos y cambios requeridos.

2.4.2. Análisis y diseño

Transformación de los objetivos de prueba en condiciones y casos de prueba. Consta de las siguientes tareas [3]:

- Revisar la base de pruebas (requisitos), los informes de riesgo, la arquitectura, diseño y especificaciones [3].
- Evaluar la testeabilidad de la base de pruebas y los objetivos de la prueba.
- Identificar y priorizar las condiciones de prueba en base al análisis de los elementos de prueba, especificación, comportamiento y estructura del software.
- Diseñar y priorizar los casos de prueba de alto nivel.

- Identificar los datos de prueba que soporten las condiciones y casos de prueba.
- Diseñar la configuración del entorno de pruebas e identificar cualquier infraestructura de herramientas requerida.
- Crear una trazabilidad bidireccional entre la base de pruebas y los casos de prueba.
- Determinar qué pruebas deben automatizarse y cuáles no.

2.4.3. Implementación y ejecución

Configuración del entorno de pruebas para la ejecución de las mismas mediantes la implementación de los casos de prueba. Consta de las siguientes tareas [3]:

- Preparación y verificación del entorno y datos de prueba.
- Programación de los procedimientos combinando y construyendo las series de pruebas necesarias.
- Finalizar, implementar y priorizar los casos de prueba.
- Desarrollar y priorizar los procedimientos de prueba, crear datos de prueba.
- Automatizar las pruebas definidas.
- Crear juegos de pruebas a partir de los procesos de pruebas definidos
- Verificar y actualizar la trazabilidad bidireccional entre la base de datos de prueba y los casos de prueba.
- Ejecutar los procedimientos de prueba manualmente o por medio de herramientas de prueba.
- Registrar los resultados de la ejecución de las pruebas y registrar las versiones del software probado, herramientas y productos de soporte de pruebas.
- Comparar los resultados reales con los esperados.
- Reportar las discrepancias en forma de incidencias y analizarlas para establecer las causas.
- Repetir las actividades de pruebas para verificar las discrepancias encontradas, de tal manera que se verifique la solución de las mismas sin que se afecten otras partes del software.
- Ejecutar las pruebas de forma ascendente (iniciar con las más pequeñas).

- Producir diarios de pruebas, de incidentes y reportes.
- Obtener aceptaciones o alcanzar los criterios de salida antes de pasar al próximo nivel de pruebas

2.4.4. Evaluación de criterios de salida e informes

Se evalúan los objetivos definidos contra los resultados obtenidos en la ejecución de las pruebas, sus principales tareas son [3]:

- Comprobar los registros de las pruebas con los criterios de salida previstos en la planificación de la prueba.
- Evaluar si los criterios de salida han sido logrados para cada nivel de prueba.
- Evaluar si se requieren más pruebas o si deberían modificarse los criterios de salida especificados.
- Preparar reportes de síntesis de pruebas realizadas.

Los **Criterios de Salida** definen cuando terminan las pruebas y cubren lo siguiente:

- Medidas de cobertura de código, funcionalidad y riesgo.
- Estimaciones de densidad de defectos o medidas de fiabilidad.
- Costo.
- Riesgos residuales, como ausencia de cobertura o defectos no corregidos.
- Calendarios para la entrega del software al usuario final.

2.4.5. Seguimiento y control

Dentro del proceso de pruebas es necesario realizar **Seguimiento** del progreso real con el plan previsto, informar sobre el estado de las pruebas y complicaciones. Las métricas más comunes dentro de este proceso incluye [3]:

- Porcentaje de trabajo realizado durante la preparación del caso de prueba.
- Porcentaje de casos de prueba planificados y preparados.
- Porcentaje de trabajo realizado durante la preparación del entorno de pruebas.
- Número de casos de prueba ejecutados/no ejecutas y superados/fallidos.

- Densidad de defectos detectados/corregidos, frecuencias de fallo y resultados de la repetición de las pruebas.
- Cobertura de pruebas de los requisitos, riesgos o código.
- Coste de las pruebas.

Sus principales actividades son:

- Comparar el avance real con el planificado.
- Reportar el estatus y las desviaciones con lo planeado.
- Evaluación de nivel de riesgo alcanzado.

El **Control** permite tomar decisiones para ejercer una acción orientada o correctiva a partir de la información y métricas de los informes generados e incluye las siguientes actividades[3]:

- Tomar decisiones en base a la información obtenida del seguimiento de las pruebas.
- Restablecer la prioridad de las pruebas si sucede algún riesgo.
- Cambiar el calendario de las pruebas por motivos de disponibilidad/no disponibilidad de un entorno de pruebas.
- Establecer un criterio de entrada que requiere la repetición de las pruebas de las correcciones por parte del desarrollador antes de aceptarlas en una construcción.

2.4.6. Actividades de cierre

Recopilación de los resultados obtenidos y observaciones durante el proceso de pruebas. Permite dar paso al despliegue o lanzamiento del software, finalización de versiones, entre otros. Sus principales actividades son [3]:

- Recolectar la información de las pruebas realizadas.
- Consolidar y sintetizar la información obtenida, para sacar conclusiones.
- Comprobar cuáles productos entregables previstos han sido efectivamente entregados.
- Cerrar los informes de incidencias o aportar modificaciones a aquellos que siguen abiertos.
- Documentar la aceptación del sistema.
- Finalizar y archivar los productos de soporte, entorno e infraestructura de pruebas para su posterior uso.

- Entregar los productos de soporte de pruebas para su posterior mantenimiento.
- Analizar los resultados y deducir lecciones aprendidas para aplicar en proyectos futuros (documentar).

2.5. Enfoque de las pruebas

Un proceso de pruebas puede ser ejecutado por diferentes motivos, dependiendo de los objetivos del software, ajustes y mantenimiento del mismo, funcionalidades, entre otros. Los enfoques para las pruebas son [3]:

- Enfoque analítico: Pruebas basadas en riesgo.
- Enfoque basado en modelos: Pruebas estocásticas que utilizan información estadística (frecuencia de fallos, uso, entre otros).
- Enfoque metódico: Basados en fallos, experiencia, listas de comprobación y características de calidad.
- Enfoque de Proceso: Especificados en las normas o metodologías ágiles.
- Enfoque dinámico y heurístico: Pruebas exploratorias reactivas a eventos.
- Enfoque consultivo: Basadas en recomendaciones y orientaciones de los expertos.
- Enfoque anti-regresión: Reutilización de material de pruebas.

2.6. Técnicas de diseño de pruebas

Con base en los objetivos y evaluación de los riesgos del sistema, se da inicio a la planificación de las pruebas, se seleccionan las técnicas de diseño, tipos de pruebas a aplicar y se definen los criterios de entrada y salida. La selección de la técnica depende de:

- Tipo del sistema.
- Tipo de riesgo.
- Nivel de riesgo.
- Estándares normativos.
- Requisitos del cliente.
- Objetivo de la prueba.

- Documentación disponible.
- Ciclo de vida de desarrollo.
- Modelos de caso de uso.
- Experiencia en los defectos previamente detectados.
- Análisis de impacto: Cuántas pruebas de regresión son necesarias, a partir de la determinación, de cómo el sistema actual se ve afectado por los cambios realizados.

De esta manera, se tiene que las técnicas de diseño de pruebas son:

2.6.1. Pruebas de Caja Blanca basadas en la estructura

Se basan en el análisis de la estructura interna del sistema; por tanto, los casos de prueba se seleccionan a partir del mismo código, información de diseño de software, información técnica de la implementación, arquitectura, entre otros.

2.6.2. Pruebas de Caja Negra basadas en la especificación

Se seleccionan los casos de prueba a partir de unos criterios de usuario u objetivos de software previamente definidos.

- **Partición de equivalencia**: Los valores de entrada y salida se agrupan de tal manera que tengan un comportamiento similar, tanto para datos válidos como no válidos. Puede indicar cobertura de entrada y salida.
  Permite detectar inconsistencias en las especificaciones y minimiza los casos de prueba; sin embargo, no tiene en cuenta las posibles interacciones entre las condiciones de prueba y no resuelve problemas de estabilidad del software.

  Generación de casos de prueba [4]:

  1. Análisis de especificaciones.
  2. Identificación de comportamientos de la funcionalidad a ser probada, teniendo en cuenta entradas y salidas.
  3. Identificar las variables de entrada pertinentes.
  4. Analizar subconjunto de valores con el mismo comportamiento para cada variable.
  5. Generar combinaciones de las clases de equivalencia.

- **Análisis de valores límite**: Valores máximos y mínimos de una partición válida/no válida que permiten determinar los defectos en el software.

- **Tablas de decisión**: Permite la identificación de las condiciones lógicas (entradas)y acciones (salidas) del sistema a partir de los requisitos del mismo, con el cual se establecen reglas por medio de la combinación de dichas condiciones.
- **Transición de estado**: El diagrama de transición de estados permite definir todos los estados posibles del software y las interacciones o transiciones correctas/incorrectas entre dichos estados. Un **estado** representa una situación estable de un sistema hasta que sea modificado por un evento. Un **evento** representa una situación exterior que puede ocurrir cuando el sistema se encuentra en determinado estado.
- **Pruebas de caso de uso**: Describe los flujos del proceso o las interacciones entre los usuarios del sistema y el sistema, para producir un resultado específico que tiene valor para el usuario. Tienen unas precondiciones necesarias para su correcto funcionamiento, unas post-condiciones que evalúan que los resultados esperados sean los obtenidos. Por lo general se basan en un escenario más probable y otras alternativas.

2.6.3. Pruebas basadas en la experiencia

Las pruebas se derivan de la habilidad e intuición del probador, de su experiencia con las aplicaciones, herramientas similares, errores previos, entre otros; se elabora una lista con las posibles fallas, las cuales deben ser atacadas por medio de la implementación de las pruebas adecuadas.

2.7. Tipos de pruebas

Dentro del proceso de pruebas, existen dos tipos de pruebas que deben ser ejecutadas siempre en todo tipo de software, estas pruebas se enfocan en: el código desarrollado como para las pruebas estáticas y en la funcionalidad del mismo para las pruebas dinámicas, a continuación se explica con más detalle:

2.7.1. Pruebas Estáticas

Permiten identificar fallas en el sistema, por medio de la verificación de la sintaxis del código desarrollado, por tanto no se ejecuta el código y se basan en:

- Revisión / Inspección de código.
- Análisis estático de código.

2.7.2. Pruebas Dinámicas

Permiten identificar defectos en el sistema. Se basa en las siguientes pruebas:

- **Pruebas funcionales**: Valida que el software cumpla su función y que de acuerdo a los requerimientos de usuario las funcionalidades se lleven a cabo como fueron planteadas en cuanto al qué, cómo, cuándo y dónde.
- **Pruebas no funcionales**: Valida el comportamiento del software, el cómo funciona. Dentro de las pruebas no funcionales se encuentran:
  - Usabilidad
  - Seguridad
  - Rendimiento
  - Estabilidad
  - Robustez
  - Disponibilidad
  - Portabilidad
  - Integridad
  - Eficiencia
- **Pruebas de estructura/arquitectura**: Mide exhaustividad de las pruebas mediante la evaluación de cobertura (Pruebas de Caja Blanca).
- **Pruebas de regresión**: son las pruebas asociadas a cambios y se caracterizan por se pruebas reiteradas de un programa ya probado.

2.8. Niveles de pruebas

Dependiendo de los objetivos de las pruebas, los requerimientos del software, funcionalidades, enfoque y técnicas a implementar en dicho proceso, se pueden ejecutar diferentes niveles de prueba dentro del proceso pruebas:

2.8.1. Pruebas de Componente

Son pruebas unitarias cuyo objetivo es comprobar el funcionamiento correcto y localizar los defectos de los módulos de software tales como clases, métodos, objetos, entre otros.

Base de Pruebas [3]:

- Requisitos de componente.

- Diseño de detalle.
- Código.

Objetos de prueba típicos [3]:

- Componentes.
- Programas.
- Conversión de datos.
- Programas de Migración.

2.8.2. Pruebas de Integración

Su objetivo es comprobar la interacción correcta entre diferentes componentes del sistema o interfaces (hardware/software).

Base de Pruebas [3]:

- Diseño de software y sistema.
- Arquitectura.
- Flujos de trabajo.
- Casos de uso.

Objetos de prueba típicos [3]:

- Implementación de bases de datos de subsistemas.
- Infraestructura.
- Interfaces.

2.8.3. Pruebas de Sistema

Su objetivo es comprobar el comportamiento adecuado de todo el sistema de software a nivel del producto; es decir, que el entorno de pruebas debe ser lo más parecido al entono de producción (del usuario final), para minimizar el riesgo de fallas y defectos.

Por lo general se prueba el comportamiento del software antes diferentes sistemas operativos, recursos del sistema, requisitos funcionales y no funcionales, calidad de los datos, entre otros.

Base de Pruebas [3]:

- Especificación de requisitos del sistema y software.
- Casos de uso.
- Especificaciones funcionales.
- Informes de análisis de riesgos.

Objetos de prueba típicos [3]:

- Manuales del sistema.
- Configuración del sistema.

2.8.4. Pruebas de Aceptación

Su objetivo es probar una funcionalidad específica del software que permita definir la buena disposición del sistema para su posterior despliegue y uso.

Base de Pruebas [3]:

- Requisitos de usuario.
- Requisitos del sistema.
- Casos de uso.
- Procesos de negocio.
- Informes de análisis de riesgos.

Objetos de prueba típicos [3]:

- Procesos de negocio en un sistema completamente integrado.
- Procesos operativos y de mantenimiento.
- Procedimientos de usuario.
- Formularios.
- Informes.

El aseguramiento de la calidad del software es un proceso detallado que requiere de una metodología adecuada que permita la implementación de procesos de pruebas que se ajusten a las necesidades del usuario. Por tanto, incluir técnicas y herramientas que permitan ejecutar de forma más rápida y continua estos procesos reduciendo la inducción de errores humanos, como la **automatización de pruebas de software** dentro de estos procesos de aseguramiento de calidad es de vital importancia.

2.9. Automatización de pruebas de software

En primer lugar se debe tener claro que es una prueba automatizada y automática. Las pruebas automáticas son «“pruebas de software que no requieren de intervención humana alguna para su generación, ejecución, evaluación y reportez son autónomas en todas sus dimensiones. Mientras que en las pruebas automatizadas a pesar de utilizar una máquina si hay intervención humana en cualquiera de las dimensiones» [10].

Por tanto, la automatización de pruebas permite validar el software en entornos de prueba realistas en cualquier momento y lugar, y al ser un proceso autónomo permite la retroalimentación inmediata de los resultados de las pruebas donde se indique si un software cumple o no con los criterios de calidad.

Como las pruebas son la mayor razón de los retrasos del despliegue de software al cliente es importante optimizar el esfuerzo que requieren las pruebas y su implementación, para lograr entregar software de calidad en menor tiempo y de manera continua. Por tanto, es importante implementar un proceso de automatización que permita librar a los probadores de tareas manuales repetitivas para que puedan enfocarse y dedicar más tiempo en explorar el software y encontrar errores que un robot no puede encontrar.

Este proceso de pruebas adicionalmente permite la ejecución continua de las pruebas en cualquier entorno y en cualquier momento según como se configure. De esta manera, la automatización permite llevar a cabo una buena práctica del desarrollo de pruebas denominada Pruebas Continuas (Continuous Testing [5]) que se muestra en la Figura **2-5**.

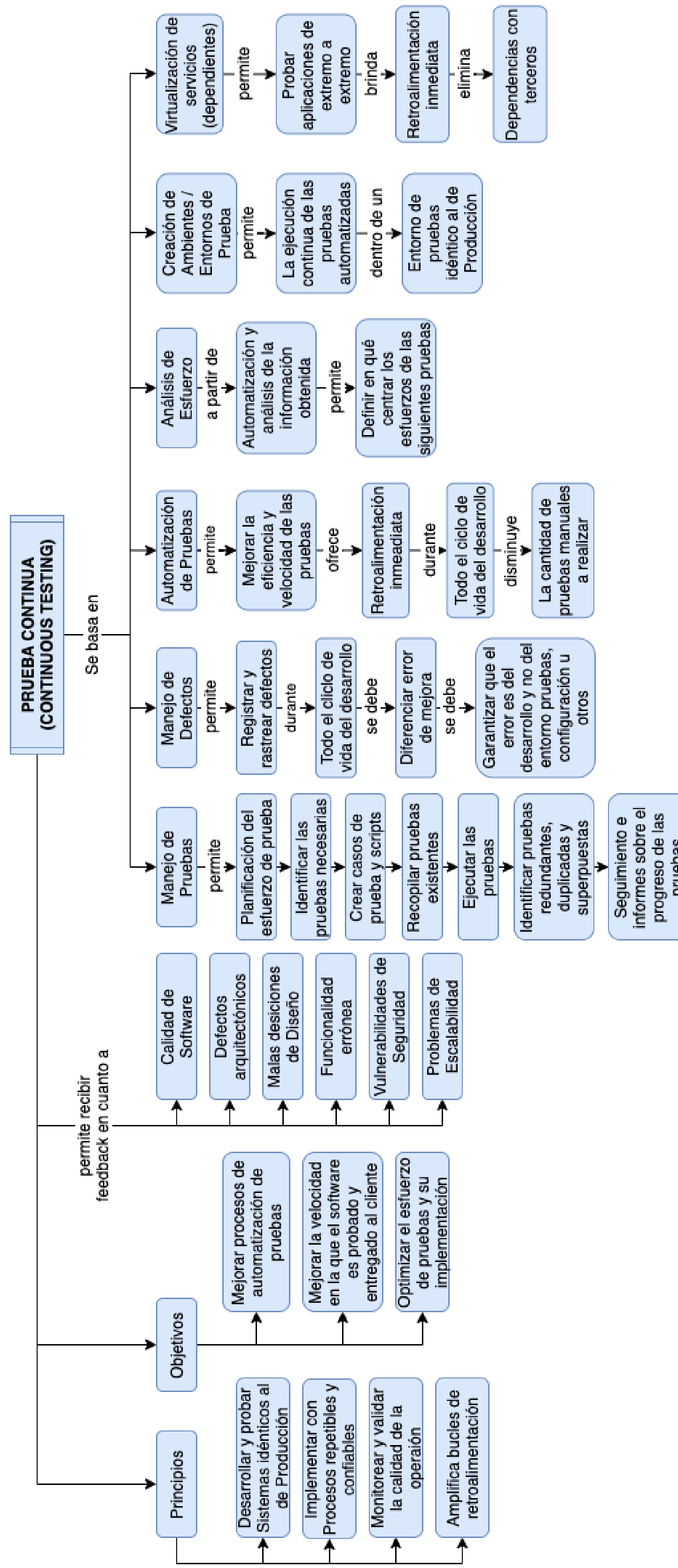


Figura 2-5.: Pruebas continuas de software.

Este proceso implica:

- Garantizar la cobertura de las pruebas en los puntos de contacto (APIs) en toda la aplicación.
- Validar el correcto comportamiento del software dadas sus dependencias con múltiples sistemas.
- Garantizar el cumplimiento de los escenarios de negocio de extremo a extremo.
- Certificar la escalabilidad del software y los procesos del negocio.

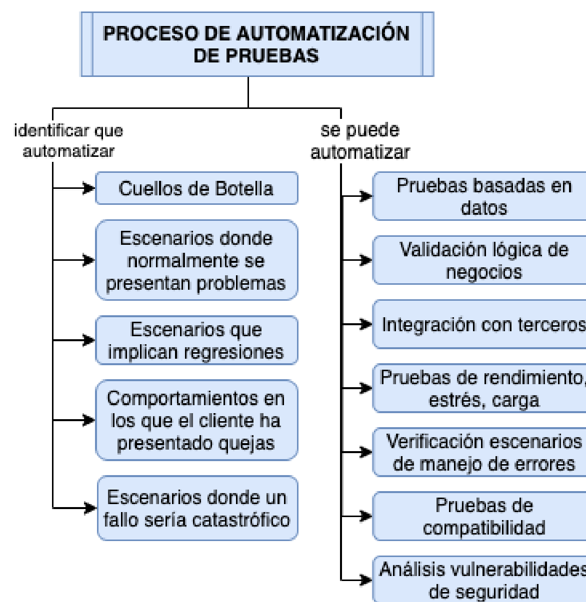


Figura 2-6.: Automatización de pruebas.

Sin embargo, es necesario identificar qué se debe automatizar en este proceso de pruebas continuas. Como se observa en la Figura 2-6, se debe automatizar un conjunto adecuado de pruebas para mantener las pruebas de regresión de tal forma que se pueda garantizar que un cambio no afecte la funcionalidad y además:

- Pruebas basadas en datos complejos para cubrir aspectos críticos donde una prueba manual podría tardar demasiado dada la complejidad de los datos.
- Validación lógica de negocios, es decir la correcta funcionalidad de la aplicación es sus aspectos más críticos.
- Integración con terceros para garantizar la completa y correcta integración de todo el sistema.

- Pruebas de Rendimiento, Carga, Estrés, Vulnerabilidad, entre otros, que permitan verificar la calidad del software en cuanto a su comportamiento no funcional.
- Verificación de escenarios de manejo de errores para garantizar el correcto comportamiento de la aplicación cuando se presente algún error.
- Pruebas de compatibilidad para comprobar el correcto funcionamiento de la aplicación en diferentes plataformas, sistemas operativos, entornos, etc.
- Escenarios en los que normalmente se encuentra errores basados en la experiencia de los probadores, para garantizar que el evento no se repita.
- Escenarios donde una falla sería catastrófico, para garantizar que los errores críticos no se presenten en la aplicación.
- Escenarios donde los cliente se han quejado, para garantizar que el evento no se repita.
- Pruebas que toman mucho tiempo de ser ejecutadas manualmente dada su complejidad.

Un escenario típico aplicado en pruebas continuas es el siguiente ([5]):

1. Crear una nueva compilación, donde se ejecutan las pruebas unitarias.
2. Instalar el entorno de pruebas, donde será desplegada la nueva compilación.
3. Iniciar los Stubs de los servicios (terceros) requeridos.
4. Ejecutar las pruebas de integración y rendimiento.
5. Creación del ambiente de pruebas a partir de la generación de los artefactos requeridos para su despliegue y posteriormente se realiza el despliegue del ambiente.
6. Ejecución de las pruebas de interfaz de usuario en el entorno de pruebas desplegado.
7. Capturar resultados de las pruebas y resolver defectos para a continuación generar una nueva compilación cuando sea requerido.

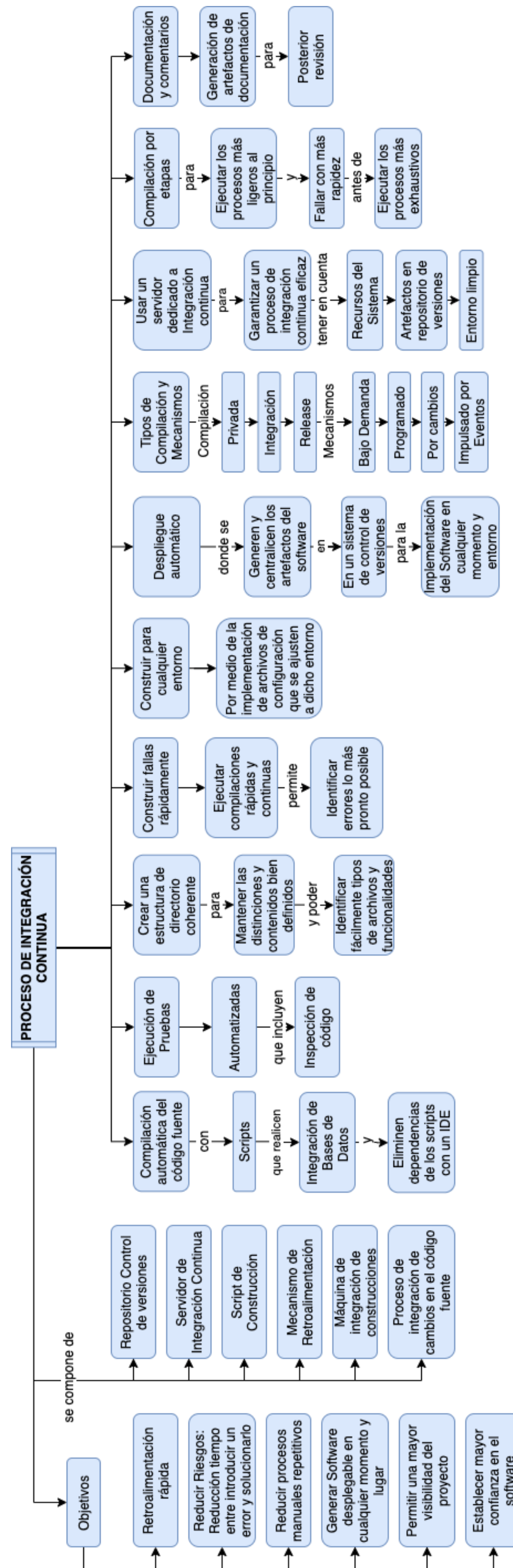


Figura 2-7.: Integración Continua dentro del ciclo de desarrollo de software.

Por tanto, para la automatización de un sistema de pruebas es necesario comprender cómo un sistema automatizado puede ser integrado, implementado y ejecutado, dentro de los procesos del software para el que fue creado. De esta manera, llevar a cabo la automatización de un sistema de pruebas dentro de un entorno de **integración continua** permite la automatización de principio a fin del ciclo de desarrollo de un software como se muestra a continuación:

2.10. Integración Continua

El Proceso de Integración Continua permite la integración del desarrollo de software con el proceso de pruebas de forma inmediata en cualquier entorno y momento como se muestra en la Figura **2-7** de forma resumida y se compone de ([6]):

- Repositorio de control de versiones donde se encuentra siempre disponible el código fuente, allí se gestionan los cambios en el mismo y los artefactos de software.
- Servidor de Integración Continua donde se ejecuta una compilación de integración cada vez que se confirma un cambio en el repositorio de control de versiones y publica los resultados de la ejecución.
- Script de compilación o conjunto de scripts, que permiten compilar, probar, inspeccionar e implementar el software.
- Mecanismos de retroalimentación donde se generan los reportes/comentarios para cada compilación de tal forma que permita hacer seguimiento del estado de la misma.
- Máquina de integración de construcciones donde se realiza la integración completa del software.

Un proceso de Integración continua permite llevar a cabo ([6]):

1. Compilación automática del código fuente mediante la ejecución de un script que permite crear código ejecutable en una máquina (binarios), a partir de una fuente legible para los humanos desarrollada con diferentes lenguajes de programación. Dicho Script permitirá adicionalmente la integración del código fuente con bases de datos cuyo código estará almacenado en un repositorio de control versiones y no debe depender de ningún IDE para su ejecución en cualquier máquina, entorno. etc.
2. Ejecución de pruebas automatizadas como pruebas unitarias, componente, integración, sistema, carga, rendimiento, seguridad, entre otros, que permitan detectar fallas tempranas. Así mismo, se implementa la inspección automática de código estático que permite mejorar la calidad del código mediante la aplicación de reglas de acuerdo a los estándares de vinificación y métricas de calidad.

3. Crear una estructura de directorio coherente para mantener las distinciones y contenidos definidos consistentemente. El directorio está definido de tal manera que se puede identificar fácilmente los diferente tipos de archivo de acuerdo a su función (compilación, requisitos, diseño, administración, despliegue, pruebas, configuración, herramientas, etc.) para facilitar su compilación y ejecución.

4. Construir fallas rápidamente al ejecutar compilaciones continuas permite identificar errores inmediatamente para poderlos solucionar tan pronto como sea posible.

5. Construir para cualquier entorno por medio de la implementación de archivos de configuración que permitan identificar el entorno y ajustar automáticamente las variables de entorno (servidores, bases de datos, servicios, etc.).

6. Despliegue automático para la implementación del software en cualquier momento y lugar, mediante la generación y centralización de los artefactos (librerías, archivos de datos, archivos de configuración, scripts de instalación, etc.) en un sistema de control de versiones, para asegurar una correcta compilación y ejecución en cualquier máquina.

7. Tipos de Compilación y Mecanismos.
   - Tipos de Compilación
     - Privada: Antes de ser enviada al repositorio de control de versiones se realiza una compilación localmente, así se integran los cambios locales con los últimos cambios disponibles en el repositorio. Los pasos para ejecutar una compilación privada son ([6]):
       a) Consultar el código que se altera desde el repositorio.
       b) Hacer cambios al código.
       c) Obtener los últimos cambios del repositorio.
       d) Ejecutar una compilación que incluya la ejecución de todas sus pruebas unitarias.
       e) Confirmar los cambios del código en el repositorio.
     - Integración: Su objetivo es integrar otras ramas con la rama principal.
     - Release: Su objetivo es producir una versión de lanzamiento para el usuario.

   - Tipos de Mecanismos
     - Bajo Demanda: Impulsado por el usuario manualmente.
     - Programado: Para ser ejecutado en un momento en específico (una hora).
     - Por cambios: Ejecutado al detectar cambios en el repositorio de código cada cierto tiempo.

- Impulsado por eventos: Cuando el repositorio de control de versiones detecta un cambio, inmediatamente inicia el script de compilación.

8. Usar un servidor dedicado a la integración continua para garantizar un proceso de integración continua correcto y eficiente, en el cual se debe tener en cuenta:
   - Recursos del Sistema.
   - Todos los artefactos del software deben estar en el repositorio de control de versiones.
   - El entorno debe estar limpio, se deben eliminar cualquier dependencia de código, configuraciones previas, entre otros.

   Así mismo este servidor tiene las siguientes funciones:
   - Verificar cambios en el repositorio de control de versiones en un intervalo de tiempo especificado.
   - Soportar diferentes herramientas de creación de scripts.
   - Enviar correos electrónicos a las partes interesadas.
   - Mostrar un historial de compilaciones anteriores.
   - Mostrar un panel de control accesible desde la Web para revisión de la información de compilación de integración en cualquier momento.
9. Compilación por etapas para ejecutar los procesos más ligeros al principio y fallar con más rapidez antes de ejecutar los procesos más exhaustivos, así mismo, para llevar un seguimiento de los procesos requeridos para la compilación y distinguir los diferentes pasos que se requieren para la misma.
10. Generación de documentación y comentarios para la posterior revisión de diagramas, diseño y demás.

Un ejemplo de dicho proceso de integración continua se puede observar en la Figura **2-8**, presentado en el seminario de «El futuro de las Pruebas Automáticas de Software en Colombia» en la Universidad de los Andes en Mayo de 2019 por la compañía GreenSQA; donde resaltan que la ejecución de las pruebas automatizadas se llevan a cabo dentro de un proceso de integración continua. En primer lugar un ingeniero de pruebas diseña las pruebas, las automatiza por medio de la interacción con frameworks[1] de prueba tales como selenium, el desarrollo (código) de esas pruebas es almacenado en un repositorio (Git), donde al hacerse cualquier cambio (push) en el mismo, los agentes de pruebas automáticas (máquinas propias o en la nube) ejecutan dichas pruebas automáticas.

[1] «Un framework es una combinación de prácticas y herramientas diseñadas para ayudar a los testers a probar de forma más eficiente» [11], Cristian Arandia de Endava

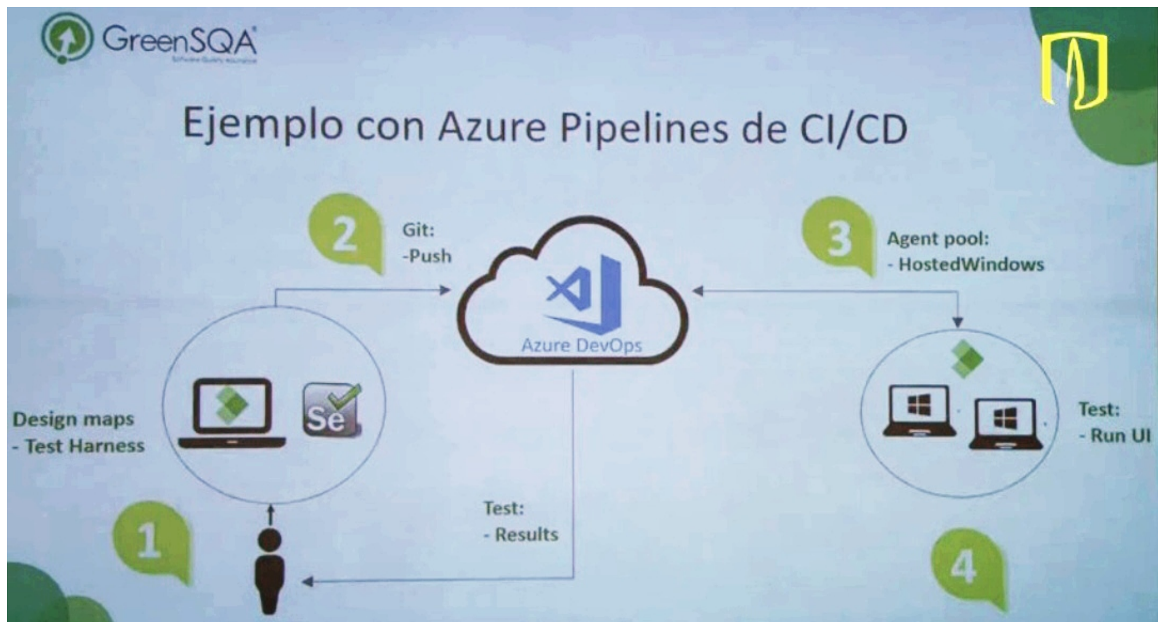


Figura 2-8.: Pipeline CI/CD GreenSQA [11].

En conclusión, seleccionar e incluir las metodologías, técnicas y tecnologías descritas en el presente capítulo dentro del ciclo de desarrollo de software dependiendo de las necesidades del mismo permiten la ejecución de un ciclo más eficiente donde se garantiza la calidad del software con una retroalimentación continua que da paso a la pronta detección y corrección de posibles problemas dentro del mismo. Adicionalmente, garantizan la entrega final a usuario de forma continua dado que la automatización de los procesos permite la ejecución de tareas de forma más rápida, reducción de tiempos de ejecución, generación continua de informes con información general y específica de las ejecuciones y garantizar la estabilidad y calidad del software.

3. Estado del Arte

Definir las metodologías, herramientas y técnicas que mejor se ajustan al ciclo de desarrollo de un software en específico requiere de un conocimiento previo del software y los procesos que este involucra dentro de su ciclo de desarrollo. Así mismo es importante conocer las alternativas que se implementan en la actualidad en otros ciclos de desarrollo. De esta manera en el presente capítulo se dan a conocer:

- Las metodologías y procesos de automatización de pruebas en la actualidad: estrategias de pruebas, la pirámide de pruebas, estrategias de automatización, problemas en la automatización de pruebas y composición de un sistema automatizado de pruebas, que permitan definir la mejor estrategia a implementar de acuerdo a las necesidades del software de DetectID™.
- Software de DetectID™: se realiza una introducción a este software, sus objetivos, composición, funciones, entre otros.
- La metodología y proceso de desarrollo de software en DetectID™: con su respectivo ciclo de desarrollo de software actual a partir del cual se analiza e implementan las mejoras requeridas que permitan llevar a cabo un ciclo de desarrollo de dicho software dentro de un entorno de integración continua.

3.1. Metodologías/Procesos de automatización de pruebas en la actualidad

Es necesario conocer las experiencias previas que se han tenido en la implementación de metodologías de automatización de pruebas para mejorar los procesos. En este aspecto, es importante investigar las estrategias, dificultades, metodologías, composición, estructura, entre otros, que han sido llevadas a cabo en la evolución del software.

3.1.1. Estrategias de pruebas

Una pregunta a resolver para establecer la metodología o proceso de pruebas con mejor desempeño es "**¿Cómo debe ser el esfuerzo para los niveles de prueba?**", para dar solución a este interrogante Mario Linares y Camilo Escobar en [10] muestran diferentes estrategias de pruebas como se observa en la Figura **3-1** donde dichas estrategias dependiendo del punto de vista del esfuerzo o importancia que se le dará a cada nivel o tipo de prueba.

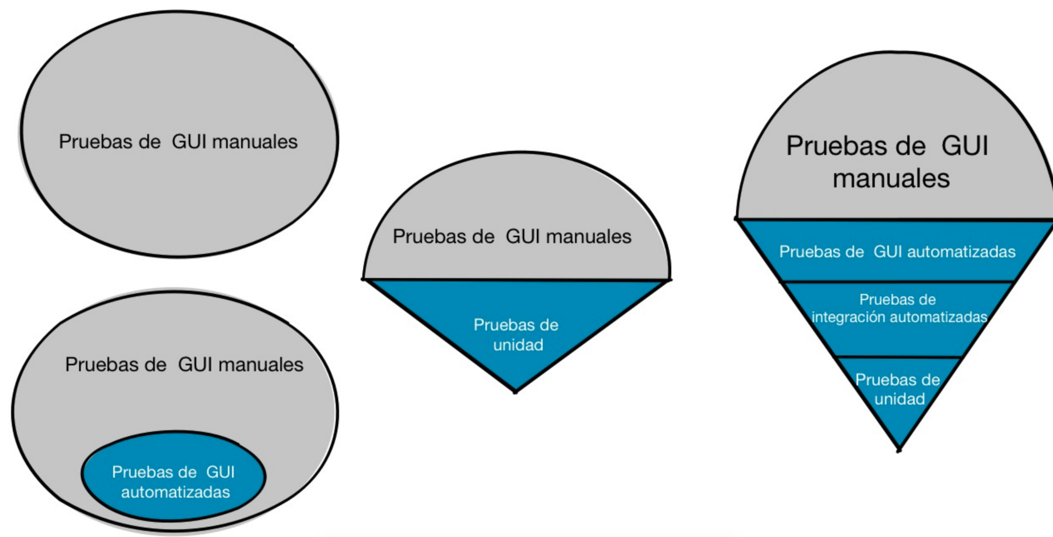


Figura 3-1.: Estrategias de pruebas [10].

La estrategia de la imagen arriba a la izquierda indica que la totalidad de las pruebas a ejecutar son manuales sobre la interfaz gráfica. Así mismo la imagen abajo a la izquierda con la adición de un pequeño subconjunto de pruebas de interfaz automatizadas. En la mitad se indica que se deben ejecutar pruebas manuales en la interfaz gráfica, pero adicionalmente pruebas de unidad. Finalmente la imagen de la derecha (cono de helado), sugiere un conjunto de pruebas más elaborado.

Esta estrategia de cono de helado «sugiere que las pruebas de unidad tienen el menor porcentaje en toda la estrategia y que la mayor cantidad del esfuerzo debe ser dedicada a pruebas (manuales o automatizadas sobre la GUI). ¿Cuál es el nivel de pruebas menos costoso de ejecutar y que proporciona detalles más exactos de dónde ocurre el error? LAS PRUEBAS DE UNIDAD!!. ¿Y cuáles son los niveles de pruebas más costosos? SISTEMA y ACEPTACIÓN. En esa medida, el esquema de cono de helado muestra un anti-patrón de estrategia muy común en las empresas que desaprovecha las pruebas automáticas y favorece las pruebas más costosas sobre las menos costosas y rápidas.» [10]

Por tanto, ¿Cuál sería la mejor estrategia que permita aprovechar al máximo las pruebas automáticas?. La respuesta es una Pirámide como se muestra en la Figura **3-2** la cual **3-2** «representa la “ideal software testing pyramid”propuesta por Alister Scott que sugiere que una estrategia de pruebas debe estar basada en su mayoría en pruebas automatizadas con mayor esfuerzo en el nivel de pruebas de unidad. Note que las pruebas manuales no desaparecen en la pirámide porque hay funcionalidades y flujos complejos de automatizar, donde es más barato realizar sesiones de pruebas manuales » [10]. Adicionalmente, que las pruebas exploratorias son necesarias y se hacen manualmente porque a partir de ellas se identifica el flujo completo y correcto en la ejecución de las pruebas para lograr automatizarlas de forma eficiente.

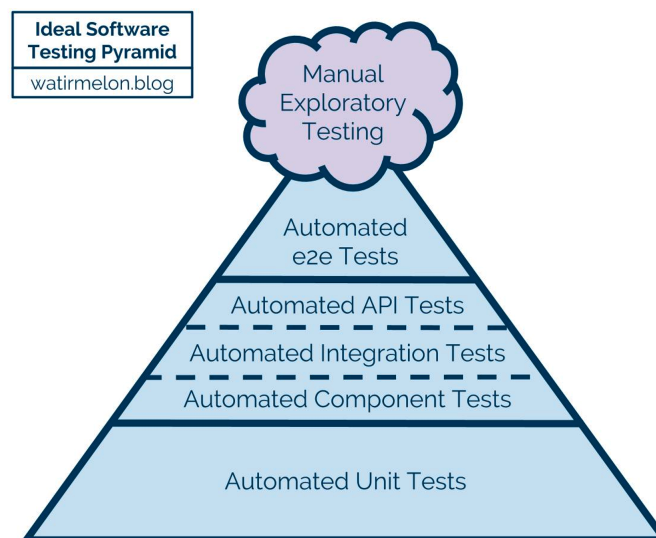


Figura 3-2.: Pirámide de pruebas de © Ideal [10].

En Mayo de 2019 en el seminario de «El futuro de las Pruebas Automáticas de Software en Colombia» en la Universidad de lo Andes, Cristian Arandia Senior Software Tester de la compañía Endava mencionó la estrategia de pruebas más común y usada en la actualidad denominada **BDD** (Behavior Driven Development), es una estrategia de desarrollo dirigido por comportamiento del sistema donde el principal actor es el usuario dada la interacción de éste con el sistema. La estructura de esta estrategia se puede observar en la Figura **3-3**. Con el **Given** se definen las precondiciones y se prepara el ambiente de pruebas, con el **When** se ejecutan las acciones que dentro de los requisitos el usuario ejecutaría dentro del sistema (son los pasos puntuales para obtener un resultado) y finalmente con el **Then** se analizan los resultados obtenidos de las acciones previamente ejecutadas.

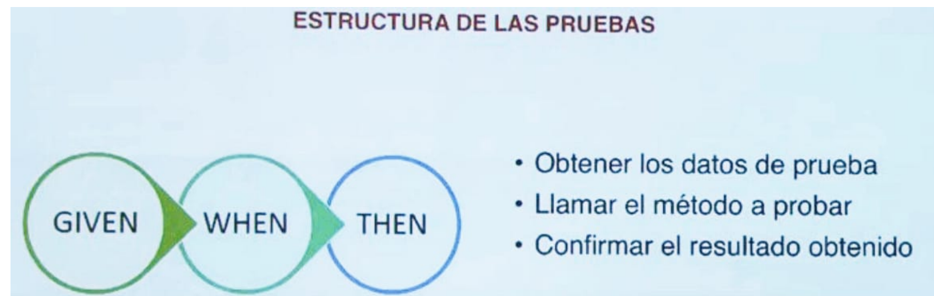


Figura 3-3.: Estructura de las pruebas en la estrategia de BDD [11].

3.1.2. Pirámide de pruebas

Muchas empresas a nivel mundial están implementando automatización en sus sistemas de calidad principalmente porque reduce los tiempos de ejecución de pruebas, sin embargo, se debe tener claro que se debe automatizar, cómo, con qué herramientas, dado que cada nivel de prueba tiene una determinada complejidad que puede hacer que la automatización termine siendo más costosa y en muchos casos el tiempo no se reduzca a grandes proporciones.

Aslak Hellesøy (creador de cucumber) en una conferencia en GOTO del 2019 [9] quien con la imagen de la Figura **3-4** muestra la Pirámide de pruebas reducida con respecto a la de la Figura **3-2**. Indicó cuales son las pruebas que deben tener mayor cobertura (pruebas unitarias en la base de la pirámide) y por tanto ejecutar la mayor parte de pruebas de la funcionalidad, seguido de las pruebas de los servicios y/o componentes que se integran en la funcionalidad, para finalizar con las pruebas de interfaz gráfica (en la punta de la pirámide) las cuales deben requerir menos esfuerzo y dada la cobertura de las pruebas unitarias en estas pruebas de interfaz se deben probar determinados casos que no puedan ser cubiertos por las pruebas unitarias.

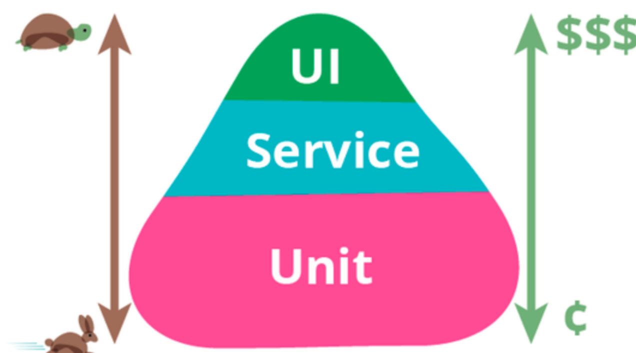
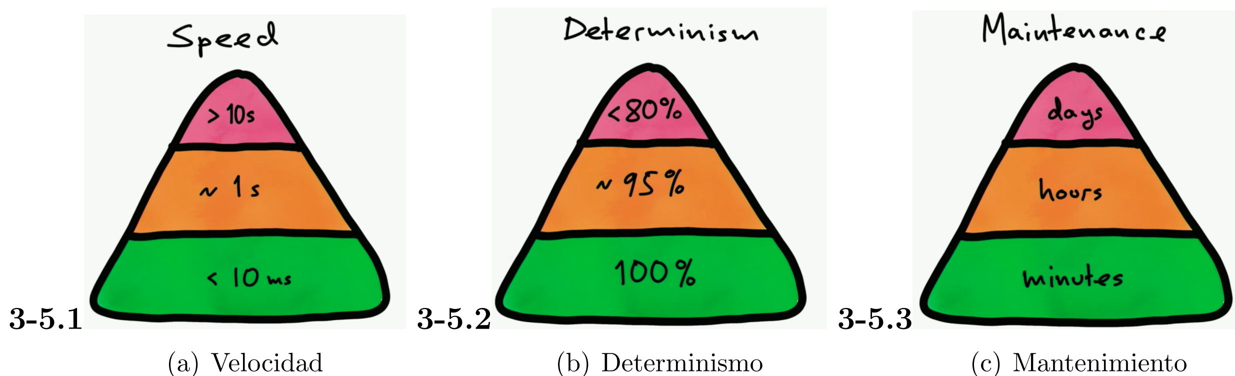


Figura 3-4.: Pirámide de pruebas reducida [9].

El porque de la estructura de esta pirámide de pruebas se puede observar con más detalle en la Figura 3-5 donde se muestra el comportamiento de la velocidad, complejidad del mantenimiento y determinismo de dichos niveles de prueba.



(a) Velocidad (b) Determinismo (c) Mantenimiento

Figura 3-5.: Velocidad, determinismo y mantenimiento en la pirámide de pruebas [9].

Las pruebas unitarias son más rápidas, menos costosas, más fáciles de mantener, 100 % determinística y por tanto el 100 % de las veces que se ejecutan tienen el mismo comportamiento. Las pruebas de interfaz no son el 100 % determinísticas por lo que no siempre tienen el mismo comportamiento y por tanto para un mismo escenario pueden fallar o ser exitosas porque dependen de muchos factores externos, son más costosas, más difíciles de mantener y su ejecución es más demorada que las unitarias. Además, la ejecución de una prueba automática y/o automatizada es más rápida que una prueba netamente manual.

3.1.3. Estrategias de automatización

La estrategia de automatización de pruebas complementa la estrategia de pruebas definida. Por tanto, a partir de la estrategia de pruebas se busca definir cómo llevar a cabo la implementación del conjunto de pruebas definido dentro de un ambiente automatizado. En Mayo de 2019, en el seminario de «El futuro de las Pruebas Automáticas de Software en Colombia» en la Universidad de lo Andes, Carlos Álvarez director técnico de la compañía Globant indicó que en primer lugar al establecer una estrategia se debe tener claro respecto al producto/proyecto [11]:

- Procesos.
- Niveles de Calidad.
- Alcance y Cobertura.
- Equipo de trabajo.

- Herramientas.
- Ambientes (móvil, web, nube).
- Métricas.

Adicionalmente los factores clave para definir la estrategia de automatización se basan en dos grupos [11]:

- Fit Organizacional
  - Complejidad del proyecto.
  - Recursos económicos y humanos.
  - Tecnologías existentes.
  - Ambientes de pruebas (plataformas, dispositivos, browser).
  - Tipos de pruebas.
  - Reportes.
- Fit Técnico
  - Tamaño de la comunidad, soporte y documentación.
  - Ciclo de desarrollo (integración y metodología).
  - Ciclos de Feedback Loop y reportes.
  - Cobertura de la automatización.
  - Robustez y mantenibilidad de la automatización.

De esta manera, se establece el proceso de automatización como el siguiente [11]:

1. Conocer bien el producto para definir qué herramientas permiten llevar a cabo la automatización en el mismo.
2. Definir la estrategia que permita determinar la evolución de la automatización a medida que avanza el producto.
3. Diseñar y construir el Framework de automatización propio; pueden utilizarse frameworks de la industria, pero lo más aconsejable es crear el framework propio.
4. Definir las prioridades de la automatización, que permita la evolución de las mismas a lo largo de su ciclo de vida.
5. Desarrollo, ejecución de las pruebas y entrega de resultados de las mismas.
6. Integración de la automatización con los equipos de desarrollo.
7. Automatización en un entorno de integración continua.

3.1.4. Problemas en las pruebas automatizadas

«El fenómeno de pruebas "problemáticas"más analizado se conoce como flaky test» [10] o pruebas inestables. Algunos scripts o casos de pruebas automatizados se pueden comportar de forma no determinística (comportamientos diferentes en cada ejecución) y son frágiles a cualquier cambio en el ambiente de ejecución de las pruebas, lo que puede catalogar una prueba como exitosa o fallida de forma errónea. Las pruebas de interfaz gráfica son las más inestables en la industria porque interfieren muchos factores externos que las hacen precisamente no determinísticas. Algunos generadores de pruebas flaky son [10]:

- Fallas en el hardware.
- Problemas de configuración del ambiente, motor de integración continua y dependencias.
- Latencia en la red y conectividad eventual.
- Sobrecarga de recursos tanto en la máquina que corre las pruebas como en la que ejecuta la app.
- Carga y concurrencia actual de la app bajo pruebas.
- Condiciones de ejecución (race conditions).
- Problemas de desempeño en la app bajo pruebas.
- Datos/caché que se mantiene de pruebas anteriores.
- App está en estado idle (es decir, en espera para un nuevo evento).
- Lanzar y ejecutar más emuladores/browsers lleva a que los tiempos de respuesta y procesamiento decaigan.

«¿Cómo se evitan las pruebas "flaky¿ En el caso de usar APIs de automatización que no auto detectan, cuándo la app está en estado idle (es decir, en espera para un nuevo evento), se deben usar timeouts grandes, con efecto colateral en la duración de las pruebas. Otra opción es implementar mecanismos propios de detección. Independientemente de la API, se deben reducir al máximo cualquier elemento que pueda introducir latencia durante las pruebas» [10].

3.1.5. Composición del sistema automatizado de pruebas

Para implementar un sistema automatizado de pruebas es importante conocer cuál es su composición para poder establecer cuales son las necesidades, requerimientos y proyecciones en la implementación de dicho sistema, debe ser claro que requiere este sistema para poder se implementado porque de lo contrario su ejecución podría no ser la más efectiva y esperada.

En la Figura **3-6** se puede observar la composición de un sistema automatizado de pruebas mencionada por Cristian Arandia de Endava en el seminario de «El futuro de las Pruebas Automáticas de Software en Colombia» en la Universidad de los Andes en Mayo de 2019 ([11]).

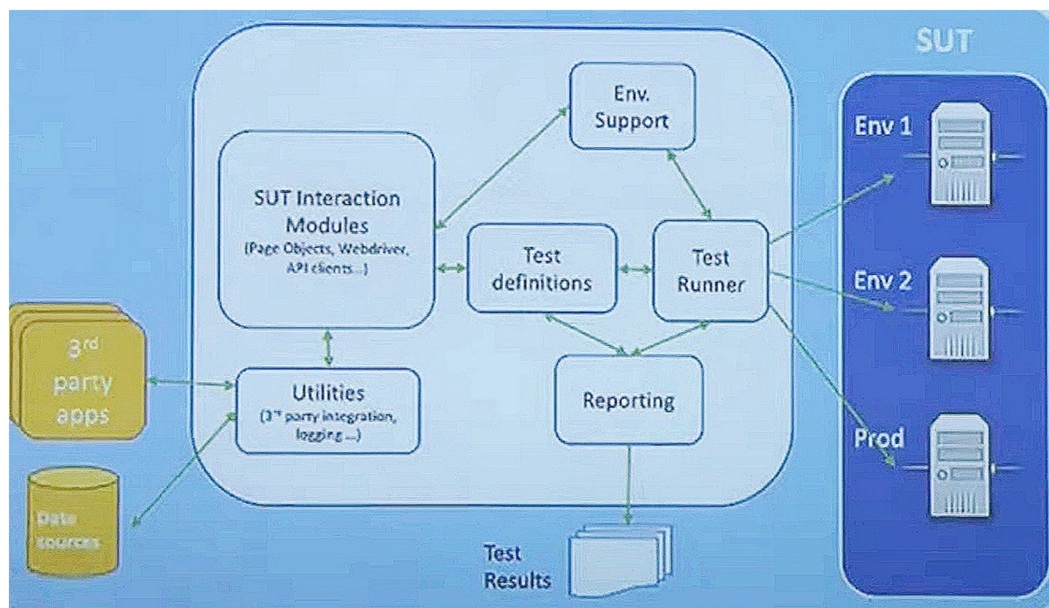


Figura 3-6.: Composición del sistema automatizado de pruebas [11].

Por tanto, la ejecución de las pruebas requiere de un hardware sobre el cual se ejecutará el software que implementa dichas pruebas (SUT parte derecha), dicho software a su vez se compone de:

- **SUT Interaction Modules**: Módulo de interacción con el sistema bajo pruebas que permite ejecutar acciones, interacciones y/o paso a paso del set de pruebas sobre el software desarrollado, por tanto es el encargado de:
  - Construcciones de las peticiones a enviar.
  - Realizar llamados a APIs.
  - Realizar llamados a métodos específicos del software.
  - Consultar Bases de Datos.

- Ejecutar los asserts/verificaciones de los resultados obtenidos de las pruebas.
- Creación y ejecución de Mocks/stubs o llamados a terceros.
- Interacciones con el hardware externo y/o sensores (aplicaciones móviles).

- **Test Definition**: Definición de los pasos a ser ejecutados.
  Durante el diseño de pruebas se establecen los escenarios con los cuales se indica el proceso con el paso a paso a seguir para garantizar y validar el cumplimientos de los requisitos del software, esta secuencia de pasos es la que compone el conjunto de los Test Definition.
  En algunos casos como en las pruebas unitarias se ejecutan directamente los test definition, en otros casos con la implementación de estrategias de pruebas como en el caso de BDD, en primer lugar se definen los escenarios con un lenguaje comprensible para el usuario dentro de unos archivos conocidos como feature a partir de los cuales se implementan los Test Definition con la ayuda de herramientas como Cucumber.

- **Test Runner**: Encargado de establecer la comunicación entre el set de pruebas y el SUT, es quien lanza la ejecución de las pruebas y ejecuta el paso a paso (test definition) de los escenarios previamente definidos.

- **Reporting**: Módulo de reporte de resultados para su posterior análisis. La creación de reportes es un aspecto necesario e importante para el análisis de la calidad del software dado que en estos se define si las pruebas automáticas se ejecutaron correctamente, el porcentaje de pruebas que fallaron, los problemas presentados durante la ejecución de las pruebas, los tiempos de ejecución de las pruebas, detectar a tiempo los errores y posibles causas para su posterior solución, de forma más rápida y eficiente. Así mismo permite llevar un seguimiento de los recursos y tiempo utilizados en la ejecución de las pruebas.

- **Utilities**: Módulo de integración con librerías o APIs de terceros/externos. Dada la amplia oferta en desarrollo de software actualmente existen muchas implementaciones que pueden ser reutilizadas dentro de otros desarrollos. La integración con terceros es importante y necesaria dado que un software debe realizarse de tal manera que sea integrable con otros.

Teniendo en cuenta las metodologías implementadas actualmente en cuanto a los procesos de pruebas y automatización de los procesos de pruebas es importante tener claros los objetivos, funcionalidades y procesos del software sobre el cual se implementará un sistema de pruebas y por tanto sobre el cual se ejecutará la automatización de dicho sistema de pruebas. Cabe aclarar que el presente proyecto se realiza con el fin de mejorar el proceso de pruebas de **Software DetectID™** por medio de la automatización del sistema de pruebas actual dentro de un entorno de integración continua. La información más detallada de este software se muestra a continuación:

3.2. Software DetectID™

DetectID™ [8] es un producto distribuido por la compañía **AppGate** cuyo objetivo principal es ofrecerle seguridad al usuario final en su interacción con aplicaciones que involucren cualquier tipo de información sensible para el mismo, las características de este producto se pueden observar en la Figura **3-7** (más información en https://www.easysol.net/eng/tfp/strong-authentication).

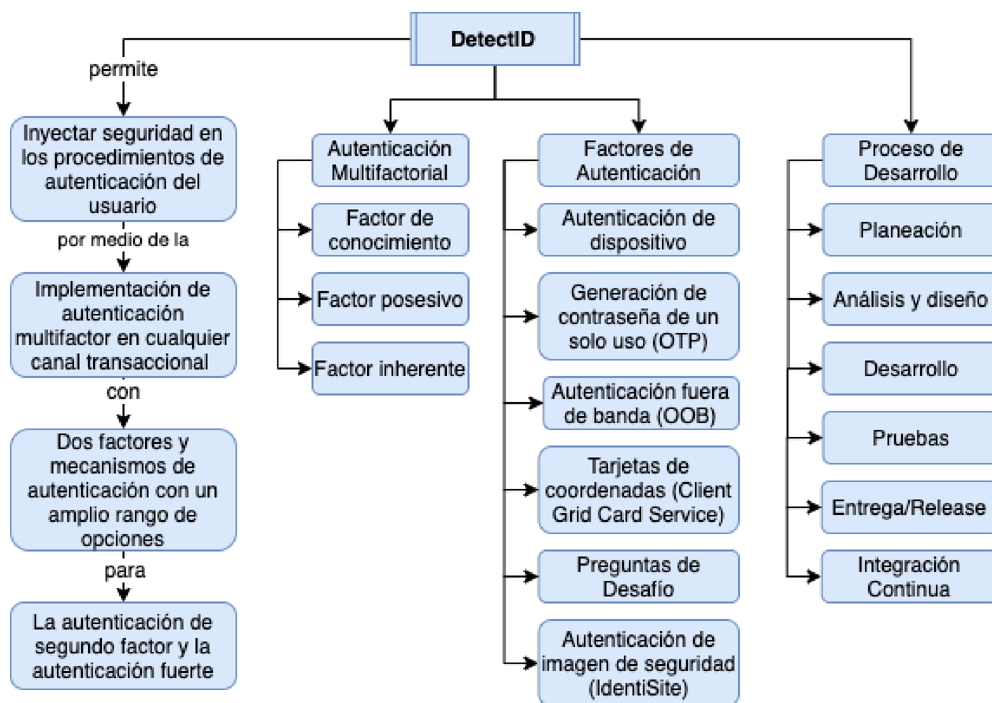


Figura 3-7.: DetectID™.

DetectID™ permite:

- Fortalecer el eslabón más débil de la cadena de seguridad del sistema de información: la autenticación del usuario final, dado que hoy día se utilizan técnicas muy simples para hacerse pasar por un usuario final de tal manera que logran robarse la identidad del usuario afectando no sólo su patrimonio sino además su vida social.

- Implementar autenticación multifactor en cualquier canal transaccional donde interactúen los usuarios finales con los sistemas de la entidad relacionada por medio de la autenticación adaptativa basada en el riesgo.

- Inyectar seguridad en los procedimientos de autenticación del usuario y seleccionar las opciones que mejor se adapten a sus problemas y necesidades de seguridad.

- Integración fácil con los canales transaccionales que exigen una mayor seguridad.
- Implementar diferentes tipos de factores y mecanismos de autenticación con un amplio rango de opciones para la autenticación de segundo factor y para la autenticación fuerte, incluidas las preguntas de desafío, el reconocimiento del sitio, las tarjetas de cuadrícula (tarjetas de bingo), los tokens OTP, fuera de banda autenticación por SMS y correo electrónico, y reconocimiento de máquina.

3.2.1. Autenticación multifactorial

Autenticación de múltiples factores donde se solicita dos o más factores de autenticación para verificar su identidad. Entre dichos factores de autenticación se tienen:

- Factor de conocimiento: algo que el usuario sabe.
- Factor posesivo: algo que tiene el usuario.
- Factor inherente: algo que el usuario es.

De esta manera, la combinación de factores de autenticación crean claves de seguridad para la protección de datos del usuario. La autenticación multifactor es diferente de la autenticación fuerte dado que la autenticación fuerte se puede lograr con dos o más métodos de autenticación del mismo tipo de factor (ejemplo, usando preguntas de desafío después de una autenticación de contraseña, ambas opciones son del mismo tipo de factor: algo que el usuario conoce).

3.2.2. Factores de autenticación DetectID ™

- **Autenticación de dispositivo**: Permite registrar dispositivos utilizados frecuentemente por un usuario final en un proceso de autenticación determinado, otorgando acceso al canal transaccional solo a aquellos que se registraron y autorizaron previamente. Así mismo, puede negar el acceso y/o generar alertas de dispositivos no registrados o no autorizados. De esta manera, se crea una huella digital única basada en las características de hardware del dispositivo. Es compatible con los sistemas operativos más utilizados: Windows, Linux, Mac, iOS y Android.

- **Generación de contraseña de un solo uso (OTP)**: La cual es válida únicamente para acceder a un canal transaccional una sola vez, así se reduce el riesgo de robo de credenciales relacionado con los mecanismos tradicionales de autenticación de contraseñas estáticas. Cada OTP es generado con Algoritmos basados en el tiempo (TOTP) o Algoritmos de respuesta de desafío OATH (OCRA). Un OTP puede ser proporcionado por:

- Tokens físicos: Dispositivos portátiles que generan contraseñas automáticamente ya sea en intervalos de tiempo TOTP o por respuesta de desafío OCRA.
- Tokens digitales: Aplicaciones instaladas en el dispositivo que generan contraseñas automáticamente ya sea en intervalos de tiempo TOTP o por respuesta de desafío OCRA.
- Integración con otros fabricantes: DetectID ^{TM} actúa como intermediario para la verificación de los OTP generados por otro sistema.

- **Autenticación fuera de banda (OOB)**: Esta categoría de autenticación permite agregar mayor seguridad al proceso de autenticación del usuario mediante el uso de diferentes canales para enviar contraseñas. De esta manera, los OTP se envían al usuario final a través de un canal diferente del que se utiliza para cumplir con el proceso de autenticación, se puede enviar por SMS o correo electrónico.

- **Tarjetas de coordenadas (Client Grid Card Service)**: Cada usuario final recibe una unidad física o digital donde se muestran diferentes valores alfanuméricos dentro de una matriz. Una vez que el usuario accede al canal transaccional debe ingresar los valores existentes en la tarjeta de acuerdo con las coordenadas que solicita el sitio web, entonces los valores serán verificados por el sistema de autenticación.

- **Preguntas de desafío**: Representa una autenticación fuerte basada en preguntas de desafío previamente definidas por la entidad y/o los propios usuarios. Por lo tanto, después de la verificación de la respuesta el acceso del usuario se otorga o se niega dependiendo de si la respuesta es correcta o no.

- **Autenticación de imagen de seguridad (IdentiSite)**: Se define una imagen que el usuario selecciona y asocia con una descripción para ayudar a identificar un portal transaccional real. De esta manera, cada vez que el usuario envía las credenciales la imagen seleccionada aparecerá junto con su descripción y, por lo tanto, el usuario valida que no está ingresando a un sitio falso.

DetectID^{TM} es un software amplio con los siguientes objetivos principales:

- Desarrollo de servicios o APIs que permiten la Autenticación Multifactorial.
- Desarrollo del **SDK** para la integración de los servicios de **DetectID^{TM}** con aplicaciones móviles.

- Desarrollo de interfaces para adaptarse a sistemas existentes de clientes.
- Desarrollo de interfaces gráficas para la gestión y administración del servicio.

De esta manera, cabe aclarar que **DetectID™** es un software para aplicaciones móviles, y por tanto se basa en un **SDK**. El SDK (software development kit) o Kit de Desarrollo de software es un conjunto de herramientas que permiten desarrollar aplicaciones para hardware o software en un lenguaje de programación específicos, permite la integración de diferentes tipos de software, funcionalidades, APIs dentro de una aplicación móvil. De esta manera, el **SDK** de **DetectID™** da paso a la integración de cualquier tipo de hardware y aplicaciones móviles con los servicios o APIs de **DetectID™** que permiten la implementación de Autenticación Multifactorial, garantizando la interacción segura del usuario dentro de dichas aplicaciones. A continuación se muestra en detalle la **metodología/proceso de desarrollo de software en DetectID ™ móvil**:

3.3. Metodología/proceso de desarrollo de software en DetectID ™ móvil

A partir del ciclo de vida de software se establece una metodología del producto que permita garantizar la distribución efectiva y de calidad del mismo, dicha metodología se basa en el siguiente proceso:

3.3.1. Planeación

Durante la planeación se establecen las prioridades a ejecutar durante un ciclo de desarrollo, se establecen los criterios de usuario, los resultados esperados, las tareas requeridas para la ejecución completa del ciclo de desarrollo y se estima el esfuerzo que se necesita para la ejecución de todas esas tareas.

3.3.2. Análisis y Diseño

Durante esta etapa los desarrolladores establecen la arquitectura, los flujos de información, el proceso requerido dentro de la funcionalidad y las tareas necesarias, para el desarrollo de código a partir de los requerimientos definidos.

Así mismo, el equipo de QA analiza y diseña los casos de usuario o escenarios que deben ser implementados dentro de la funcionalidad, cuando esta esté completamente desarrollada para verificar todos los caminos posibles en la misma, de tal manera que se cumpla con los criterios de usuario previamente definidos.

De igual forma, se analizan los escenarios previos que puedan ser afectados por los nuevos requerimientos y se realizan las respectivas modificaciones a estos. Finalmente se seleccionan los dispositivos donde se llevará a cabo la ejecución de los escenarios previamente definidos.

3.3.3. Desarrollo

Durante esta etapa se realiza el respectivo desarrollo de las tareas planeadas para llevar a cabo la funcionalidad de tal manera que se cumpla con los criterios establecidos. Por tanto, se crea el nuevo repositorio de código de ser necesario, o se actualiza un previo repositorio de código como se define en la etapa de diseño y posteriormente se realiza la implementación de código.

Así mismo, se lleva a cabo la creación de las pruebas unitarias con las cuales se hace una prueba general del código desarrollado para verificar el correcto comportamiento de los métodos implementados.

Finalmente, el desarrollo llevado a cabo se corre dentro de un pipeline de Jenkins donde se compila el código, ejecutan las pruebas unitarias, hace un análisis de código estático y crean los artefactos necesarios para su posterior implementación. De esta manera, si el pipeline es ejecutado con éxito el equipo de QA puede proceder con las pruebas.

3.3.4. Pruebas

Durante esta etapa, el equipo de QA implementa la pruebas previamente diseñadas. De esta manera, con los escenarios diseñados, modificados y/o seleccionados en la etapa de diseño, se procede a ejecutarlos dentro de los dispositivos seleccionados para poder verificar el correcto comportamiento de la nueva funcionalidad y de las previas de tal manera que se pueda rectificar que no hubo afectación de estas últimas.

En dado caso de encontrar algún error durante la ejecución de las pruebas en primer lugar se repite el escenario nuevamente para replicar el error y asegurar que dicho error es presentado por el desarrollo implementado. Por tanto, al rectificar el error, el desarrollador procede a darle solución y una vez resuelto el problema se reinicia el proceso de pruebas.

Cabe aclarar que actualmente el proceso de pruebas es completamente manual y no hay pruebas automatizadas de momento.

3.3.5. Entrega/Release

Una vez un desarrollo específico ha sido certificado por QA y definidas una fechas de entrega al cliente, se da inicio al proceso de Release el cual se basa en la ejecución de pruebas de Regresión donde se prueban unos escenarios específicos y fundamentales de toda la aplicación en general para comprobar el correcto comportamiento de toda la funcionalidad integrada y descartar cualquier anomalía.

Una vez certificada la integración correcta de toda la aplicación se procede a generar los artefactos requeridos para la implementación de la misma en el entorno del usuario final, se genera la documentación requerida para la entrega al mismo, como manuales, release notes, informes (Third party licenses, Technological surveillance), entre otros.

3.3.6. Integración Continua

Actualmente se ejecuta un pipeline por cada librería implementada dentro del producto donde se compila el código, ejecutan las pruebas unitarias, realiza análisis de código estático y generan los artefactos en cadena, dada la dependencia de una librería con otra. Sin embargo, no se tiene un proceso completamente automatizado y que realmente se ejecute de forma continua.

En conclusión, es importante que todo producto de software se encuentre a la vanguardia de los avances tecnológicos que están en constante actualización y renovación. Por tanto, en DetectID^{TM} móvil en donde ya se tiene una metodología y proceso de desarrollo de software claro, es importante y necesario mejorar su ciclo de desarrollo de tal manera que este se ajuste a los requerimientos tecnológicos actuales y permitan mejorar y garantizar un software de calidad al usuario final de forma más rápida y eficiente. En los siguientes capítulos se mostrarán dichas mejoras en el proceso actual que involucra la implementación de un sistema automatizado de pruebas dentro de un entorno de integración continua.

4. Implementación: Automatización del Sistema de Pruebas de DetectID^{TM}

Teniendo en cuenta las necesidades de mejora dentro del ciclo de desarrollo del software DetectID^{TM} móvil, se establece la metodología a implementar para la automatización de un sistema de pruebas y se lleva a cabo la implementación de la misma dentro de un entorno de integración continua. Por tanto, en el presente capítulo se muestra como llevar a cabo la implementación de dicho sistema teniendo en cuenta:

- Metodología para la automatización de un sistema de pruebas: se indican los requerimientos para la selección de la metodología, flujo de la metodología con el paso a paso, condiciones de ejecución, herramientas y estrategias de implementación (Cucumber, Mock, Jenkins), generación de reportes, entre otros. De esta manera, con la metodología definida, se procede a la creación del framework de automatización de pruebas.

- Creación del framework de automatización de pruebas: se muestra la estructura de los proyectos en ambas plataformas (Android-iOS) y de igual forma, la creación de dichos frameworks requiere de la implementación de escenarios a automatización.

- Implementación de escenarios a automatización: se muestran dos features creados para la automatización de dos funcionalidades del SDK de DID y a partir de dichos features se muestra a continuación como llevar a cabo la configuración del entorno de pruebas.

- Configuración del entorno de pruebas: se indica como hacer la implementación y configuración paso a paso de los frameworks de prueba en ambas plataformas. A continuación se muestra como realizar la configuración e implementación de un Mock.

- Implementación de un Mock: para la ejecución de las pruebas es necesario implementar Mocks en ambas plataformas. A continuación, para la correcta ejecución de los frameworks de automatización se requiere llevar a cabo la integración con el SDK-DID.

- Integración con el SDK-DID: el proyecto de automatización de pruebas debe integrase con el software sobre el cual va a ejecutar las pruebas programadas en ambas plataformas. Por tanto, cuando el software esta integrado con los frameworks de automatización, se procede a la implementación de los pasos a ejecutar en cada escenario.
- Implementación de los pasos a ejecutar en cada escenario: a partir de los features previamente definidos donde se muestra como se crean las ejecuciones de cada paso definido en un feature en ambas plataformas. De esta manera, con el sistema de pruebas ya automatizado, se procede a definir como llevar a cabo la ejecución de las pruebas.
- Ejecución de las pruebas: donde se indica paso a paso como ejecutar las pruebas en ambas plataformas y los requerimientos de software adicionales para llevarlo a cabo. A partir de la ejecución de dichas pruebas, se indica adicionalmente como realizar la generación de informes.
- Generación de informes: donde se indica como configurar y generar los informes de la ejecución de las pruebas en ambas plataformas. Finalmente el capítulo concluye indicando como dentro del ciclo de desarrollo de software de DetectID™ se lleva a cabo la implementación en un entorno de integración continua.
- Implementación en un entorno de integración continua: donde se indican las herramientas requeridas, la configuración y ejecución de un proyecto en un entorno de integración continua paso a paso, incluyendo el sistema de pruebas automatizado dentro de dicho proyecto.

4.1. Metodología para la automatización de un sistema de pruebas

Para seleccionar la estrategia o metodología de automatización, se consideraron los siguientes requerimientos:

1. Posibilidad de ejecutarlo en más de un dispositivo (simulado o físico) simultáneamente.
2. Evitar el uso de componentes de interfaz gráfica (GUI) para disminuir los bugs asociados a la interacción con la aplicación de pruebas.
3. Posibilidad de ser ejecutado e integrarse con una estrategia de integración continua.
4. Uso de una DSL (lenguaje específico de dominio) para que los Clientes-Usuarios (Stakeholders) pudieran validar la funcionalidad en términos del negocio.
5. Disminuir la dificultad en la dependencia con servicios de terceros o externos (DID - Backend), de tal forma que existiera un único punto de control.

6. Permitir la visualización de reportes para su posterior análisis y seguimiento de errores.

De esta manera se define el siguiente **flujo de la metodología para la automatización de un sistema de pruebas**:

4.1.1. Flujo de la metodología de automatización de pruebas

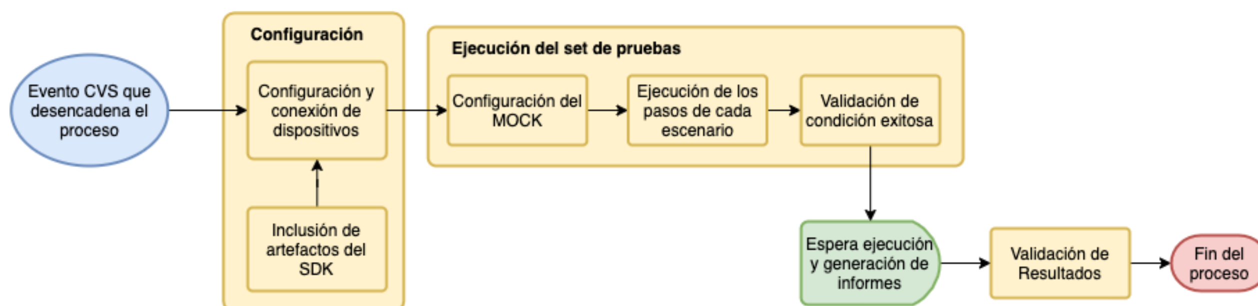


Figura 4-1.: Flujo de la metodología de automatización de pruebas.

En la Figura 4-1 se muestra el flujo de la metodología a implementar, donde:

1. Evento CVS (Sistema de Versiones Concurrentes) que desencadena el proceso: El código se encuentra en un repositorio de código (CVS) dentro del cual, en cualquier momento que se realice una modificación en el mismo se genera un evento indicando los cambios realizados y dando inicio a la ejecución del proyecto modificado.

2. Configuración y conexión de dispositivos: La ejecución de las pruebas se lleva a cabo en dispositivos simulados que deben indicarse cuáles son y a su vez ejecutarlos en la máquina de pruebas o en dispositivos físicos que deben ser conectados a dicha máquina.
   - Inclusión de artefactos del SDK: Para ejecutar las pruebas sobre el SDK, los artefactos del mismo deben ser ejecutados dentro de cada dispositivo. De esta manera, estos artefactos son incluidos dentro del proyecto de pruebas. Para el caso de Android los artefactos se encuentran en un servidor de dependencias Nexus de la compañía y para el caso de iOS los artefactos son copiados desde Jenkins.

3. Ejecución del set de pruebas: Con los artefactos y los dispositivos activos se procede a ejecutar las pruebas automatizadas que incluyen:
   - Configuración del MOCK: Para eliminar dependencia con terceros se mockean los servicios llamados por el SDK.
   - Ejecución de los pasos de cada escenario: Con el mock configurado y activo, se procede a ejecutar uno a uno los pasos dentro de cada escenario.

   - Validación de condición exitosa: Dentro de la ejecución de los pasos de cada escenario se hace la verificación del cumplimiento de los criterios establecidos para determinar si un escenario se ejecutó o no exitosamente.

4. Espera ejecución y generación de informes: Se ejecutan la totalidad de los escenario, para dar paso a la creación de los reportes o informes de prueba donde se indica la ejecución exitosa o no exitosa de los escenarios implementados.

5. Validación de resultados: Finalmente si la ejecución es exitosa el proceso de ejecución termina exitosamente. En caso de que exista un error en la ejecución de los escenarios el proceso finalizará como fallido y se indicará el estado del mismo y sus errores.

Dentro de una ejecución se pueden presentar dos estados finales, una condición de falla o condición exitosa. Por tanto, la metodología debe definir cuales son los **requerimientos para la condición exitosa**.

4.1.2. Requerimientos para la condición exitosa

Para poder llevar a cabo la ejecución más adecuada y correcta del sistema de pruebas:

- Debe existir una versión pública de las Librerías del DID-SDK que permita al sistema de pruebas tener acceso a dichos artefactos. Actualmente, estos artefactos se almacenan en un servidor de dependencias denominado NEXUS de la compañía.

- Se debe contar con un ambiente permanente de dispositivos físicos y emuladores sobre los que se ejecutarán las pruebas.

- El proyecto debe tener un evento adecuado que inicie el proceso de pruebas.

La metodología implementada incluye la estrategia de pruebas BDD. Por tanto, se debe incluir herramientas como **Cucumber** para poder llevar a cabo las pruebas con esta estrategia.

4.1.3. Ejecución de pruebas con Cucumber

La inclusión de la herramienta de **Cucumber** es una formalidad que permite retroalimentar al “Stakeholder”de acuerdo a sus requerimientos.

Mediante Cucumber se puede implementar la estrategia de pruebas de BDD y se crean los escenarios de prueba a partir de los denominados archivos feature, los cuales por medio de cucumber se sincronizan con las clases de prueba o step definitions desde los cuales se ejecutan el paso a paso de los escenarios descritos en cada feature.

Para la ejecución de pruebas en Android, se utiliza la herramienta de **cucumber** y para iOS se utiliza la herramienta disponible denominada **cucumberish**.

Como se ha mencionado, la implementación de un **Mock**, permite eliminar cualquier dependencia con servicios de terceros, por tanto es incluido como estrategia de pruebas dentro de dicha metodología.

4.1.4. Mock como estrategia para las pruebas

Para eliminar la dependencia con terceros se utilizan Mocks para simular los servicios llamados por el SDK. De esta manera, se usan stubs para simular los servicios de DID Server por el tiempo de economía al incluir dentro del proyecto esta tarea y porque permite tener un único punto de mantenimiento del proceso de pruebas.

Esto quiere decir que cada uno de los dispositivos conectados, tendrá el mock correspondiente para responder a los requerimientos del SDK, para el caso de Android. Para iOS se levanta un servidor dentro de la máquina de pruebas para ejecutar dichos stubs.

La herramienta seleccionada fue http://wiremock.org/, que entrega un API java que permite tener control del stub a través de código.

Dentro de la metodología se incluye la **generación de reportes de las pruebas automatizadas** porque permiten hacer seguimiento de las pruebas para detectar los errores presentados, o asegurar el correcto funcionamiento de las mismas.

4.1.5. Generación de reportes de las pruebas automatizadas

Para el caso de Android, los reportes generados de cucumber en el pipeline de Jenkins, son generados mediante el Plugin de Cucumber Reports, que permiten visualizar cada ejecución de Jenkins con el respectivo reporte y la fecha de ejecución.

En el caso de iOS los reportes se generan en un archivo XML, que mediante el Plugin de JUnit permiten visualizar cada ejecución de Jenkins en cada dispositivo utilizado.

Finalmente, la metodología permite que el sistema de pruebas automatizado sea implementado dentro de un entorno de **integración continua** por medio de la herramienta de **Jenkins** como se indica en 4.1.6.

4.1.6. Jenkins como estrategia de integración continua (CI)

La inclusión de un pipeline declarativo en el proyecto, permite la ejecución de las pruebas dentro de un sistema de CI por medio de la interacción con herramientas como **Jenkins**. De esta manera, las pruebas podrán ser ejecutadas automáticamente dentro de la máquina de pruebas cada vez que se requiera validar el comportamiento correcto del software luego de los cambios realizados en el mismo, los criterios de usuario, entre otros. Existen dos pipelines para cada proyecto de pruebas que se mostrarán en detalle más adelante.

En dado caso de desear ejecutar el pipeline cada tiempo determinado se debe agregar al pipeline `triggers - cron`, teniendo en cuenta la estructura de cron: `'Minutos (0-59) Horas (0-23) Días del Mes (1-31) Meses (1-12) Días de la Semana (0-7)'`, de la siguiente manera:

```
triggers{
    cron('0 12,23 * * *')
}
```

Por tanto, el pipeline será ejecutado diariamente a mediodía y a las 11 de la noche.

De esta manera, la definición de la metodología permite dar paso a la **creación de los frameworks de automatización de pruebas**.

4.2. Creación framework/proyecto de automatización de pruebas

Se crean dos frameworks de prueba para cada sistema operativo de dispositivos móviles uno para Android y otro para iOS. La estructura de dichos frameworks/proyectos de prueba se basa en una estructura por módulos que se muestra de forma general en la Figura 4-2, donde se discriminan los features, pasos de ejecución (steps), entidades configuraciones de las pruebas, mocks/stubs. Dentro de cada directorio o paquete se discriminan las configuraciones específicas de dicho módulo de las clases de implementación y en donde se requiera se discrimina por funcionalidad del SDK como se muestra a continuación:

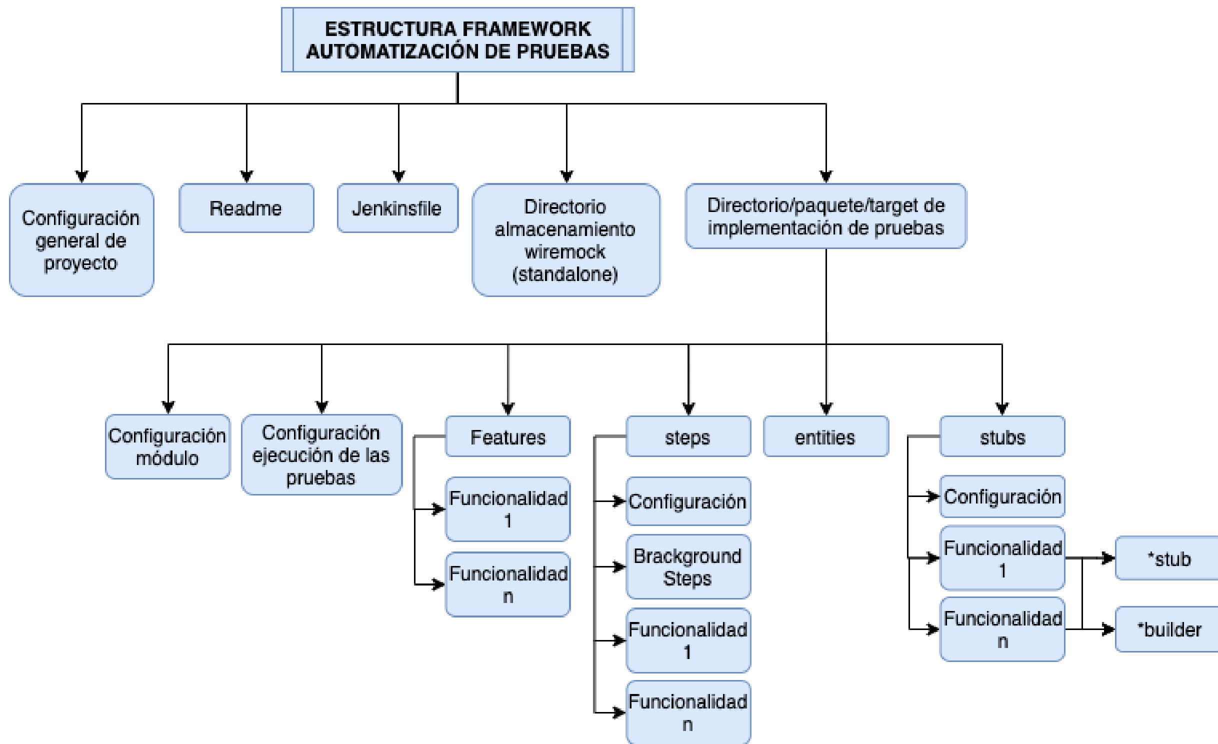


Figura 4-2.: Estructura general por módulos del proyecto de automatización de pruebas.

4.2.1. Estructura del proyecto Android

La estructura del proyecto de Android se ve de forma general en la Figura 4-3, y se explica con más detalle a continuación:

→ **app**

- **manifests**
  - **AndroidManifest.xml**: Archivo básico de configuración de la aplicación, en donde se definen los activities de la app, sus permisos (de acceso al dispositivo), entre otros.
- **java**
  - **assets**: Recursos adicionales para la ejecución de pruebas y features.
    - **features**: Carpeta de almacenamiento de los archivos .feature para la ejecución de los escenarios.

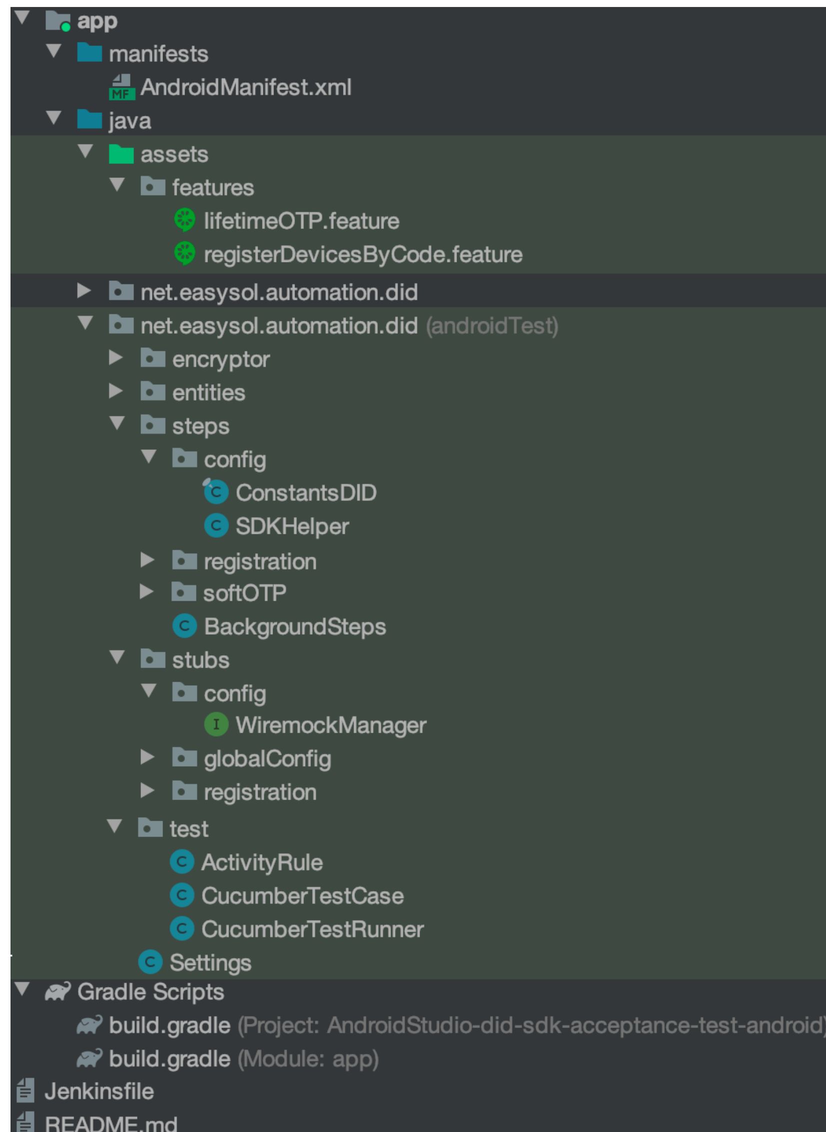


Figura 4-3.: Estructura del proyecto de automatización de pruebas Android, tomada desde Android Studio.

- **net.easysol.automation.did (androidTest)**: Directorio/paquete para las pruebas de aceptación.
  - **encryptor**: Permite encriptar la información entregada por el servidor de DID.
  - **entities**: Son las entidades del SDK de DID necesarias para la implementación de los Mocks.
  - **steps**: Pasos de ejecución de los features:
    - **config**: Archivos de configuración para la ejecución de los escenarios.

  - ◦ **ConstantsDID**: Constantes requeridas para la ejecución de los casos de prueba, relacionados con variables de ambiente, respuestas a servicios definidas, directorios, entre otros.
  - ◦ **SDKHelper**: Métodos comunes usados en las pruebas relacionados con la ejecución del SDK de DID.
- ▷ **registration** y **softOTP**: Son los directorios donde se encuentran los steps específicos de cada funcionalidad por a parte. De esta manera, en registration, se encuentran los los steps relacionados con el registro de dispositivos; mientras que en softOTP, se encuentran los steps relacionados con la generación de un OTP. Estos se componen de diferentes archivos como:
  - ◦ ***Steps**: Ejecución de cada paso especificado en los casos de pruebas dentro del feature.
  - ◦ ***Logic**: Capa lógica, implementación de los métodos correspondientes para la ejecución de los pasos, llamados al SDK, llamados a API, entre otros.
- ▷ **BackgroundSteps**: Ejecución de los pasos de prueba descritos en el background del feature, donde se definen los pasos comunes a los escenarios de prueba.

☐ **stubs**: Configuración e implementación de los Mocks del servidor de DID.

- ▷ **config**:
  - ◦ **WiremockManager**: Interfaz a implementar en cada stub creado.
- ▷ **globalConfig** y **registration**: Son los directorios donde se encuentran los mocks específicos de cada servicio por aparte, los cuales se componen de diferentes archivos como:
  - ◦ ***Stub**: Cada stub de determinado servicio creado para la ejecución adecuada de los escenarios.
  - ◦ ***Builder**: Constructores de la respuesta entregada por cada Stub.

☐ **test**: Configuración de las pruebas y ejecutores de las pruebas.

- ▷ **ActivityRule**: Actividad para la implementación de las reglas para la ejecución de las pruebas.
- ▷ **CucumberTestCase**: Configuración (features, steps, tags, reports).
- ▷ **CucumberTestRunner**: Clase de instrumentación para la creación del paquete de pruebas y ejecución de las pruebas.

☐ **Settings**: Configuración del contexto, con el cual es posible ejecutar las pruebas.

→ **GradleScripts**

- **build.gradle** (Project): Configuración de dependencias, plugins, paquetes, entre otros, para el proyecto en general.
- **build.gradle** (Module:app): Configuración de dependencias, plugins, paquetes, versiones, pruebas, creación de tareas gradle específicas de la app.

→ **Jenkinsfile**: Configuración para la ejecución del pipeline de pruebas.

→ **README.md**: Archivo con información sobre la instalación y ejecución de dispositivos emulados, con la preparación y ejecución de las pruebas de aceptación y generación de reportes de prueba.

4.2.2. Estructura del proyecto iOS

La estructura del proyecto de iOS se ve de forma general en la Figura 4-4, y se explica con más detalle a continuación:

→ **wiremock**: Carpeta de almacenamiento del wiremock-standalone-2.25.1.jar, para su posterior ejecución, así como de los stubs previamente creados (mappings).

→ **Jenkinsfile**.

→ **README.md**.

→ **automationCucumberishCucumberTests**: Directorio para las pruebas de aceptación.

- **DID-SDK-Libraries**: Librerías (framework) del SDK, necesarias para la implementación de las pruebas dentro del SDK.
- **Features**.
- **entities**: Carpeta con las estructuras requeridas para la construcción de los JSON, que permitirán la correcta creación de los stubs.
- **stubbing**: Carpeta con las clases correspondientes a la administración del wiremock, creación y ejecución de los stubs (`http://wiremock.org/docs/api/#tag/Stub-Mappings/paths/~1__admin~1mappings/post`):
  - ▷ **config**:
    - ◦ **WiremockClient**: Permite la creación de stubs que serán guardados dentro del mappings del Wiremock Server ejecutado, por medio de la instrucción: createMapping.

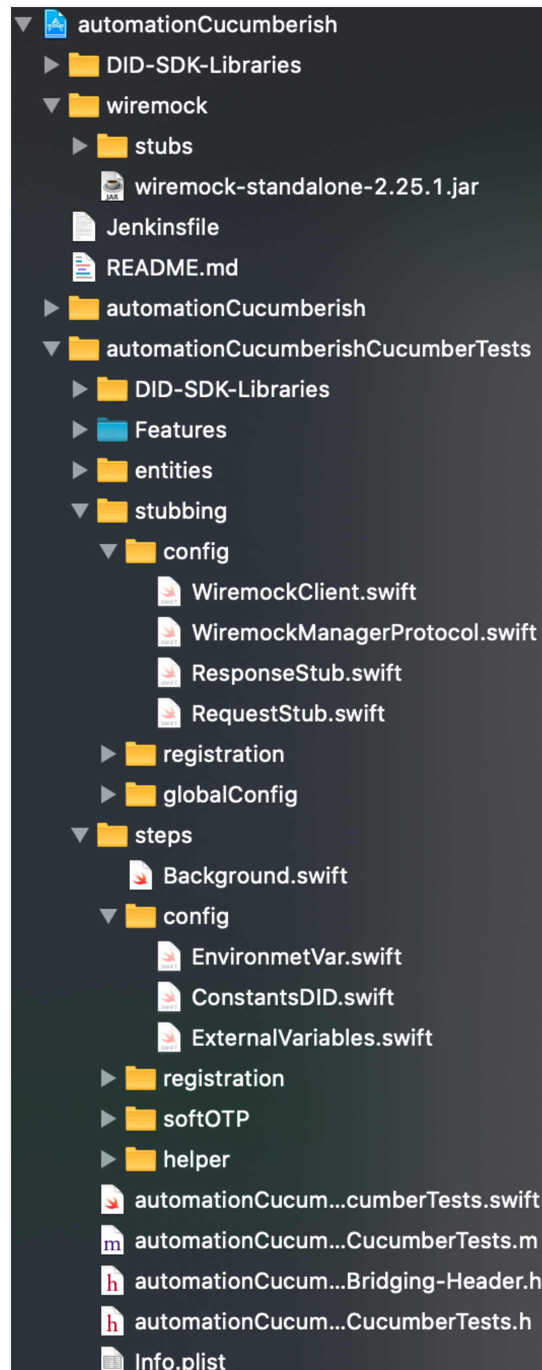


Figura 4-4.: Estructura del proyecto de automatización de pruebas iOS, tomada desde Xcode.

- **WiremockManagerProtocol**: Protocolo definido para ejecutar cada stub.
- **ResponseStub**: Clase para la creación de un stub, con la estructura correspondiente al Response con el cual responderá un stub a una previa solicitud.

    - **RequestStub**: Clase para la creación del stub, con la estructura correspondiente al Request donde se establece cuando deberá responder un stub ante una petición.
  - **registration** y **globalConfig**: Con sus respectivos archivos ***Stub** y ***Builder**.
- **steps**:
  - **Background**.
  - **config**:
    - **EnvironmentVar**: Variables de ambiente requeridas para la ejecución de los stubs, tales como la IP, que es capturada desde **ExternalVariables** y el puerto.
    - **ConstantsDID**: Constantes requeridas para la ejecución de las pruebas.
    - **ExternalVariables**: Clase creada al principio de la ejecución de las pruebas donde se establecen las variables de ambiente tales como la IP de la máquina donde se están ejecutando las pruebas.
  - **registration** y **softOTP**: Con sus respectivos archivos ***Steps** y ***Logic**.
  - **helper**: Métodos comunes usados en las pruebas relacionados con la ejecución del SDK de DID, llamados API y administración de llaves pública y privada.
- **automationCucumberishCucumberTests.swift**: Clase para la ejecución de los escenarios, levanta la aplicación, configura el directorio Features y las clases para ejecución de los Steps Definitions. Adicionalmente ejecuta las acciones anteriores (before) y/o posteriores (after) a cada escenario, según lo deseado.
- **automationCucumberishCucumberTests.m**: Configura la clase automationCucumberishCucumberTests para la ejecución de los escenarios por medio de la función CucumberishSwiftInit.
- **automationCucumberishCucumberTests-Bridging-Header.h**: Utilizado para la importación del SDK.
- **Info.plist**: Propiedades del proyecto.

Con la estructura de los frameworks/proyectos de automatización de pruebas ya definida, se procede a realizar la creación de los features donde se lleva a cabo la **implementación de escenarios a automatizar**.

4.3. Implementación de escenarios a automatizar

Se han implementado dos features para las siguientes funcionalidades:

- Registro de dispositivos por código (URL): Esta funcionalidad permite el registro de dispositivos móviles a través de la generación de una URL que contiene un código de registro.
- Periodo de vida del OTP: La funcionalidad del OTP, permite la generación de un número aleatorio que cambia cada cierto tiempo (definido como periodo de vida).

De esta manera los archivos features implementados se muestran en los Anexos A.1.1 como: «`registerDevicesByCode.feature`». El feature para el **registro de dispositivos por código URL** contempla el flujo completo de la funcionalidad que involucra los caminos correctos e incorrectos y se compone de:

- **Background**: Configuración y ejecución previa a cada escenario de los stubs del servidor. Configuración y ejecución del mock que se encargará de las funcionalidades correspondientes al servicio de registro, comprende:
  - Dado que el servicio de registro se configura para la compañía "AppGate": En este paso se realiza la configuración e inicialización del Mock de Registro y se establece una compañía para la ejecución de las pruebas.
- **Scenario**: El paso a paso, para la preparación, ejecución y validación de la prueba corresponde a:
  - Dado que el dispositivo ha sido registrado (SI/NO): Se verifica que el dispositivo a enrolar no se encuentra o si se encuentra registrado según lo requiere al escenario.
  - Dado que el dispositivo tiene acceso al servidor (SI/NO): según requiera el escenario. Si el dispositivo no tiene acceso al servidor el proceso de enrolamiento no se puede llevar a cabo, es indispensable tener conexión a la red para poder establecer comunicación con los servicios de registro.
  - Cuando se realiza el registro con un CÓDIGO determinado: En este paso se envía al servidor (Mock) el respectivo código (URL) requerido para llevar a cabo el registro del dispositivo, sea un código válido (VALID), erróneo (WRONG), expirado (EXPIRED) o en dado caso de que dicho dispositivo ya haya sido registrado (REGISTERED).
  - Entonces una RESPUESTA determinada es generada: Se procede a verificar que efectivamente la respuesta al ingresar cada código (URL) sea la correcta. Si no se tiene comunicación con el servidor (SERVER_ERROR), si el código fue erróneo (WRONG_CODE), si el código ya expiró (EXPIRED_CODE), si el dispositivo ya ha sido registrado (DEVICE_ALREADY_REGISTERED), o si el registro se llevó a cabo con éxito (POSITIVE).

- Entonces el dispositivo ha sido registrado (SI/NO): Se verifica que el dispositivo que se trató de enrolar no se encuentra o si se encuentra registrado según lo requiere al escenario.

De igual forma en el feature «`lifetimeOTP.feature`» que se muestra en los Anexos A.1.2, el feature para el **cambio del tiempo de vida del OTP** contempla el flujo correcto de la funcionalidad y se compone de:

- **Background**: Configuración y ejecución previa a cada escenario de los stubs del servidor. Configuración y ejecución del mock que se encargará de las funcionalidades correspondientes al servicio de registro y actualización de cuentas, comprende:
  - Dado los factores de autenticación: En primer lugar se proveen los factores de autenticación con los que el usuario puede autenticarse.
  - Dada la configuración del Tipo de Autenticación TOTP con un tamaño de 6 dígitos: En este paso se configura el tamaño del OTP y se indica que el registro se realizará con un factor de autenticación TOTP.
  - Dado el tiempo de vida del OTP inicial de 3 segundos: En este paso se configura el tiempo de expiración del OTP en 3 segundos.
  - Dada la configuración de los servicios de registro para la compañía "AppGate": En este paso se realiza la configuración e inicialización del Mock de Registro y se establece una compañía para la ejecución de las pruebas.
- **Scenario**: El paso a paso, para la preparación, ejecución y validación de la prueba corresponde a:
  - Dado que el dispositivo ya ha sido registrado: En primer lugar se debe realizar el registro del dispositivo y su verificación de registro.
  - Dada la configuración del Servicio de GlobalConfig con un tiempo de vida definido en value: En este paso se realiza la configuración e inicialización del Mock que permite la actualización del dispositivo registrado, en este caso se modifica el tiempo de vida del OTP al definido en el example.
  - Dado que el cliente cambia el tiempo de vida del OTP al valor definido en value: En ese paso se realiza la actualización del tiempo de vida del OTP.
  - Cuando el usuario solicita un OTP: En este paso se solicita el valor del OTP al definido en el example.
  - Entonces el OTP es generado con 6 caracteres: En este paso se valida el tamaño del OTP obtenido en el paso anterior, dado que se configuró con un tamaño de 6 debe tener ese tamaño.

- Entonces el OTP debe mantenerse durante el tiempo establecido en value antes de cambiar: En este paso se valida el tiempo en que el OTP cambia con el tiempo actualizado puesto que se configuró inicialmente con un valor, pero en el paso a paso del escenario este tiempo es modificado, debe validarse que efectivamente este cambio se haya realizado al definido en el example.

Para poder ejecutar los escenarios previamente definidos, en primer lugar se debe llevar a cabo la **configuración del entorno de pruebas** sobre el cual dichos escenarios serán ejecutados.

4.4. Configuración del entorno de pruebas

Dependiendo de cada proyecto, la configuración del entorno de pruebas se lleva a cabo de la siguiente manera con dos herramientas diferentes para cada plataforma:

4.4.1. Implementación con Cucumber - Android

La implementación de las pruebas en Android o conocidas también como pruebas instrumentadas, se ejecutan en dispositivos físicos y emuladores, se utilizan cuando las pruebas necesitan acceso a información de instrumentación (como el Contexto de la aplicación de destino) o si requieren la implementación real de un componente de marco de Android (como un objeto Parcelable o SharedPreferences). También ayudan a reducir el esfuerzo requerido para escribir y mantener códigos simulados. La implementación de estas pruebas instrumentadas se lleva a cabo de la siguiente manera:

En el proyecto de **Android Studio** se debe almacenar los archivos de origen para las pruebas instrumentadas en `module-name/src/androidTest/java/`, este directorio se crea por defecto en el proyecto y contiene un ejemplo de prueba instrumentada. Antes de comenzar, se debe descargar la configuración de la biblioteca de soporte de pruebas de Android (https://developer.android.com/training/testing/#setup), que proporciona APIs para crear y ejecutar rápidamente código de prueba instrumentado para las aplicaciones. La Biblioteca de soporte de pruebas incluye un corredor de prueba JUnit 4 (AndroidJUnitRunner) y APIs para pruebas de interfaz gráfica funcionales (Espresso y UI Automator).

También se debe configurar las dependencias de prueba de Android para que el proyecto pueda usar el corredor de prueba y las APIs de reglas proporcionadas por la biblioteca de soporte de pruebas. Para simplificar el desarrollo de las pruebas también se debe incluir la biblioteca Hamcrest que permite crear aserciones más flexibles utilizando las APIs de comparación de Hamcrest.

En el archivo «build.gradle» de nivel superior de la aplicación se deben especificar estas bibliotecas como dependencias como se observa en los Anexos A.2.1.

A continuación, se deben agregar los archivos que hacen que Cucumber ejecute el conjunto de pruebas. De esta manera, en el directorio donde debe estar el proyecto de prueba app/java/androidTest/ se agrega una clase de instrumentación donde se realiza la configuración para la ejecución de las pruebas instrumentadas como se observa en los Anexos A.2.1 «CucumberTestRunner.java».

Nota: Usar en build.gradle/testInstrumentationRunner para ejecutar las pruebas de Cucumber con la variable 'testInstrumentationRunner' en build.gradle esta clase debe estar en un paquete diferente al de los steps y debe apuntar al paquete donde se encuentran las pruebas, tanto los features como los steps (glue). La instrumentación permite que Cucumber y cree el paquete de prueba. De esta manera, se agrega el corredor de pruebas, como se observa en los Anexos A.2.1 «CucumberTestCase.java».

En la clase CucumberTestCase.java se tiene la etiqueta @CucumberOptions en la que se agrega la configuración llamada **features**. Por tanto, no es necesario llamar a la clase desde ningún otro lugar porque Cucumber buscará ese directorio para saber cómo encontrar características (features) y dónde configurar los informes.

También se debe ajustar el «build.gradle» en la sección de Android donde la referencia de sourceSets debe apuntar al entorno androidTest en la dirección correcta para encontrar los features de Cucumber y el código de prueba como se observa en los Anexos A.2.1

Adicionalmente, se debe agregar en la sección de configuración predeterminada defaultConfig en el «build.gradle» lo que permite a androidTest saber cómo se llama el paquete de prueba y qué tipo de corredor de instrumentación desea administrar al ejecutar las pruebas como se observa en los Anexos A.2.1.

Nota: Ya se tiene la clase de Instrumentación ubicada en una subcarpeta de androidTest llamada test. Por tanto, se debería poder ejecutar ./gradlew connectedCheck y obtener el mensaje "0 tests".

Ahora, dentro de la carpeta java se agrega un nuevo directorio de tipo assets dentro del cual se debe crear la carpeta denominada features, donde se crearan los múltiples escenarios a ejecutar según como se desee. Este directorio será el que buscará el corredor de pruebas para ejecutar los escenarios allí propuestos.

A continuación se crea la clase `Settings` dentro del paquete de `androidTest` con la cual se lanza la aplicación por medio de la ejecución de la actividad (Activity) y se configura el contexto (context) que serán usados en las pruebas como se observa en los Anexos A.2.1.

La etiqueta `@rule` permite una adición o redefinición muy flexible del comportamiento de cada método de prueba en una clase de prueba, cada `ActivityRule` debe extenderse desde `ActivityTestRule` como se muestra en la siguiente clase implementada en `ActivityRule.java` en los Anexos A.2.1.

Después de obtener toda la configuración se puede usar la clase `Settings` para implementar los métodos `getInstance()` y `getContext()` para obtener el contexto que se utilizará en las pruebas.

4.4.2. Implementación con Cucumberish - iOS

La implementación de las pruebas en dispositivos iOS, se lleva a cabo por la herramienta Cucumberish que es herramienta alterna a Cucumber para este sistema operativo. De la misma manera que en Android las pruebas pueden ser implementadas tanto en dispositivos físicos como en emuladores. Sin embargo, en muchos casos los emuladores tienen limitaciones que no permiten que las pruebas sean implementadas en ellos. Para el caso de iOS no se requiere un contexto para la ejecución de las pruebas, la máquina para la creación y ejecución de las mismas debe tener un sistema operativo **MacOS** y la implementación de éstas se lleva a cabo en los Anexos A.2.2.

Con las herramientas instaladas y la ejecución del `Podfile` se abre el proyecto desde su extensión **Workspace** y se inician a configurar las pruebas.

La configuración de las pruebas requiere de la implementación del corredor de pruebas. Para esto, se deben configurar los archivos creados por defecto dentro del target de pruebas `automationCucumberishCucumberTests`. Por tanto, en el archivo `automationCucumberishCucumberTests.m` se indica dónde se encuentra el ejecutor de las pruebas (resaltado en rojo) como se observa en los Anexos A.2.2.

Adicionalmente, se configura el ejecutor de pruebas como se observa en los Anexos A.2.2 «`automationCucumberishCucumberTests.swift`».

En la clase `automationCucumberishCucumberTests.swift` dentro `beforeStart` se indican las clases donde de encuentran los pasos a ejecutar, se inicializan las variables, clases y métodos que se requieran antes de lanzar la aplicación la cual se ejecuta con `application.launch()`.

Con: `Cucumberish.executeFeatures(inDirectory: ‘‘Features", from: bundle, includeTags:nil, excludeTags: nil)` se establece el directorio donde se encuentran los features o escenarios de prueba y en caso de requerirlo los tags con los que se caracterizan determinados pasos a ejecutar.

Finalmente dentro de `before` se indica los pasos a implementar antes de la ejecución de cada escenario como eliminar dispositivos previamente registrados, llaves creadas, stubs y peticiones a dichos stubs para garantizar la correcta ejecución de los escenarios en cualquier momento.

Ahora, el proyecto de pruebas debe incluir la **implementación de Mocks** mencionada en la metodología y la **creación de stubs** que permiten simular los servicios de terceros. Por tanto, cuando se utiliza el wiremock standalone (para el caso de iOS) éste debe ejecutarse (el archivo .jar) en primer lugar para luego proceder a la creación de los respectivos stubs, cuando se utiliza directamente la librería no se requiere ejecutar el wiremock standalone (para Android) si no que se crean directamente los stubs como se indica en 4.5.

4.5. Implementación de Mocks y creación de stubs

La implementación del Mock se hace por medio de la herramienta de **WireMock** como se muestra a continuación para cada plataforma:

4.5.1. Implementación en Android

En Android cada clase stub debe implementar la interfaz «`WiremockManager.java`» que permite la creación de cada stub y en dado caso de consumir APIs asíncronas, se implementa un método que da espera al consumo de dichas APIs como se observa en los Anexos A.3.1.

Un ejemplo de la implementación del Stub se muestra en los Anexos A.3.1 para «`GlobalConfigStub.java`». Con el método `startStub` se configura y/o inicializa el stub que respondería a la petición que solicite la url:
`http://localhost:8080/ConstantsDID.PATH_GLOBAL_CONFIG`, con el encabezado:
`Accept=ENCABEZADO_ACCEPT`. La respuesta del stub es la correspondiente a `dataResponse` con un código de respuesta definido en `STATUS_CODE_OK`. Cuando se implementan peticiones asíncronas el método `waitToConsume` da una espera para que la petición se ejecute efectivamente y otorgue la respuesta del stub, dicha espera es de 30 segundos para que se haga la petición a la misma url de lo contrario la ejecución fallará.

Como se indicó en la estructura para cada stub creado es importante crear una clase constructora que será la encargada de construir el cuerpo de la respuesta de cada stub. Por tanto, para este caso se tiene la clase «`GlobalConfigBuilder.java`».

4.5.2. Implementación en iOS

En iOS en primer lugar se debe levantar el servidor de **wiremock** que permita la creación y ejecución de los stubs. Al igual que en Android (donde se implementa una interfaz) en este caso se crea el protocolo que será implementado por cada stub «`WiremockManagerProtocol.swift`» como se muestra en la Anexos A.3.2.

Dentro de la clase «`WiremockClient.swift`» que se muestra en los Anexos A.3.2 se encuentran los métodos con los cuales los stubs serán creados por medio de la implementación del método `createStub`, al cual se le debe indicar cual es la petición a la cual responderá el stub (`RequestStub`), y la respuesta que mostrará el mismo ante dicha petición (`ResponseStub`). Este método a su vez llama al método `createMapping` el cual indica al siguiente método `sendWiremockOperation` que el stub sea creado con las características previamente definidas, por medio de una petición `POST` al servidor WireMock con el path: `/__admin/mappings`.

De igual manera, esta clase contiene otros métodos para el manejo de los stubs no sólo de su creación, si no adicionalmente para el reinicio o eliminación de las peticiones realizadas durante la ejecución de las pruebas a todos stubs con el método `deleteRequests` por medio de una petición `DELETE` al servidor WireMock con el path: `/__admin/requests`. Con el método `deleteStub` se eliminan los stubs que no se requieran en determinado escenario y que podría causar problemas en la ejecución de las pruebas, como por ejemplo en caso de que un stub contenga exactamente la misma petición pero con diferente respuesta puede presentarse algún conflicto, esto por medio de una petición `DELETE` al servidor WireMock con el path: `/__admin/mappings`. Finalmente con el método `shutDownMockServer` se da de baja al servidor del Wiremock en dado caso de requerirlo por medio de una petición `POST` al servidor WireMock con el path: `/__admin/shutdown`.

Finalmente, la clase «`RequestStub.swift`» que se muestra en los Anexos A.3.2 contiene la estructura de la petición a la que responderá el stub la cual debe contener una url, path, parámetros, encabezados y/o un cuerpo según como se requiera en cada stub. La clase «`ResponseStub.swift`» que se muestra en los Anexos A.3.2 contiene la estructura de la respuesta que responderá cada stub la cual puede contener un estado de la respuesta, mensaje de dicho estado, encabezados y un cuerpo. Existen más parámetros relacionados con cada una de estas estructuras, para mayor información consultar:
http://wiremock.org/docs/api/#tag/Stub-Mappings/paths/~1__admin~1mappings/get
en la sección: `Create a new stub mapping`.

Un ejemplo de la implementación del stub se muestra en los Anexos en A.3.2 para «`GlobalConfigStub.swift`». Con el método `setBodyStub` se crea el cuerpo de la respuesta del stub, con el método `startStub` se configuran la petición a la cual responderá el stub cuando se solicite la url: `http://localhost:9090/ConstantsDID().PATH_GLOBAL_CONFIG`, con el encabezado: `Accept=ENCABEZADO_ACCEPT`. Así mismo se configura la respuesta del stub es la correspondiente a `GlobalConfigStub.dataBody` con un código de respuesta `200` y un encabezado: `Content-Type:eapplication/json`.

Con el método `countRequests` se realiza una petición `POST` al servidor del Wiremock con path: `/__admin/requests/count` para determinar cuántas peticiones se han hecho al stub con url `http://localhost:9090/ConstantsDID().PATH_GLOBAL_CONFIG`. Para peticiones asíncronas el método `waitToConsume` da una espera para que ésta se ejecute efectivamente y otorgue la respuesta del stub, dicha espera es de 30 segundos para que se haga la petición a la misma url de lo contrario la ejecución fallará.

De igual manera que en Android para cada stub creado es importante crear una clase constructora que será la encargada de construir el cuerpo de la respuesta de cada stub. Por tanto, para este caso se tiene la clase «`GlobalConfigBuilder.swift`».

Para que el sistema de pruebas automatizado pueda ser implementado y pruebe efectivamente el software para el que fue diseñado, éste sistema debe integrarse con dicho software. De esta manera, a continuación se muestra como se lleva a cabo la **integración del sistema de pruebas automatizado con el SDK-DID**.

4.6. Integración del sistema de pruebas automatizado con el SDK-DID

Para integrar las pruebas con el SDK de DID se requieren implementar las librerías del mismo dentro del código de la siguiente manera:

4.6.1. Implementación en Android

Se recomienda agregar el repositorio google() en ambas secciones del «`build.gradle`» principal (del proyecto) y los repositorios y dependencias como se observa en los Anexos en A.4.1.

Dentro de la configuración de android se indica la versión soportada dentro del «`build.gradle`» de la aplicación, como se observa en los Anexos en A.4.1.

Adicionalmente se agregan las librerías dentro de las dependencias del «`build.gradle`» de la aplicación las cuales se encuentran en un repositorio Nexus de la compañía, se agregan algunas librerías de Android requeridas para la correcta implementación del SDK como se observa en los Anexos A.4.1.

Finalmente en el manifiesto de Android: «`AndroidManifiest.xml`» se declaran los servicios que son necesarios para que el SDK funcione correctamente bajo la etiqueta Application como se observa en los Anexos A.4.1. En el mismo archivo se otorgan los permisos necesarios para que la aplicación utilice las funcionalidades del dispositivo como se observa en los Anexos A.4.1.

4.6.2. Implementación en iOS

En primer lugar las librerías de iOS deben ser incluidas dentro del proyecto (archivos *.framework) como se muestra en la Figura **4-4**. Estas librerías deben ser implementadas dentro del target de la aplicación (Figura **4-5**) dando click en el botón + y dentro de la configuración del target de pruebas (Figura **4-6**) arrastrando las librerías dentro de **Copy Files**. Así mismo las librerías deben ser agregadas dentro del target de la aplicación como un grupo y dentro de la raíz del proyecto.

Así mismo, se realiza la siguiente configuración dentro del `Build Settings` para el target de la app y target de pruebas:

- Linking: Other Linker Flags: `-ObjC`.
- Build Options: Enable Bitcode: `No`.
- Search Paths: Framework Search Paths: `$(PROJECT_DIR)/DIRECTORIO_LIBRERIAS_SDK` y `$(SRCROOT)/DIRECTORIO_LIBRERIAS_SDK`. Donde `DIRECTORIO_LIBRERIAS_SDK`, es el directorio donde se encuentran almacenadas las librerías, para este caso: `automationCucumberish/DID-SDK-Libraries`.
- Swift Compiler - General: Install Objective-C Compatibility Header: `Yes`.
- Swift Compiler - General: Objective-C Bridging Header: `$(PROJECT_DIR)/DIRECTORIO_LIBRERIAS_SDK/didm_auth_sdk_iOS.framework/Headers/DetectID.h`. **Únicamente para el target de la app.**
- Build Options: Always Embed Swift Standard Libraries: `Yes`.

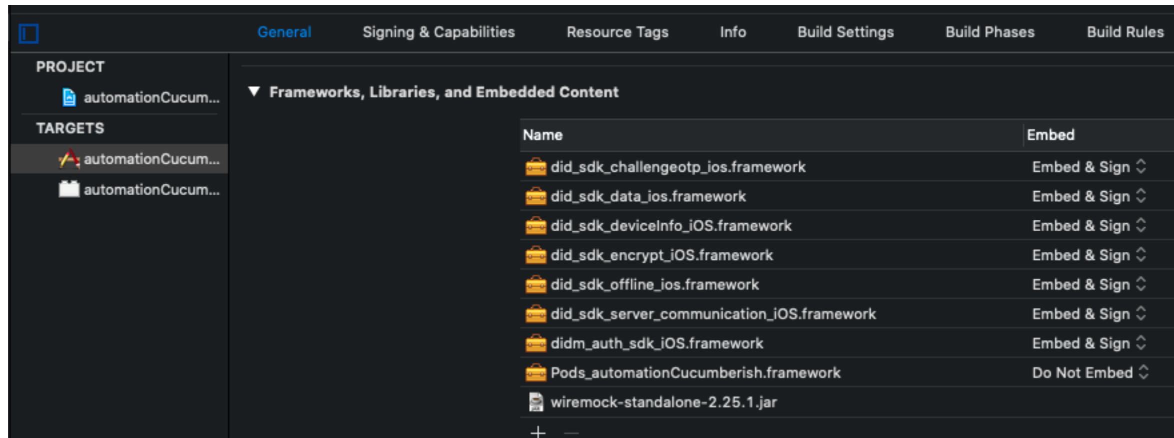


Figura 4-5.: Target de la aplicación

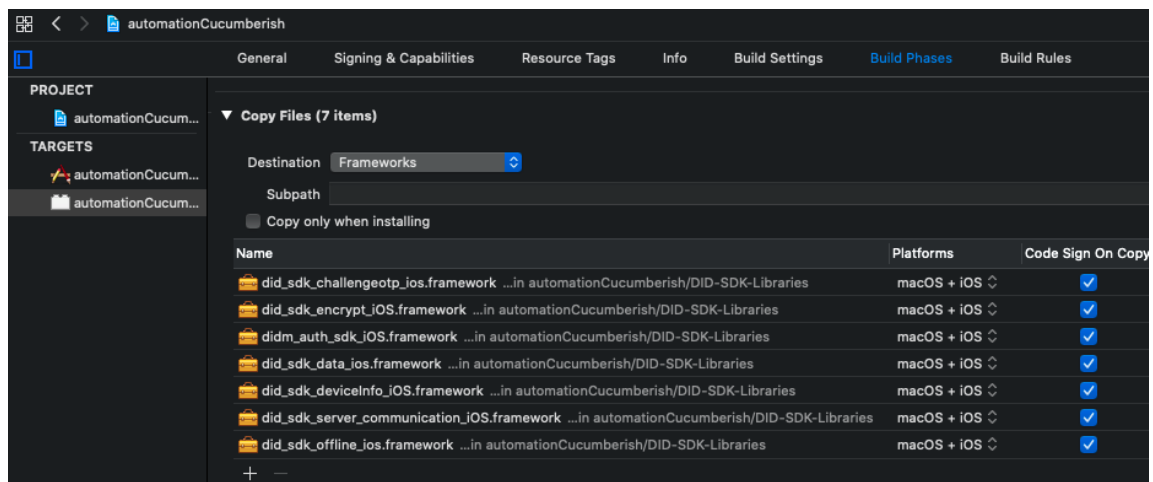


Figura 4-6.: Target de pruebas

Cuando el proyecto de automatización es integrado con el software se puede dar inicio a la **creación y/o codificación de los pasos a ejecutar** previamente definidos en cada feature como se indica en 4.7.

4.7. Implementación de los pasos a ejecutar en cada escenario

Los steps se crean teniendo en cuenta las funcionalidades a proba al igual que los feature. Por tanto, se crea una clase general con los pasos más comunes entre funcionalidades (Background) y por cada funcionalidad se crea una nueva carpeta dentro del directorio de steps y dentro de dicha carpeta se crean las clases necesarias para la ejecución de los mismos. De esta manera, se tiene por ahora la implementación de dos funcionalidades y por tanto dos carpetas adicionales para los steps:

- **Registration**: donde se ejecutan los escenarios correspondiente al registro de un dispositivo, tanto los escenarios correctos como incorrectos indicados en el feature: `registerDevicesByCode.feature`.
- **Soft OTP**: donde se ejecutan el escenario correspondiente a la actualización del tiempo de vida del OTP generado para un dispositivo previamente registrado indicado en el feature: `lifetimeOTP.feature`.

Así mismo, contiene la configuración de variables de entorno o constantes requeridas para la ejecución de los escenarios dentro de un directorio a parte (`config`) y métodos comunes para la ejecución del SDK, llamados APIs, administración de llaves, entre otros.

4.7.1. Implementación en Android

La implementación de los step definitions en java se hace dentro de la clase «`StepDefinitions.java`» como se observa en los Anexos A.5.1.

4.7.2. Implementación en iOS

La implementación de los step definitions en swift se hace dentro de la clase «`StepDefinitions.swift`» como se observa en los Anexos A.5.2.

Con el proyecto creado, configurado y con las respectivas implementaciones de los escenarios definidos, se procede a la **ejecución de las pruebas** tanto en dispositivos físicos como emuladores para Android, y dada las condiciones del SDK para iOS únicamente se ejecutan en dispositivos físicos como se indica en 4.8.

4.8. Ejecución de las pruebas

La ejecución de las pruebas se lleva a cabo la siguiente manera en cada plataforma:

4.8.1. Ejecución en Android

Para la ejecución de los escenarios es necesario llevar a cabo el paso a paso que se indican en los Anexos A.6.1 y finalmente la ejecución de las pruebas de aceptación en todos los dispositivos ejecutados de la siguiente manera:

Ejecución de las pruebas en Android

```
./gradlew connectedCheck
```

4.8.2. Ejecución en iOS

Para la ejecución de los escenarios es necesario llevar a cabo el paso a paso que se indican en los Anexos A.6.2 y finalmente la ejecución de las pruebas de aceptación en todos los dispositivos ejecutados de la siguiente manera:

Ejecución de las pruebas en iOS para dispositivos físicos

```
xcodebuild \
-workspace automationCucumberish.xcworkspace \
-scheme automationCucumberish \
-destination 'name=NAME_DEVICE' \
test
```

Dentro de la ejecución de las pruebas existe la posibilidad de crear los respectivos **reportes de las pruebas** ejecutadas con los cuales se puede hacer seguimiento de las mismas para la detección de errores, verificación de tiempos de ejecución, validación de funcionamiento, entre otros. La generación de estos informes se lleva a cabo como se indica en 4.9.

4.9. Generación de informes con los resultados de las pruebas ejecutadas

La generación y visualización de reportes se llevan a cabo de la siguiente manera para cada plataforma:

4.9.1. Generación de informes en Android

→ **Generación de los reportes de Cucumber**
Para generar los reportes de Cucumber en todos los dispositivos ejecutados se debe implementar la siguiente secuencia de comandos:

Generación de reportes de las pruebas ejecutadas en Android

```
./gradlew grantPermissions
./gradlew connectedCheck
./gradlew exportCucumberReports
```

En primer lugar se deben conceder los permisos necesarios para la lectura y escritura de los reportes en cada dispositivo ejecutado, para continuar con la ejecución de las pruebas y finalmente la exportación de los reportes generados en cada dispositivo a un directorio de la máquina del usuario dentro del proyecto: `app/build/reports/cucumber/`.

→ **Estructura generación de reportes de Cucumber**

Los reportes son generados inicialmente dentro de un directorio del dispositivo móvil y deben ser copiados y movidos a un directorio de acceso rápido al usuario. De esta manera los reportes son almacenados en el directorio del proyecto como se muestra en la Figura 4-7.

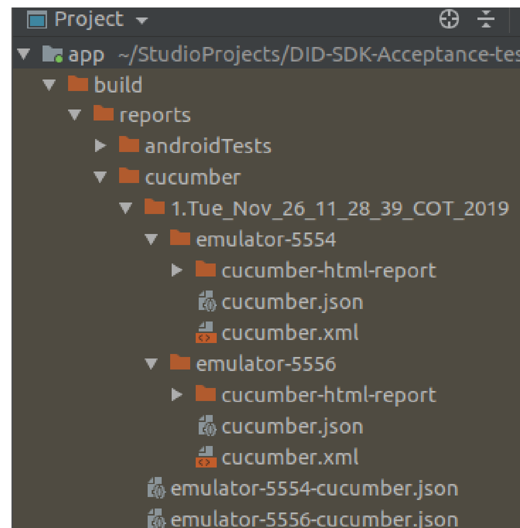


Figura 4-7.: Estructura generación de reportes de Cucumber.

En **app/build/reports** se crea la carpeta **cucumber** dentro de la cual se crea otra carpeta con la fecha y hora de ejecución de las pruebas y se incluyen carpetas con cada uno de los emuladores ejecutados, cada una con los reportes en los formatos `JSON`, `JUNIT` y `HTML`. Adicionalmente dentro de la carpeta **cucumber** se generan los últimos reportes en formato `JSON` de cada emulador para la captura de dicha información desde el **Jenkins** y la posterior visualización de estos reportes en dicha herramienta.

→ **Reportes de Cucumber generados**

Los reportes generados de Cucumber en el pipeline de Jenkins mediante el **Plugin de Cucumber Reports** permiten visualizar cada ejecución de Jenkins que genera el respectivo reporte con la fecha de ejecución (Figura **4-8**).

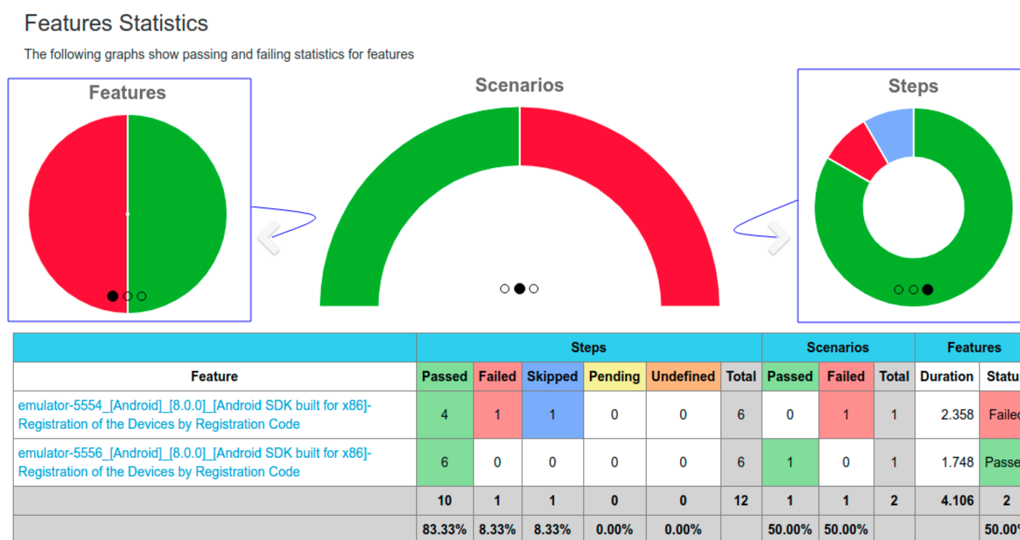
Cucumber Report | Jenkins | Previous results | Latest results | Features | Tags | Steps | Trends | Failures

Project	Number	Date
master	34	20 Nov 2019, 20:07

Figura 4-8.: Cucumber Reports.

Adicionalmente:

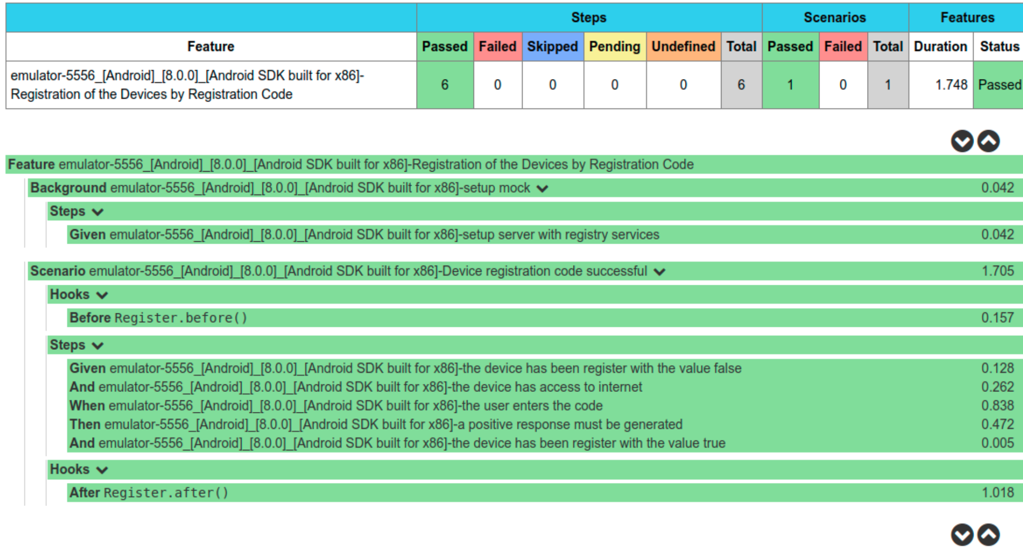
- **Features (Features Statistics)**: Muestra los resultados generales de la ejecución, los pasos, escenarios y features que se implementaron correctamente o que fallaron en cada dispositivo móvil por medio de cantidades y porcentajes (Figura **4-9**).



Feature	Steps						Scenarios			Features	
	Passed	Failed	Skipped	Pending	Undefined	Total	Passed	Failed	Total	Duration	Status
emulator-5554_[Android]_[8.0.0]_[Android SDK built for x86]-Registration of the Devices by Registration Code	4	1	1	0	0	6	0	1	1	2.358	Failed
emulator-5556_[Android]_[8.0.0]_[Android SDK built for x86]-Registration of the Devices by Registration Code	6	0	0	0	0	6	1	0	1	1.748	Passed
	10	1	1	0	0	12	1	1	2	4.106	2
	83.33%	8.33%	8.33%	0.00%	0.00%		50.00%	50.00%			50.00%

Figura 4-9.: Features Statistics.

- **Feature Report**: Al dar click a un feature específico dentro de **Features Statistics** se muestran los resultados específicos de la ejecución de dicho feature en determinado dispositivo móvil, junto con los pasos y escenarios que se implementaron correctamente o que fallaron en dicho dispositivo de manera más detallada (Figura **4-10**).



Feature Report

	Steps						Scenarios			Features	
Feature	Passed	Failed	Skipped	Pending	Undefined	Total	Passed	Failed	Total	Duration	Status
emulator-5556_[Android]_[8.0.0]_[Android SDK built for x86]-Registration of the Devices by Registration Code	6	0	0	0	0	6	1	0	1	1.748	Passed

Feature emulator-5556_[Android]_[8.0.0]_[Android SDK built for x86]-Registration of the Devices by Registration Code

Background emulator-5556_[Android]_[8.0.0]_[Android SDK built for x86]-setup mock 0.042

Steps

Given emulator-5556_[Android]_[8.0.0]_[Android SDK built for x86]-setup server with registry services 0.042

Scenario emulator-5556_[Android]_[8.0.0]_[Android SDK built for x86]-Device registration code successful 1.705

Hooks

Before `Register.before()` 0.157

Steps

Given emulator-5556_[Android]_[8.0.0]_[Android SDK built for x86]-the device has been register with the value false 0.128
And emulator-5556_[Android]_[8.0.0]_[Android SDK built for x86]-the device has access to internet 0.262
When emulator-5556_[Android]_[8.0.0]_[Android SDK built for x86]-the user enters the code 0.838
Then emulator-5556_[Android]_[8.0.0]_[Android SDK built for x86]-a positive response must be generated 0.472
And emulator-5556_[Android]_[8.0.0]_[Android SDK built for x86]-the device has been register with the value true 0.005

Hooks

After `Register.after()` 1.018

Figura 4-10.: Feature Report.

- **Tags (Tags Statistics)**: En caso de que los features contengan Tags estos serán mostrados en esta sección (Figura 4-11).

Tags Statistics

The following graph shows passing and failing statistics for tags

	Steps						Scenarios			Features	
Tag	Passed	Failed	Skipped	Pending	Undefined	Total	Passed	Failed	Total	Duration	Status
@checkout	10	1	2	1	2	16	1	1	2	231ms	Failed
@fast	6	0	0	0	0	6	1	0	1	139ms	Passed
@featureTag	6	0	0	0	0	6	1	0	1	139ms	Passed
3	**22**	**1**	**2**	**1**	**2**	**28**	**3**	**1**	**4**	**509ms**	
	78.57%	**3.57%**	**7.14%**	**3.57%**	**7.14%**		**75.00%**	**25.00%**			**25.00%**

Figura 4-11.: Tags Statistics.

- **Steps (Steps Statistics)**: Muestra el paso a paso para la ejecución de un feature, junto con cantidades de ocurrencias, tiempos de duración, ejecuciones correcta y fallidas, entre otros (Figura 4-12).

Steps Statistics

The following graph shows step statistics for this build. Below list is based on results. step does not provide information about result then is not listed below. Additionally @Before and @After are not counted because they are part of the scenarios, not steps.

Implementation	Occurrences	Average duration	Max duration	Total durations	Ratio
BackgroundSteps.setRegirtryServices()	2	0.046	0.050	0.093	100.00%
Register.after()	2	0.510	1.018	1.021	50.00%
Register.before()	2	0.208	0.258	0.416	100.00%
Register.validateInternetConnection()	2	0.216	0.262	0.433	100.00%
Register.validateNoDevicesRegistration(boolean)	4	0.076	0.170	0.304	75.00%
RegisterDevicesByCodeSteps.sendURL()	2	0.675	0.838	1.351	100.00%
RegisterDevicesByCodeSteps.validateRegistrationSuccessful()	2	0.962	1.452	1.924	50.00%
7	**16**	**0.346**	**1.924**	**5.544**	**Totals**

Figura 4-12.: Steps Statistics.

- **Tag Report**: Al dar click a un tag específico dentro de **Tags Statistics** se muestran los resultados específicos de la ejecución de dicho tag en determinado dispositivo móvil, junto con los pasos y escenarios que se implementaron correctamente o que fallaron en dicho dispositivo de manera más detallada (Figura 4-13).

Tag Report

	Steps						Scenarios			Features	
Tag	Passed	Failed	Skipped	Pending	Undefined	Total	Passed	Failed	Total	Duration	Status
@checkout	10	1	2	1	2	16	1	1	2	231ms	Failed

View Feature *1st feature*
@fast @featureTag @checkout
Scenario Outline Account has <sufficient funds>
Account holder withdraws cash

View Feature *Second feature*
@checkout
Scenario Outline Account may not have sufficient funds
Account holder withdraws more cash

Hooks

Before MachineFactory.findCachMachine() 010ms
Before MachineFactory.wait() 001ms

Steps

Given the account balance is 100 000ms
And the card is valid 000ms
And the machine contains 100 000ms
When the Account Holder requests 20 000ms
Then the ATM should dispense 20 000ms
And the account balance should be 90 001ms

Figura 4-13.: Tag Report.

- **Trends (Trends Stastistics)**: Se muestran gráficas en el tiempo de la ejecución llevada a cabo para cada feature, escenarios, pasos y duraciones de ejecución de los mismos (Figura 4-14).

Figura 4-14.: Trends Stastistics.

- **Failures (Failures Overview)**: Muestra el detalle de las fallas presentadas en los pasos implementados dentro de los features (Figura 4-15).

Failures Overview

The following summary displays scenarios that failed.

Feature: emulator-5554_[Android]_[8.0.0]_[Android SDK built for x86]-Registration of the Devices by Registration Code

Scenario emulator-5554_[Android]_[8.0.0]_[Android SDK built for x86]-Device registration code successful 2.307

Hooks

Before `Register.before()` 0.258

Steps

Given emulator-5554_[Android]_[8.0.0]_[Android SDK built for x86]-the device has been register with the value false 0.170

And emulator-5554_[Android]_[8.0.0]_[Android SDK built for x86]-the device has access to internet 0.170

When emulator-5554_[Android]_[8.0.0]_[Android SDK built for x86]-the user enters the code 0.513

Then emulator-5554_[Android]_[8.0.0]_[Android SDK built for x86]-a positive response must be generated 1.452

org.junit.ComparisonFailure: Bad Response: Device registration fail expected:<Device [successfully registered]> but was:<Device [registration fail]>

```
org.junit.ComparisonFailure: Bad Response: Device registration fail expected:<Device [successfully registered]> but was:<Device
        at org.junit.Assert.assertEquals(Assert.java:115)
        at net.easysol.automation.did.cucumber.steps.RegisterDevicesByCodeSteps.validateRegistrationSuccessful(RegisterDevicesBy
        at ✱.Then a positive response must be generated(features/registerDevicesByCode.feature:10)
```

And emulator-5554_[Android]_[8.0.0]_[Android SDK built for x86]-the device has been register with the value true 0.000

Hooks

After `Register.after()` 0.003

Error message

```
java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
        at java.util.ArrayList.get(ArrayList.java:437)
        at net.easysol.automation.did.cucumber.steps.RegisterLogic.removeDevice(RegisterLogic.java:92)
        at net.easysol.automation.did.cucumber.steps.Register.after(Register.java:50)
        at java.lang.reflect.Method.invoke(Native Method)
        at cucumber.runtime.Utils$1.call(Utils.java:40)
        at cucumber.runtime.Timeout.timeout(Timeout.java:16)
        at cucumber.runtime.Utils.invoke(Utils.java:34)
        at cucumber.runtime.java.JavaHookDefinition.execute(JavaHookDefinition.java:60)
        at cucumber.runtime.Runtime.runHookIfTagsMatch(Runtime.java:224)
        at cucumber.runtime.Runtime.runHooks(Runtime.java:212)
        at cucumber.runtime.Runtime.runAfterHooks(Runtime.java:206)
        at cucumber.runtime.model.CucumberScenario.run(CucumberScenario.java:46)
        at cucumber.runtime.model.CucumberFeature.run(CucumberFeature.java:165)
        at cucumber.runtime.android.CucumberExecutor.execute(CucumberExecutor.java:112)
        at cucumber.api.android.CucumberInstrumentationCore.start(CucumberInstrumentationCore.java:88)
        at net.easysol.automation.did.cucumber.runner.CucumberTestRunner.onStart(CucumberTestRunner.java:39)
        at android.app.Instrumentation$InstrumentationThread.run(Instrumentation.java:2074)
```

Figura 4-15.: Failures Overview.

4.9.2. Generación de informes en iOS

Para la generación de reportes en iOS es importante seguir el paso a paso que se indican en los Anexos A.7 los cuales finalmente se generan por medio de la ejecución de:

Ejecución pruebas y generación de reportes en ruta específica

```
xcodebuild \
-workspace PROJECT_NAME.xcworkspace \
-scheme PROJECT_NAME \
-destination 'platform=iOS Simulator,name=iPhone 11 Pro Max,OS=13.3' \
test|XCPRETTY_JSON_FILE_OUTPUT=PATH/NAME.json xcpretty -f
    'xcpretty-json-formatter'  \
-r html -o PATH/NAME.html  \
-r junit -o PATH/NAME.xml
```

Visualización de reportes

El reporte se genera dentro de un archivo **html** que permite visualizar los features y escenarios ejecutados con el tiempo de implementación de cada uno. Muestra las escenarios ejecutados correctamente y los errores presentados en aquellos que tienen alguna falla como se observa en la Figura **4-16**.

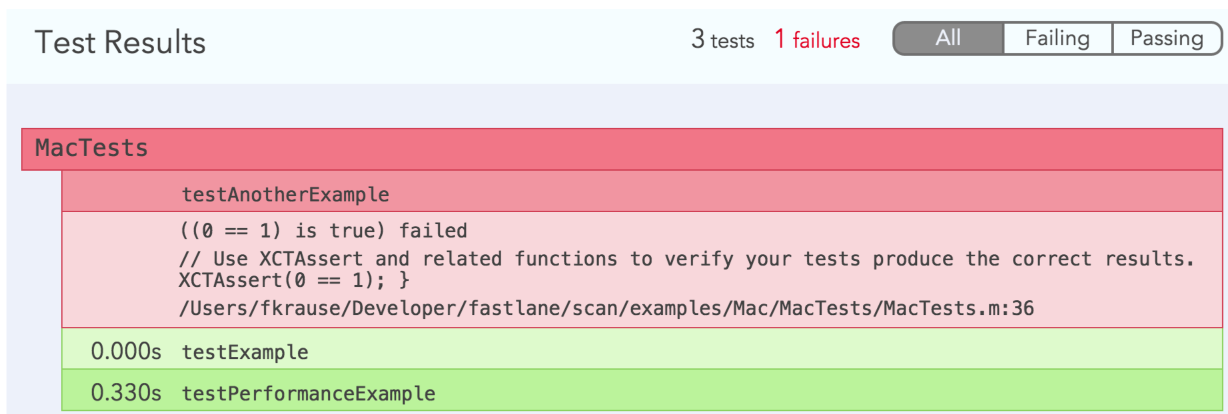


Figura 4-16.: Reporte generado en la ejecución de pruebas en iOS.

Finalmente, garantizando el correcto funcionamiento y ejecución de cada sistema de pruebas automatizado se procede a implementarlo dentro de un **entorno de integración continua** como se muestra en 4.10.

4.10. Implementación del sistema de pruebas automatizado en un entorno de integración continua

Para la ejecución automática de las pruebas se implementaron dos pipeline para cada plataforma de la siguiente manera:

4.10.1. Implementación en Android

El pipeline de Android se muestra en los Anexos A.8.1 y se compone de las siguientes etapas:

- **Etapa 1 - stage('Clear gradle cache')**:
  Elimina el caché o artefactos de las pruebas previamente ejecutadas en el dispositivo.
- **Etapa 2 - stage('Build')**:
  Construye y compila las librerías necesarias para la ejecución de pruebas.

Para el caso de los emuladores: Ejecuta los emuladores disponibles en donde se implementarán las pruebas.

- **Etapa 3 - stage('Testing')**:
  Consulta los dispositivos físicos disponibles conectados al servidor para la implementación de las pruebas y ejecución de las pruebas tanto en los emuladores como en los dispositivos físicos.

- **Acciones Post**:
  Exportación de información a punto único dentro de la máquina de Jenkins. Generación de reportes para fácil lectura.

4.10.2. Implementación en iOS

El pipeline de iOS se muestra en los Anexos A.8.2 permite establecer las variables de ambiente como puerto de configuración del mock y se compone de las siguientes etapas:

- **Etapa 1 - stage('Copy Artifacts')**:
  Copia los artefactos del SDK dentro del proyecto de pruebas. Dichos artefactos se generan dentro del mismo Jenkins pero en otros pipeline. Por tanto, se copian los artefactos generados en la última ejecución correcta de cada proyecto del SDK.

- **Etapa 2 - stage('Start Mockup Server')**:
  Ejecuta el Mock con los stubs previamente definidos en el proyecto: `http://wiremock.org/docs/running-standalone/`

- **Etapa 3 - stage('Enable POD to Cucumberish')**:
  Instala las librerías y/o herramientas requeridas para la ejecución del proyecto de pruebas tales como cucumberish contenidas dentro del archivo **Podfile**.

- **Etapa 4 - stage('Executing scenarios')**:
  Consulta los dispositivos físicos disponibles conectados al servidor para la implementación de las pruebas. Genera una lista de dispositivos para definir en cuales serán ejecutadas las pruebas. Ejecución de las pruebas en los dispositivos físicos de la lista y a la vez genera los reportes de cada ejecución.

- **Acciones Post**:
  Exporta los archivos **json** y **html** de los reportes de prueba generados dentro de cada ejecución del pipeline y elimina directorios/archivos no requeridos y permite la visualización de los reportes de las pruebas.

Para llevar a cabo la automatización completa de una ejecución de software, tanto el software como el sistema de pruebas automatizado deben ser implementados e integrados dentro del entorno de integración continua. De esta manera, la **integración del SDK-DID y el sistema de pruebas automatizado dentro de un entorno de integración continua** se lleva a cabo como se indica en 4.10.3.

4.10.3. Integración del SDK-DID y el sistema de pruebas automatizado dentro de un entorno de integración continua

En la Figura **4-17**, se puede observar el flujo del proceso completo, que involucra la integración del SDK-DID y el sistema de pruebas automatizado en un entorno de integración continua, desde que se ejecuta un evento CVS (push/merge request) hasta la generación de reportes y/o envío de notificaciones; lo cual se explica con mayor detalle más adelante.

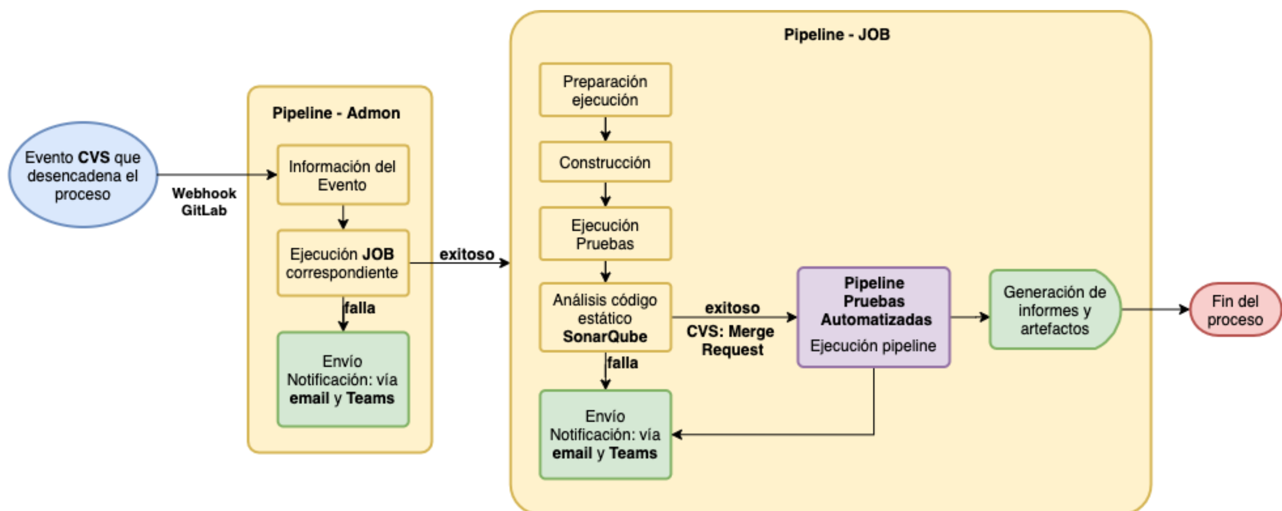


Figura 4-17.: Arquitectura Jenkins

Dado que actualmente, cada proyecto-librería del SDK, es ejecutado de forma individual, sin tener en cuenta tipo de evento CVS, pruebas automatizadas, análisis de cobertura, entre otros; de la arquitectura definida en la Figura **4-17**, se determina implementar mejoras en el **proceso de integración continua del SDK** como se indica a continuación:

→ **Proceso de CI en cada proyecto-librería del SDK**

Cada proyecto-librería del SDK contiene su propio pipeline encargado de la construcción, ejecución de pruebas, generación de reportes y artefactos. Sin embargo, para lograr un óptimo ambiente de integración continua no es suficiente con crear un pipeline, es necesario lograr que éste interactúe adecuadamente con los plugins y/o herramientas de tal forma que permita hacer un proceso completamente automático, que indique lo más pronto posible el estado de cada ejecución y un registro detallado de la misma para fallar lo más pronto posible y por tanto dar solución más rápida a los problemas presentados.

De esta manera, en el equipo se definen dos etapas dentro del proceso de integración continua:

- **Etapa de desarrollo PUSH**: Cuando se desarrolla una funcionalidad específica el desarrollo se realiza sobre una rama única para dicha funcionalidad. Por tanto, dentro del repositorio de código se ejecutarán únicamente acciones **PUSH** para dicha rama.

  En el momento que el pipeline de dicha funcionalidad específica sea ejecutado correctamente; es decir, las pruebas unitarias, de integración, de contrato, el análisis de cobertura, estén en un estado correcto se procede a llevar a cabo la segunda etapa de CI.

- **Etapa de MERGE-REQUEST**: Cuando la funcionalidad específica está verificada y aceptada para ser expuesta en la rama principal se realiza un **MERGE-REQUEST** a la rama **development/master**. Esta segunda etapa es una réplica de la primera etapa pero con unos pasos adicionales: la integración de todas las funcionalidades del SDK para llevar a cabo la ejecución de las pruebas de aceptación automatizadas.

Sin embargo, para poder ejecutar el proceso de CI planteado con sus dos etapas es necesario llevar a cabo:

1. **Implementación WEB HOOKS en GitLab**:
   En el GitLab dentro de cada proyecto en **Settings** → **Web Hooks**, se configura la URL de Jenkins (`http://USER:TOKEN@URL_JENKINS/generic-webhook-trigger/invoke`) que lanzará el pipeline cuando ocurra alguno de los eventos seleccionados en el Web Hook como se observa en la Figura **4-18**.

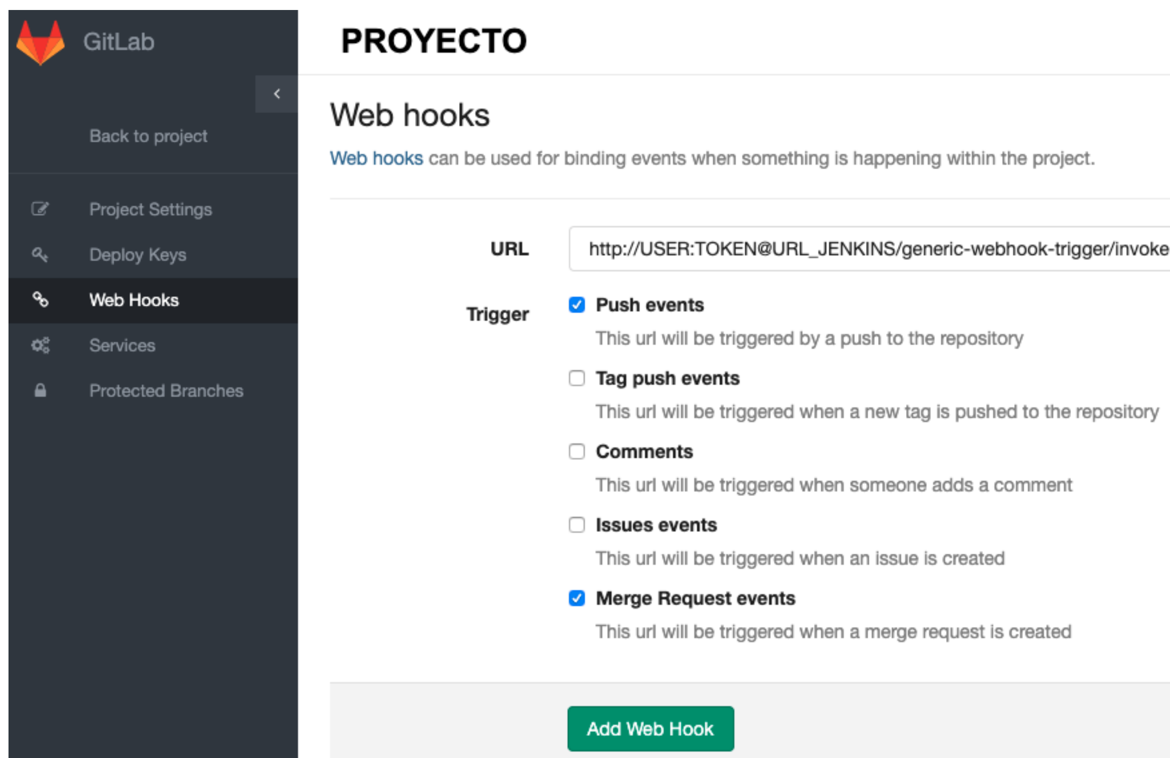


Figura 4-18.: Configuración del Web Hook dentro del proyecto de GitLab.

Donde finalmente al establecer los eventos y la URL correspondiente se podrá lanzar una prueba que ejecute el pipeline asociado al Web Hook con el botón Test Hook (Figura **4-19**).

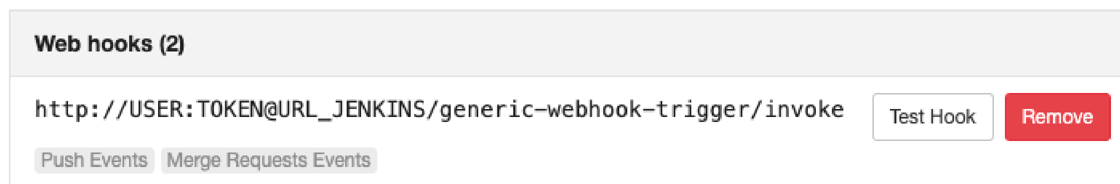


Figura 4-19.: Web Hook configurado.

Así mismo, en Jenkins debe instalarse el Plugin: **Generic Webhook Trigger Plugin** y dentro del pipeline debe activarse y configurarse el Web Hook para que éste logre ejecutarse cuando en el GitLab se lleve a cabo alguno de los eventos configurados desde la configuración del proyecto de Jenkins (Figura **4-20**) o dentro del Jenkinsfile de la siguiente manera:

Jenkinsfile: Configuración Web Hook

```
triggers {
  GenericTrigger(
   genericVariables: [
       [key: ’object_kind’, value: ’$.object_kind’],
       [key: ’user_name’, value: ’$.user_name’],
       [key: ’user_email’, value: ’$.user_email’],
       [key: ’repository_name’, value: ’$.repository.name’],
       [key: ’total_commits_count’, value: ’$.total_commits_count’],
       [key: ’commit_messages’, value: ’$.commits[*].message’],
       [key: ’ref’, value: ’$.ref’]
   ],
          printContributedVariables: true,
          printPostContent: true,
          silentResponse: false
  )
}
```

En el **GenericTrigger - genericVariables** se configuran las variables que se desean extraer del `JSON` entregado por GitLab. En este caso en **key** se nombra como se desee la variable y en **value** se indica específicamente el parámetro a extraer del `JSON`. Por ejemplo, el `JSON` completo se extrae como ‘`$..*`’ pero en dado caso de requerir el usuario quien ejecutó el PUSH o el MERGE_REQUEST sería de la siguiente manera: ‘`$.user_name`’. Es así cómo se extraen los parámetros que indican el tipo de evento realizado (PUSH o MERGE_REQUEST), el correo y nombre de la persona quien ejecutó tal evento, el nombre del repositorio, rama del repositorio y commits realizados en dicho evento.

Generic Webhook Trigger

Is triggered by HTTP requests to **http://JENKINS_URL/generic-webhook-trigger/invoke**

There are example configurations in the Git repository.

You can fiddle with JSONPath here. You may also want to checkout the syntax here.

You can fiddle with XPath here. You may also want to checkout the syntax here.

You can fiddle with regular expressions here. You may also want to checkout the syntax here.

Figura 4-20.: Habilitar Web Hook en el proyecto de Jenkins.

2. **Integración con notificaciones por correo electrónico**:
   Las notificaciones por correo electrónico son posibles gracias al Plugin: **Email Extension** y se realiza de la siguiente manera:

Jenkinsfile: Notificación por correo electrónico

```
def notifyEmail() {
    emailext (
      subject: 'ASUNTO',
      body: 'MENSAJE'
      to: 'CORREO_ELECTRONICO'
    )
}
```

3. **Integración con TEAMS**:
   Las notificaciones por Teams son posibles por el Plugin: **Office 365 Connector**. Para poder establecer comunicación entre el Jenkins y Teams, desde Teams se crea el canal que recibirá estas notificaciones y se le asocia un conector Jenkins (Figura 4-21).

Figura 4-21.: Agregar conector Jenkins en Teams y asociar un canal.

Adicionalmente, se le configura un nombre específico al conector y al final de la configuración se genera una URL (Figura 4-22) la cual se configura dentro del Jenkinsfile de la siguiente manera:

Jenkinsfile: Notificación por Teams

```
def notifyTeams(){
      office365ConnectorSend webhookUrl: 'URL_TEAMS',
            factDefinitions: [[name: "PARAM_1", template: 'VALOR_PARAM_1'],
                              [name: "PARAM_2", template: 'VALOR_PARAM_2'],
                              [name: "PARAM_3", template: 'VALOR_PARAM_3']]
}
```

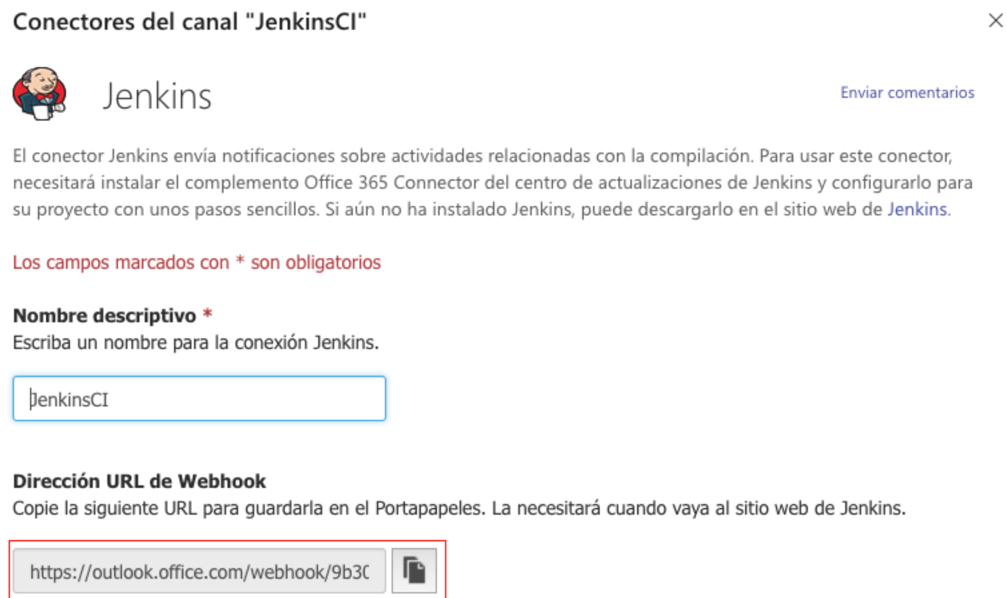


Figura 4-22.: Configuración canal para el conector de Jenkins, y generación de la URL.

4. **Pipeline Administrador**:
   Dado que el Web Hook ejecuta todo pipeline que tenga configurado se crea un pipeline administrador que se encargue de ejecutar el proyecto específico sobre el cual se ejecuta algún evento. De esta manera se tiene el Jenkinsfile para el pipeline Administrador que se muestra en los Anexos B.1.

   Desde dicho pipeline se obtienen los datos del evento realizado en GitLab para que el administrador tenga conocimiento de qué JOB debe ejecutar (`getJob`), por medio del método `executeJob` que se encarga de configurar los parámetros requeridos para la ejecución del JOB obtenidos desde el GitLab y lanzarlo. Finalmente, si la ejecución del JOB administrador falla se envía una notificación al correo electrónico de la persona que ejecutó el evento (`notifyEmail`) y a Teams (`notifyTeams`).

5. **Integración con JaCoCo para reportes de cobertura en Android**:
   La integración con JaCoCo es posible por el Plugin: **JaCoCo**. En primer lugar, para lograr la integración con JaCoCo con cada proyecto Android se debe agregar el Plugin de JaCoCo dentro del `build.gradle` de la aplicación y se debe agregar la tarea gradle `jacocoTestReport` encargada de generar los reportes de cobertura de las pruebas como se muestra en los Anexos B.2 en el «`build.gradle`».

Finalmente dentro del Jenkinsfile, se ejecutaría JaCoCo de la siguiente manera:

Jenkinsfile: Ejecución JaCoCo en Android

```
sh './gradlew jacocoTestReport'
step([$class: 'JacocoPublisher',
      execPattern: '**/build/jacoco/*.exec'
       ])
```

6. **Integración con SonarQube para análisis de cobertura**:
   La integración con SonarQube es posible por el Plugin: **SonarQube Scanner** y la integración con los Webhooks de SonarQube es posible por el Plugin: **Sonar Quality Gates Plugin** que permite saber al pipeline si el análisis de código estático de SonarQube falló y hará que el pipeline falle. Si este último Plugin no se ejecuta en el pipeline y algo falla en el SonarQube el pipeline no fallará.

   1) **Implementación en Android** :
      La integración con Sonarqube se lleva cabo como se indica en los Anexos B.3.1. En el Jenkinsfile se ejecuta de la siguiente manera:

      Jenkinsfile: Ejecución análisis SonarQube en Android

```
def executeSonar(){
        withSonarQubeEnv('MAQUINA_SONAR') {
            echo 'Test Coverage'
            sh './gradlew sonarqube'
        }
        timeout(time: 5, unit: 'MINUTES') {
           waitForQualityGate abortPipeline: true
        }
}
```

   2) **Implementación en iOS** :
      La integración con Sonarqube se lleva cabo como se indica en los Anexos B.3.2. En el Jenkinsfile se ejecuta de la siguiente manera:

Jenkinsfile: Ejecución análisis SonarQube en iOS

```
def executeCoverage(){
    sh 'bash ${SONAR_SCANNER}/xccov-to-sonarqube-generic.sh
    PATH_XCRESULT/*.xcresult > coverage.xml'
        withSonarQubeEnv('MAQUINA_SONAR') {
    sh '${SONAR_SCANNER}/bin/sonar-scanner
    -Dproject.settings=sonar.properties'}
    timeout(time: 5, unit: 'MINUTES') { waitForQualityGate
    abortPipeline: true }
}
```

Finalmente, la **implementación del sistema de pruebas automatizado** debe ser incluida dentro de cada **ejecución de cada proyecto/librería del SDK** como se indica a continuación:

→ **Ejecución del pipeline de pruebas desde cada proyecto/librería del SDK**

Para ejecutar el pipeline de pruebas desde otros proyectos se debe agregar el siguiente **stage** al **Jenkinsfile** de cada proyecto-librería del SDK:

Stage de pruebas - Jenkinsfile

```
stage('Run Acceptance Tests') {
    echo 'Run Acceptance Tests Pipeline'
    when{ expression { BRANCH_NAME ==~ /(development|master)/}}
    steps { build job: 'JOB/master', propagate: true, wait: true }
}
```

De esta manera, se ejecutaría el pipeline del **JOB** correspondiente a cada sistema de pruebas automatizado, tanto para Android (JOB=Android-Automation-sdk-acceptance-test-android), como para iOS (JOB=iOSAcceptanceTests) en la rama **master**, y será ejecutado en cualquier proyecto que tenga este **stage**, sólo si el pipeline del proyecto-librería ejecutado corresponda a la rama **master** o **development**.

En conclusión, la renovación de una metodología que integre nuevas herramientas que permitan la integración continua y la ejecución de un sistema de pruebas automatizado, es una tarea que requiere de la investigación en cuanto a configuración, desarrollo y ejecución de dichas herramientas y metodologías dentro de diferentes plataformas. También es importante y necesario llevar a cabo una implementación que permita definir si es posible llevarlas a cabo dentro del software, cómo se llevan a cabo, la complejidad, ventajas y desventajas de hacerlo, entre otros. Además, que con la implementación es posible obtener resultados que permitan una comparación de un antes y un después con lo cual se puede rectificar o no la importancia de llevar a cabo dichas mejoras.

Adicionalmente, es importante documentar los procesos llevados a cabo de forma clara y concisa paso a paso de tal manera que ésta pueda ser replicable y consultada a futuro para poder seguir innovando y mejorando los procesos y metodologías internas.

5. Resultados

Al llevar a cabo la automatización de un sistema de pruebas dentro de un entorno de integración continua en DetectID ™ móvil se obtiene un ciclo de desarrollo de software más eficiente, rápido, confiable, seguro y claro.

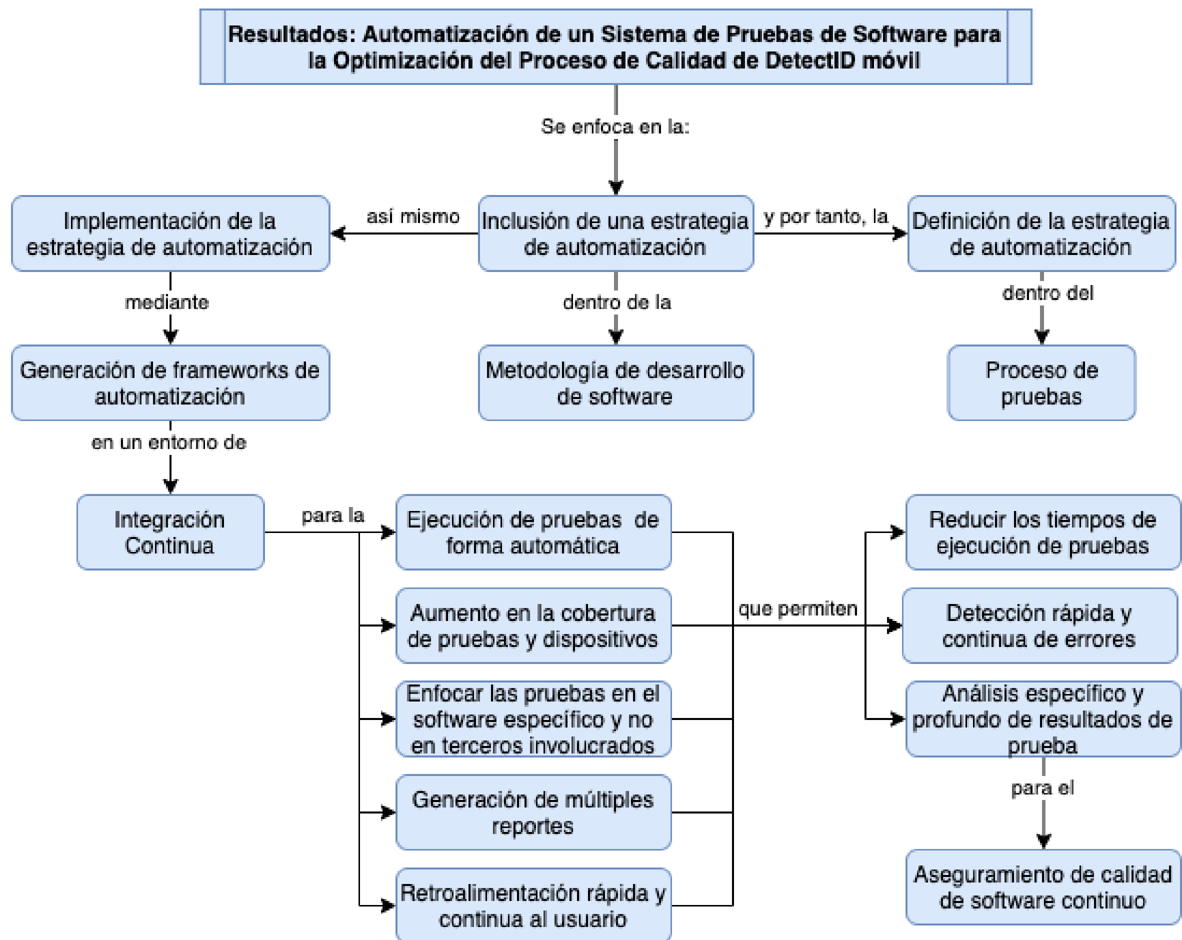


Figura 5-1.: Resultados: Automatización de un Sistema de Pruebas de Software para la Optimización del Proceso de calidad de DetectID™ móvil.

Como se puede observar en la Figura 5-1 en el presente capítulo se indican los resultados obtenidos que demuestran la importancia de la implementación del presente proyecto en el ciclo de software de DetectID ™ móvil por medio de la comparación del estado inicial y actual de dicho ciclo en cuanto a:

- Metodología de desarrollo.
- Estrategia de pruebas.
- Ejecución y cobertura de las pruebas automatizadas.
- Tiempos de ejecución de las pruebas.
- Integración continua.
- Generación de reportes.
- Detección de errores.
- Generación de notificaciones.
- Herramientas utilizadas.
- Plataformas y lenguajes soportados.

De esta manera, en las siguientes secciones se pueden observar los resultados obtenidos en cuanto a las características mencionadas anteriormente dentro del ciclo de software de DetectID ™ móvil.

5.1. Metodología de desarrollo

En la Tabla 5-1 se muestra el estado inicial y actual de la metodología de desarrollo de DetectID ™ móvil. La actual incluye los procesos de planeación, diseño, desarrollo y ejecución de las pruebas automatizadas y la implementación de mejoras en el proceso de integración continua.

Por tanto, el presente proyecto promueve la inclusión de una estrategia de automatización de pruebas dentro de la metodología de desarrollo de software desde de la etapa de planeación hasta la etapa de ejecución dentro de un entorno de integración continua, que permite mejorar la estrategia y el proceso de pruebas dentro del ciclo de desarrollo de software de DetectID ™ móvil.

Tabla 5-1.: Resultados: Metodología de desarrollo.

<table>
<tr><th>Estado inicial</th><th>Estado actual</th></tr>
<tr><td>1. Planeación sobre el desarrollo del software.
2. Análisis de requerimientos y diseño:
▪ Software.
▪ Pruebas.
3. Desarrollo y/o implementación del software.
4. Ejecución pruebas manuales.
5. Entrega al cliente / Release.
6. Integración continua parcial por cada proyecto del SDK.</td><td>1. Planeación sobre el desarrollo de:
▪ Software.
▪ Pruebas automatizadas.
2. Análisis de requerimientos y diseño:
▪ Software.
▪ Pruebas.
3. Desarrollo y/o implementación de:
▪ Software.
▪ Pruebas automatizadas.
4. Ejecución pruebas:
▪ Manuales.
▪ Automatizadas.
5. Entrega al cliente / Release.
6. Integración continua completa para todo el SDK.</td></tr>
</table>

5.2. Estrategia de pruebas

En la Tabla **5-2** se puede observar el estado inicial y actual de la estrategia de pruebas en DetectID ™ móvil. La actual estrategia incluye los procesos requeridos para la continua automatización del sistema de pruebas a partir de la estrategia y escenarios previos.

Por tanto, la revisión y adaptación de los escenarios de prueba permite definir como llevar a cabo la automatización de los mismos y la estrategia de automatización con la cual se determina llevar a cabo el proceso de automatización directamente sobre el SDK sin involucrar interfaz gráfica porque la esencia del software de DetectID ™ móvil está en el SDK.

DetectID ™ es un software integral que involucra servicios o APIs con las cuales el SDK se comunica. Por tal motivo, dentro de la estrategia de automatización se hace el manejo de Mocks que permiten simular las respuestas de esta APIs y eliminar cualquier dependencia del SDK con servicios externos.

Tabla 5-2.: Resultados: Estrategia de pruebas.

Estado inicial	**Estado actual**
▪ Escenarios de pruebas manuales. ▪ Definición del grupo específico de dispositivos físicos sobre los cuales se ejecutarán las pruebas. ▪ Ejecución manual de pruebas E2E (end to end) sobre la interfaz gráfica de una aplicación móvil.	▪ Definición de escenarios de pruebas a automatizar. ▪ Actualización de escenarios de prueba de tal forma que sean automatizables. ▪ Diseño e implementación de frameworks para la automatización del sistema de pruebas en las plataformas Android & iOS. ▪ Automatización del SDK y no de la interfaz gráfica de una aplicación móvil. ▪ Implementación de Mocks para eliminar dependencias con APIs de terceros. ▪ Ejecución del sistema de pruebas en un entorno de CI con un pipeline de Jenkins.

Así mismo, la estrategia de automatización promueve la creación de frameworks o proyectos de automatización del sistema de pruebas. Como el SDK maneja diferentes plataformas se decide llevar a cabo la implementación de esta estrategia en las dos plataformas más implementadas o conocidas como las plataformas nativas para aplicaciones móviles: iOS y Android. Finalmente, estos proyectos de automatización son implementados dentro de un entorno de integración continua que permite un proceso automático desde que se hace un cambio en el software hasta que se genera un reporte del estado de dichos cambios, ya que actualmente este proceso no es completamente automático y no incluye las pruebas automatizadas.

5.3. Ejecución y cobertura de las pruebas

En la Tabla 5-3 se muestra el estado inicial y actual de la ejecución y cobertura de las pruebas en DetectID ™ móvil. Se indican las ventajas de realizar una ejecución automatizada y las desventajas de realizar una ejecución manual.

Por tanto, como un sistema de pruebas automatizado puede ser ejecutado por una máquina las pruebas pueden llevarse a cabo simultáneamente en múltiples dispositivos emuladores de la propia máquina y dispositivos físicos conectados a la misma. Así mismo, reduce la inducción de errores que pueden presentarse en las pruebas manuales por la aplicación móvil sobre la cual se prueba el SDK o por el personal que ejecuta las pruebas.

Tabla 5-3.: Resultados: Ejecución y cobertura de pruebas.

Estado inicial	**Estado actual**
▪ Ejecución manual. ▪ Ejecución en serie en múltiples dispositivos físicos sobre un artefacto generado desde Jenkins que debe ser instalado manualmente. ▪ Ejecución sobre una aplicación móvil no distribuida que puede inducir errores ajenos al SDK y puede afectar la ejecución de las pruebas. ▪ Las pruebas manuales cubren todas las funcionalidades del SDK.	▪ Ejecución automatizada. ▪ Ejecución en serie y paralelo en múltiples dispositivos emuladores y físicos sobre artefactos/librerías generadas y almacenadas en Jenkins y/o Nexus, que son importadas automáticamente con la última versión. ▪ Ejecución sobre el SDK que no requiere de una aplicación móvil. ▪ Las pruebas automatizadas cubren actualmente las funcionalidades de Registro y OTP del SDK, las demás funcionalidades se mantienen con pruebas manuales.

Actualmente el sistema de pruebas automatizado cubre dos funcionalidades del SDK. Sin embargo, con los frameworks creados la tarea de automatización se limita a la implementación de los escenarios, lo que facilita el trabajo a futuro para la continua integración de otros escenarios de prueba al sistema de pruebas automatizado.

La estrategia de pruebas actual involucra procesos manuales y automatizados porque no todos los escenarios de pruebas se encuentran automatizados y es un proceso que debe darse con el tiempo. Por tanto, a medida que se automaticen los escenarios estos dejarán de ser ejecutados manualmente pero aquellos que no este automatizados deberán ser ejecutados de forma manual, de tal forma que no se pierda de vista ningún escenario del sistema de pruebas que pueda detectar algún error dentro del ciclo de desarrollo de pruebas.

5.4. Tiempos de ejecución de pruebas

En la Tabla **5-4** se puede observar el estado inicial y actual del promedio de los tiempos de ejecución de pruebas en DetectID ™ móvil para un único dispositivo teniendo en cuenta los siguientes escenarios de prueba:

- Device registration by code failed, without server connection.
- Device registration by code failed, using wrong code.
- Device registration by code failed, using expired code.

- Device registration by code failed, device already registered.
- Device registration code successful.
- Change the OTP lifetime for the same DetectID with especific seconds.

Tabla 5-4.: Resultados: Tiempos de ejecución de pruebas.

Estado inicial	Estado actual
15 minutos	3 minutos

Por tanto, el sistema de pruebas automatizado permite la reducción de los tiempos de ejecución de pruebas porque se centran en el SDK, no en una aplicación móvil y por la ejecución simultánea en múltiples dispositivos. Sin embargo, los tiempos podrán compararse con mayor detalle mientras se aumente la cobertura en escenarios automatizados porque el sistema de pruebas automatizado debe ser implementado continuamente en el ciclo de desarrollo de software, debe actualizarse constantemente y estar a la vanguardia de las tecnologías y cambios que se presenten en el software para el cual fue creado.

Cabe aclarar que el tiempo de ejecución de las pruebas automatizadas depende de la complejidad de cada escenario y se irá incrementando a medida que se automaticen más escenarios y se tenga mayor cobertura del sistema de pruebas. Sin embargo, en comparación con la ejecución manual los tiempos de ejecución de todo el sistema de pruebas podrían reducirse a más del 50 % a medida que se aumenta la cobertura del sistema de pruebas automatizado.

5.5. Metodología de integración continua

En la Tabla 5-5 se observa el estado inicial y actual del proceso de integración continua en DetectID ™ móvil donde se muestra con más detalle las mejoras implementadas.

Por tanto, dentro de la metodología de integración continua se incluye la ejecución del sistema de pruebas automatizado, se adiciona el análisis de cobertura por medio de Sonarqube y debido al pipeline administrador se tiene mayor información sobre la ejecución de cada proyecto del SDK. De esta manera, el reporte de cada ejecución es más amplio y específico para cada dispositivo sobre los cuales ésta es implementada y retro alimenta directamente al desarrollador involucrado en los cambios llevados a cabo en el software.

Tabla 5-5.: Resultados: Integración Continua.

Estado inicial	Estado actual
▪ Ejecución de cada librería dentro de un pipeline de Jenkins e integración de todas las librerías del SDK. ▪ Ejecución en un único dispositivo. ▪ Retroalimentación a un grupo de usuarios previamente definidos.	▪ Ejecución parametrizada dependiendo de la rama o del tipo de evento CVS llevado a cabo en cada librería dentro de un pipeline de Jenkins e integración de todas las librerías del SDK desde un pipeline administrador. ▪ Ejecución en múltiples dispositivos y generación de reportes por dispositivo. ▪ Información más completa y detallada de cada ejecución. ▪ Análisis de código estático, cobertura y generación de informes de cobertura. ▪ Retroalimentación directa con el usuario que ejecuta un evento. ▪ Ejecución de pruebas automatizadas.

5.6. Generación de reportes

En la Tabla **5-6** se puede observar el estado inicial y actual de la generación de reportes en DetectID ™ móvil. Con la implementación actual se generan más reportes dependiendo del análisis o prueba ejecutada haciendo una distinción de cada dispositivo.

Tabla 5-6.: Resultados: Generación de reportes.

Estado inicial	Estado actual
▪ Reportes de la ejecución de las pruebas unitarias en general.	▪ Reportes de la ejecución de las pruebas unitarias por cada dispositivo. ▪ Reporte de cobertura. ▪ Reportes de la ejecución de las pruebas automatizadas por cada dispositivo.

Por tanto, la ejecución de las pruebas en múltiples dispositivos debe generar un reporte de la mima discriminando dispositivos para saber con exactitud el resultado de las pruebas en cada uno de ellos y en caso de presentarse errores saber exactamente donde se presentó.

Así mismo, la ejecución del análisis de cobertura genera un reporte sobre el cual es posible ver en detalle los resultados del mismo.

5.7. Detección de errores dentro de la ejecución

En la Tabla 5-7 se muestra el estado inicial y actual de la detección de errores en DetectID ™ móvil. Con la implementación actual es posible la detección de una cantidad mayor de errores dados los análisis, pruebas y reportes.

Tabla 5-7.: Resultados: Detección de errores.

Estado inicial	Estado actual
▪ La construcción. ▪ La ejecución de las pruebas unitarias.	▪ La construcción. ▪ La ejecución de las pruebas unitarias de determinado dispositivo. ▪ El análisis de código estático. ▪ La cobertura. ▪ La ejecución de las pruebas automatizadas de determinado dispositivo.

Por tanto, debido a los reportes que son generados durante las pruebas, es posible detectar si se han presentado errores durante una ejecución a partir de la selección de unos criterios de condición exitosa donde se establece cuando una ejecución debe fallar.

Para el caso, si al menos una prueba durante la ejecución falla ésta debe fallar o en dado caso de que la cobertura se encuentre por debajo de un porcentaje establecido por todo el equipo también debe fallar. De esta manera, de deben tener claros las condiciones de falla para que durante el proceso de automatización éstas sean ajustadas correctamente según los criterios del software.

5.8. Generación de notificaciones sobre el estado de la ejecución

En la Tabla 5-8 se puede observar el estado inicial y actual de la generación de notificaciones en DetectID ™ móvil. Con la implementación actual se integran herramientas como Teams para el reporte de notificaciones y se evita la generación de correo spam para personal no involucrado.

Tabla 5-8.: Resultados: Generación de notificaciones.

Estado inicial	Estado actual
▪ Envío de correo electrónico a un grupo de usuarios previamente definidos con el estado actual de la ejecución. Sin embargo, el correo puede ser generado a personal no involucrado y generar spam o podría no ser generado para el usuario relacionado directamente.	▪ Envío de correo electrónico al usuario involucrado directamente con el estado de la ejecución. ▪ Envío de una notificación por Teams a todo el equipo con el estado de la ejecución.

Por tanto, el proyecto permite la integración de herramientas corporativas que generan notificaciones en tiempo real y permiten saber inmediatamente si se ha presentado algún problema durante la ejecución de cada proyecto del SDK.

5.9. Herramientas utilizadas

En la Tabla 5-9 se muestra el estado inicial y actual de las Herramientas utilizadas en DetectID ™ móvil. Con la implementación actual se incorporan más herramientas debido a la automatización de pruebas y las mejoras implementadas en la integración continua.

Tabla 5-9.: Resultados: Herramientas utilizadas.

Estado inicial	Estado actual
▪ GitLab. ▪ Jenkins. ▪ JaCoCo.	▪ GitLab. ▪ Jenkins. ▪ JaCoCo. ▪ SonarQube. ▪ Cucumber y Cucumberish. ▪ Wiremock. ▪ Teams

Por tanto, dados los requerimientos para la implementación de la estrategia de automatización en un entorno de integración continua se requieren de herramientas adicionales que permiten ejecutar efectivamente este proceso dentro una ejecución de cada proyecto del SDK. El proceso para la implementación y el uso de estas herramientas dentro de cada framework o proyecto de automatización es mostrado en el presente documento de forma clara y paso por paso.

5.10. Plataformas y lenguajes soportados

En la Tabla 5-10 se puede observar el estado inicial y actual de las plataformas y lenguajes soportados en el sistema de pruebas automatizado de DetectID ^{TM} móvil. En ambos casos se mantienen las mismas plataformas y lenguajes. Sin embargo, en la implementación actual se ha implementado la automatización para Android y para iOS por lo que en las demás plataformas se siguen realizando procesos manuales.

Tabla 5-10.: Resultados: Lenguajes soportados en el sistema de pruebas automatizado.

Estado inicial	**Estado actual**
Las pruebas manuales se ejecutan sobre: ▪ Android: Java. ▪ iOS: Swift. ▪ Xamarin: C#. ▪ Cordova: Javascript, HTML, CSS.	Las pruebas automatizadas se ejecutan sobre: ▪ Android: Java. ▪ iOS: Swift. Por tanto, actualmente en las demás plataformas se ejecutan pruebas manuales.

Finalmente, el proyecto de automatización involucra actualmente las dos plataformas nativas para aplicaciones móviles: Android y iOS. Las demás plataformas se siguen manteniendo con procesos manuales.

En conclusión, se puede determinar que la implementación del proyecto aporta a DetectID^{TM} móvil:

→ *Integración de una estrategia de automatización del sistema de pruebas dentro de la metodología de desarrollo e inclusión de procesos de automatización dentro de la estrategia de pruebas.*

→ *Ejecución de pruebas directamente sobre el SDK eliminando dependencias con interfaz gráfica para la ejecución de éstas.*

→ *Reducción de aproximadamente el 80 % del tiempo de ejecución de pruebas de los escenarios automatizados teniendo en cuenta la ejecución para un único dispositivo y para las plataformas iOS y Android.*

→ *Implementación de un sistema de pruebas automatizado dentro de un entorno de integración continua.*

→ *Implementación de mejoras dentro del proceso previo de integración continua que permite una ejecución más controlada y automatizada de cada proyecto del SDK, la incorporación del análisis de código estático y cobertura.*

→ *Generación de reportes con información más completa y detallada de cada ejecución que permiten hacer un seguimiento más específico de los errores presentados durante la ejecución.*

→ *Detección de un rango más amplio de errores dentro de cada ejecución con una retroalimentación más rápida y detallada de los mismos.*

→ *Generación de notificaciones con herramientas corporativas que permiten hacer un seguimiento más rápido y continuo de cada ejecución.*

→ *Integración con diferentes tipos de herramientas que permiten la automatización de los procesos de desarrollo, pruebas, análisis de código estático, cobertura, entre otros.*

→ *Automatización del sistema de pruebas dentro de las plataformas iOS y Android y mantenibilidad del sistema de pruebas manual para las plataformas de Xamarin y Cordova.*

6. Conclusiones y Recomendaciones

6.1. Conclusiones

→ El presente proyecto muestra la importancia y las ventajas de integrar una estrategia de automatización de pruebas dentro de la metodología de software, a partir de una investigación previa, la generación de dicha estrategia de automatización y la implementación de la misma en un entorno de integración continua teniendo en cuenta los criterios y necesidades del software de DetectID ™ móvil.

→ La estrategia de automatización de pruebas para DetectID ™ móvil determinó que las pruebas deben ser ejecutadas directamente sobre el SDK eliminando cualquier dependencia con interfaz gráfica que no es la base del software, y en cambio genera mayor costos, complejidad, tiempo y no son deterministas. Así mismo, la estrategia involucra la ejecución de Mocks dentro de las pruebas automatizadas para eliminar dependencias con servicios o APIs de terceros. Finalmente, la estrategia en cuanto a la creación de frameworks o proyectos de automatización se basa en la separación por módulos, lo que permite una mejor mantenibilidad y seguimiento de cualquier proyecto.

→ El presente proyecto demuestra que la implementación de un sistema de pruebas automatizado permite reducir los tiempos de ejecución de pruebas considerablemente, como se indicó en 5.4, para el caso en estudio de aproximadamente un 80 %. Sin embargo, es indispensable que el sistema de pruebas automatizado sea fácil de mantener para actualizarlo paralelamente conforme se actualiza el software, y lograr una mayor reducción de tiempo y costos a futuro, de lo contrario se perderán sus beneficios.

→ En el presente proyecto se muestra la metodología para el diseño e implementación de pruebas automatizadas teniendo claros los requerimientos de la misma, el proceso o flujo del software, los resultados esperados e implementando el paso a paso de dichas pruebas de manera adecuada.

→ La implementación de un sistema de pruebas automatizado permite aumentar la cobertura de dispositivos sobre los cuales se pueden ejecutar las mismas, ya que pueden ser implementadas en emuladores y en dispositivos físicos conectados en cualquier servidor donde se deseen ejecutar las pruebas. En el presente proyecto las pruebas son ejecutadas en múltiples dispositivos iOS y Android en cualquier servidor y lugar debido a los beneficios de la integración continua.

→ La implementación de un ciclo de desarrollo de software dentro de un entorno de integración continua permite obtener retroalimentación del desarrollo de software de forma rápida y continua. Este proceso es sensible a cualquier cambio de código dentro del software que sea llevado al repositorio de código y puede realizar un análisis del mismo según como sea programado, con generación de reportes de los resultados y estado del software, por medio de la generación de notificaciones. La velocidad de respuesta se puede definir a partir de los tiempos de ejecución de pruebas que se obtiene de los reportes generados en los cuales se pueden observar los estados y tiempos de cada ejecución, logs, resultados correctos e incorrectos de cada step dentro de cada escenario de forma detallado y general como se observa en 4.9.

→ Los reportes generados dentro de cada ejecución de las pruebas automatizadas permiten la detección de errores dentro de las funcionalidades desarrolladas de forma ágil y continua dada la claridad y detalle de la información que en ellos se muestra dentro del ciclo de pruebas para determinar dónde, el porqué de un error o falla en el mismo y el estado de su calidad.

→ En el presente proyecto se muestra como llevar a cabo la integración de un software con herramientas como Cucumber/Cucumberish, JaCoCo, Jenkins, Sonarqube, Wiremock, Teams, dentro de un entorno de integración continua. Adicionalmente, en el proyecto se genera un documento con el paso a paso a seguir para la implementación de la metodología y permite observar y definir con más claridad cómo aplicarla en otros proyectos de software de aplicaciones móviles.

→ El proceso de investigación, análisis, diseño, implementación y pruebas de diferentes metodologías y estrategias fue realizada en el presente proyecto y se indica en detalle en el presente documento. Por tanto la generación de los proyectos de automatización llevados a cabo en el presente proyecto permiten que el equipo de pruebas centren sus esfuerzos directamente en el sistema de pruebas, automatizar nuevos escenarios, aumentar la cobertura de pruebas, mantenerlo, mejorarlo, entre otros.

6.2. Recomendaciones

→ Es importante integrar la metodología desarrollada en el presente proyecto en otras plataformas como Xamarin y Cordova.

→ Para aumentar la cobertura del sistema de pruebas automatizado, este puede ser implementado dentro de una granja de dispositivos, la cual se basa en múltiples dispositivos físicos en la nube, y es ofrecido por empresas como SauceLabs , Amazon (AWS Device Farm), Xamarin Test Cloud (para Xamarin), entre otros.

→ La mantenibilidad del sistema de pruebas automatizado es fundamental para conservar los beneficios que trae consigo la automatización, por tanto, si éste sistema no se mantiene con el tiempo conforme el software se va actualizando, el trabajo de automatización se perdería.

→ El sistema de pruebas automatizado debe ser actualizado constantemente, no sólo conforme se realicen cambios en el software, sino adicionalmente, conforme se implementen nuevos cambios en los sistemas operativos y/o plataformas soportadas.

A. Anexo: Implementación frameworks

A.1. Features

A.1.1. Feature: Register devices by code

registerDevicesByCode.feature

```
Feature: Registration Devices by Code

  Background: I must setup a DID Server to solve requests
    Given setup mobile registration service with the company AppGate

  Scenario: Device registration by code failed, without server connection
    Given the device has been registered: false
    And the device has access to server: false
    When the registration is done with a VALID code
    Then a SERVER_ERROR response must be generated
    And the device has been registered: false

  Scenario: Device registration by code failed, using wrong code
    Given the device has been registered: false
    And the device has access to server: true
    When the registration is done with a WRONG code
    Then a WRONG_CODE response must be generated
    And the device has been registered: false

  Scenario: Device registration by code failed, using expired code
    Given the device has been registered: false
    And the device has access to server: true
    When the registration is done with a EXPIRED code
    Then a EXPIRED_CODE response must be generated
    And the device has been registered: false

  Scenario: Device registration by code failed, device already registered
    Given the device has been registered: false
    And the device has access to server: true
    When the registration is done with a REGISTERED code
    Then a DEVICE_ALREADY_REGISTER response must be generated
    And the device has been registered: false

  Scenario: Device registration code successful
    Given the device has been registered: false
    And the device has access to server: true
    When the registration is done with a VALID code
    Then a POSITIVE response must be generated
    And the device has been registered: true
```

Retornar a contenido: «registerDevicesByCode.feature»

A.1.2. Feature: Lifetime OTP

lifetimeOTP.feature

```
Feature: Changing in the OTP lifetime

Background: I must setup a DID Server to solve requests
    Given the authentication factors
    |MOBILE_TOKEN|
    And setup the Authentication Type TOTP with a size: 6
    And the initial OTP lifetime is 3 seconds
    And setup mobile registration service with the company AppGate

Scenario Outline: Change the OTP lifetime for the same DetectID with <value> seconds
    Given the device already has been registered
    And setup global config request service with the lifetime <value>
    And the client changes the OTP lifetime to <value>
    When the user requests an OTP
    Then the OTP is generated with 6 characters
    And the OTP must last <value> seconds before it changes
    Examples:
      | value |
      | 1     |
      | 5     |
```

Retornar a contenido: «lifetimeOTP.feature»

A.2. Configuración entorno de pruebas

A.2.1. Android

Configuración de dependencias en Gradle

build.gradle: dependences

```
dependencies {
    androidTestCompile 'com.android.support:support-annotations:28.0.0'
    androidTestCompile 'com.android.support.test:runner:1.0.2'
    androidTestCompile 'com.android.support.test:rules:1.0.2'
    // Optional -- UI testing with Espresso
    androidTestCompile('com.android.support.test.espresso:espresso-core:3.2.0', {
    exclude group: 'com.android.support',
    module: 'support-annotations'
})
    // Optional -- UI testing with UI Automator
    androidTestCompile 'com.android.support.test.uiautomator:uiautomator-v18:2.1.3'
}
```

Retornar a contenido: «build.gradle»

Configuración pruebas instrumentadas: CucumberTestRunner

CucumberTestRunner.java

```java
package automation.did.test;

import android.os.Bundle;
import android.support.multidex.MultiDex;
import android.support.test.runner.AndroidJUnitRunner;
import automation.did.BuildConfig;
import cucumber.api.android.CucumberInstrumentationCore;

public class CucumberTestRunner extends AndroidJUnitRunner {

    public static final String TAG = CucumberTestRunner.class.getSimpleName();
    private static final String CUCUMBER_TAGS_KEY = "tags";
    private final CucumberInstrumentationCore instrumentationCore = new
    CucumberInstrumentationCore(this);

    @Override
    public void onCreate(final Bundle bundle) {
        MultiDex.install(getTargetContext());
        super.onCreate(bundle);
        final String tags = BuildConfig.TEST_TAGS;
        if (!tags.isEmpty()) {
            bundle.putString(CUCUMBER_TAGS_KEY,
    tags.replaceAll(",","--").replaceAll("\\s", ""));
        }
        instrumentationCore.create(bundle);
    }

    @Override
    public void onStart() {
        waitForIdleSync();
        instrumentationCore.start();
    }

}
```

Retornar a contenido: «CucumberTestRunner.java»

Configuración ejecución pruebas: CucumberTestCase

CucumberTestCase.java

```java
package automation.did.test;

import automation.did.steps.config.ConstantsDID;
import cucumber.api.CucumberOptions;

@CucumberOptions(
    features = {"features"}, // Test scenarios
    glue = {"glue.automation.steps"}, // Steps definitions
    format = {"pretty",
              "html:/mnt/sdcard/did-mobile-automation/cucumber-reports/cucumber-html-report",
              "json:/mnt/sdcard/did-mobile-automation/cucumber-reports/cucumber.json",
              "junit:/mnt/sdcard/did-mobile-automation/cucumber-reports/cucumber.xml"
        }
)

class CucumberTestCase {
}
```

Retornar a contenido: «CucumberTestCase.java»

Configuración sourceSets en Gradle

build.gradle: sourceSets

```
sourceSets {
    main.jniLibs.srcDirs = ['libs']
    test.jniLibs.srcDirs = ['libs']
    androidTest {
        assets.srcDirs = ['src/main/java/assets']
    }
    main { java.srcDirs = ['src/main/java', 'src/main/java/tests'] }
}
```

Retornar a contenido: sourceSets

Configuración defaultConfig en Gradle

build.gradle: defaultConfig

```
    defaultConfig {
        testInstrumentationRunner "automation.did.test.CucumberTestRunner"
        applicationId "automation.did"
    }
```

Retornar a contenido: defaultConfig

Configuración ejecutor de actividades y/o aplicación para las pruebas: Settings

Settings.java

```
package automation.did;

import android.app.Application;
import android.content.Context;
import automation.did.test.ActivityRule;
import org.junit.Rule;

public class Settings {

    public static Context context;
    public static Settings instance;
    public static Application application;

    @Rule
    public static ActivityRule<MainActivity> activityRule = new
    ActivityRule<>(MainActivity.class);

    public static Settings getInstance() {
        activityRule.launchActivity(activityRule.getActivityIntent());
        setContext(activityRule.getActivity());
        application = activityRule.getActivity().getApplication();
        if (instance == null) {
            instance = new Settings();
        }
        return instance;
    }

    public static Context getContext() {
        return context;
    }

    public static Application getApplication() {
        return application;
    }

    private static void setContext(Context contextSet) {
        Settings.context = contextSet;
    }

}
```

Retornar a contenido: Settings

Configuración de las reglas de la actividad: ActivityRule

ActivityRule.java

```java
package automation.did.test;

import android.content.Intent;
import android.support.test.rule.ActivityTestRule;
import automation.did.MainActivity;

public class ActivityRule<A extends MainActivity> extends ActivityTestRule<A> {

    public static Intent intent;

    public ActivityRule(Class<A> activityClass) {
        super(activityClass, false);
    }

    @Override
    public void beforeActivityLaunched() {
        super.beforeActivityLaunched();
    }

    @Override
    public Intent getActivityIntent() {
        intent = new Intent();
        return intent;
    }

    @Override
    protected void afterActivityLaunched() {
        super.afterActivityLaunched();
    }

    @Override
    protected void afterActivityFinished() {
        super.afterActivityFinished();
    }

    public void finishActivity() {
        this.getActivity().finish();
    }
}
```

Retornar a contenido: ActivityRule.java

A.2.2. iOS

Instalación de herramientas y/o dependencias

En primer lugar se debe instalar el administrador de dependencias que permitirá ejecutar Cucumberish en la máquina, denominado **Cocoapods** (https://medium.com/@himalbandara84/quick-guide-to-setup-cucumberish-on-xcode-7db935d67a69):

Instalar Cocoapods

```
sudo gem install cocoapods
```

A continuación se instala **Xcfit** (XCUI, Cucumberish and Fitness Integration Tests) para la automatización y ejecución de pruebas con swift (https://github.com/XCTEQ/XCFit):

Instalar Xcfit

```
gem install xcfit
```

Finalmente se adicionan las plantillas al Xcode:

Agregar plantillas Xcifit

```
xcfit setup_xcode_templates
```

A partir de **Xcode**, se crea una aplicación simple (Single View App) sin ningún tipo de pruebas, dado que se va a crear un nuevo **Target** para las mismas a partir de Cucumberish; por tanto, dentro del nuevo proyecto, se crea un Target asociado con Cucumberish: `File ->New Target ->XCFit ->Cucumberish Bundle` y se cierra el proyecto. A continuación desde el directorio de trabajo del proyecto se ejecuta en consola para la creación del Podfile (o crearlo desde ceros):

Crear Podfile

```
pod init
```

Con el archivo **Podfile** creado, se procede a agregar el Pod de Cucumberish para el target de pruebas creado anteriormente:

Agregar Pod Cucumberish

```
use_frameworks!
target 'automationCucumberishCucumberTests' do
  pod 'Cucumberish'
end
```

Ahora se ejecuta en consola para la instalación de Pods:

Instalar Pods

```
pod install
```

Nota:

- Si se tienen problemas de duplicación de tareas asignar `File ->Workspace Settings ->Build System: Legacy Build System`
- Si se tiene problemas con la importación Cucumberish o no se encuentra el módulo: En el Xcode ir a `Product ->Scheme ->Manage Schemes` y agregar los esquemas relacionados con Cucumberish y los Pods del Target de pruebas.

Retornar a contenido: `Podfile`

Configuración para ejecución de pruebas

automationCucumberishCucumberTests.m

```
#import <Foundation/Foundation.h>
#import "automationCucumberishCucumberTests-Swift.h"
#import <XCTest/XCTest.h>

__attribute__((constructor))
void CucumberishInit()
{
    [automationCucumberishCucumberTests CucumberishSwiftInit];
}
```

Retornar a contenido: `automationCucumberishCucumberTests.m`

Configuración ejecutor de pruebas y/o de la aplicación

automationCucumberishCucumberTests.swift

```
import Foundation
import Cucumberish

class automationCucumberishCucumberTests: NSObject {

    @objc class func CucumberishSwiftInit()
    {
        var application : XCUIApplication!
        beforeStart { () -> Void in
            application = XCUIApplication()
            // StepDefinitions:
            Background().backgroundSteps()
            Registration().registrationCommonsSteps()
            RegisterDevicesByCodeSteps().registrationSteps()
            RegisterLogic().initLogic()
            LifetimeOTPSteps().otpSteps()
            // Launch App:
            application.launch()
        }

        let bundle = Bundle(for: automationCucumberishCucumberTests.self)

        Cucumberish.executeFeatures(inDirectory: "Features", from: bundle,
    includeTags:nil, excludeTags: nil)

        before { (scenario: CCIScenarioDefinition?) in
            RegisterLogic().removeDevice()
            ManageKey().removeAllKeys()
            WiremockClient().deleteRequests()
            WiremockClient().deleteStubs()
            ManageKey().removeAllPassword()
        }
    }
}
```

Retornar a contenido: «automationCucumberishCucumberTests.swift»

A.3. Implementación del Mock

A.3.1. Android

Implementación interfaz WiremockManager

WiremockManager.java

```
package automation.did.stubs.config;

import java.io.UnsupportedEncodingException;
import java.net.URISyntaxException;

public interface WiremockManager {

    void startStub(Object... args) throws Exception;

    boolean waitToConsume() throws URISyntaxException,
   UnsupportedEncodingException;
}
```

Retornar a contenido: «WiremockManager.java»

Implementación stub: globalConfig

GlobalConfigStub.java

```java
package automation.did.stubs.globalConfig;

import com.github.tomakehurst.wiremock.client.WireMock;
import com.github.tomakehurst.wiremock.verification.LoggedRequest;
import com.google.gson.Gson;
import automation.did.steps.config.ConstantsDID;
import automation.did.stubs.config.WiremockManager;
import org.hamcrest.CoreMatchers;
import java.io.UnsupportedEncodingException;
import java.net.URISyntaxException;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.TimeUnit;
import static com.github.tomakehurst.wiremock.client.WireMock.*;
import static org.awaitility.Awaitility.await;

public class GlobalConfigStub implements WiremockManager {
    private final Gson gson = new Gson();
    @Override
    public void startStub(Object... args) {
        String dataResponse = gson.toJson(args[0]);
        WireMock.stubFor(post(urlPathMatching(ConstantsDID.PATH_GLOBAL_CONFIG))
        .withHeader("Accept", equalTo("ENCABEZADO_ACCEPT"))
        .willReturn(aResponse()
        .withBody(dataResponse)
        .withStatus(Integer.parseInt(ConstantsDID.STATUS_CODE_OK))));
    }
    @Override
    public boolean waitToConsume() throws URISyntaxException,
    UnsupportedEncodingException {
        Callable<Boolean> licenseConsumed = () -> {
            List<LoggedRequest> licenseRequests =
    findAll(postRequestedFor(urlEqualTo(ConstantsDID.PATH_GLOBAL_CONFIG)));
            return licenseRequests.size() > 0;
        };
        return await().atMost(30, TimeUnit.SECONDS).until(licenseConsumed,
    CoreMatchers.is(true));
    }

}
```

Retornar a contenido: «GlobalConfigStub.java»

A.3.2. iOS

Implementación protocolo WiremockManagerProtocol

WiremockManagerProtocol.swift

```
import Foundation

protocol WiremockManagerProtocol{

    func startStub()
}
```

Retornar a contenido: «WiremockManagerProtocol.swift»

Implementación WiremockClient

WiremockClient.swift

```
import Foundation

class WiremockClient {
    var baseStub: String
    var wiremockServerIp: String = EnvironmentVar.IP_SERVER
    var wiremockServerPort: String = EnvironmentVar.PORT_SERVER
    init(ip: String, port: String){
        self.wiremockServerIp=ip
        self.wiremockServerPort=port
        self.baseStub = "{\"request\":$REQUEST_STUB,\"response\":$RESPONSE_STUB}"
    }
    init(){
        self.baseStub = "{\"request\":$REQUEST_STUB,\"response\":$RESPONSE_STUB}"
    }
    func createStub(requestStub: RequestStub, responseStub: ResponseStub) {
        let requestData = try! JSONEncoder().encode(requestStub)
        let responseData = try! JSONEncoder().encode(responseStub)
        baseStub = baseStub.replacingOccurrences(of: "$REQUEST_STUB", with: String(data: requestData,
     encoding: String.Encoding.utf8)!)
        baseStub = baseStub.replacingOccurrences(of: "$RESPONSE_STUB", with: String(data:
     responseData, encoding: String.Encoding.utf8)!)
        createMapping(stubDefinition: baseStub)
    }
    private func createMapping(stubDefinition: String) -> Dictionary<String, AnyObject>? {
        var dataResponseJson: Dictionary<String, AnyObject>?
        dataResponseJson = sendWiremockOperation(operation:"POST", stub:stubDefinition)
        return dataResponseJson
    }
    private func sendWiremockOperation(operation: String, stub: String) -> Dictionary<String,
     AnyObject>? {
        NSLog("AUTOMATION - \(stub)")
        var headers = Array<Dictionary<String,String>>()
        headers.append(["application/json": "Content-Type"])
        let dataResponseJson = RestAPI().doRequest(path: ConstantsDID().PATH_MOCK_SERVER, method:
     operation, headers: headers, bodyRequest: stub)
        return dataResponseJson
    }
    func deleteRequests() {
        RestAPI().doRequest(path: ConstantsDID().PATH_MOCK_RESQUESTS, method: "DELETE", headers: nil,
     bodyRequest: nil)
    }
    func deleteStubs() {
        RestAPI().doRequest(path: ConstantsDID().PATH_MOCK_SERVER, method: "DELETE", headers: nil,
     bodyRequest: nil)
    }
    func shutDownMockServer() {
        RestAPI().doRequest(path: ConstantsDID().PATH_MOCK_SHUT_DOWN, method: "POST", headers: nil,
     bodyRequest: nil)
    }
}
```

Retornar a contenido: «WiremockClient.swift»

Implementación RequestStub

RequestStub.swift

```
import Foundation

class RequestStub : Codable{

    var method: String?
    var url: String?
    var urlPath: String?
    var urlPathPattern: String?
    var urlPattern: String?
    var queryParameters: Dictionary<String,Dictionary<String,String>?>?
    var headers: Dictionary<String,Dictionary<String,String>?>?
    var bodyPatterns: Dictionary<String,String>?

    init(){
        method = nil
        url = nil
        urlPath = nil
        urlPathPattern = nil
        urlPattern = nil
        queryParameters = nil
        headers = nil
        bodyPatterns = nil
    }

}
```

Retornar a contenido: «RequestStub.swift»

Implementación ResponseStub

ResponseStub.swift

```
import Foundation

class ResponseStub: Codable{

    var status: Int? = nil
    var statusMessage: String? = nil
    var headers: Dictionary<String,String>?
    var body: String? = nil

    init(){
        status = nil
        statusMessage = nil
        headers = nil
        body = nil
    }

}
```

Retornar a contenido: «ResponseStub.swift»

Implementación stub: globalConfig

GlobalConfigStub.swift

```
import Foundation

class GlobalConfigStub: WiremockManagerProtocol{
    var wiremocklient: WiremockClient
    static var dataBody: String?
    init(){
        self.wiremocklient = WiremockClient()
    }
    func setBodyStub() {
        GlobalConfigStub.dataBody = CUERPO_RESPUESTA;
    }
    func startStub() {
        let requestStub = RequestStub()
        let responseStub = ResponseStub()
        requestStub.urlPathPattern = ConstantsDID().PATH_GLOBAL_CONFIG
        requestStub.method = "POST"
        requestStub.headers = ["Accept":["equalTo":"ENCABEZADO_ACCEPT"]]
        responseStub.body = GlobalConfigStub.dataBody!
        responseStub.status = 200
        responseStub.headers = ["Content-Type":"application/json"]
        wiremocklient.createStub(requestStub: requestStub, responseStub: responseStub)
    }
    func countRequests() -> Bool {
        let bodyRequest  = "{\"method\": \"POST\", \"url\": \"\(ConstantsDID().PATH_GLOBAL_CONFIG)\"}"
        var headers = Array<Dictionary<String,String>>()
        headers.append(["application/json": "Content-Type"])
        let dataResponseJson = RestAPI().doRequest(path: ConstantsDID().PATH_MOCK_COUNT, method:
     "POST", headers: headers, bodyRequest: bodyRequest)
        let requestCount = dataResponseJson != nil ? dataResponseJson![ConstantsDID().COUNT_MOCK] as!
     Int : 0
        NSLog("AUTOMATION - Requests for Global Config Stub \(requestCount)")

        return requestCount != 0 ? true : false
    }
    func waitToConsume() {
        NSLog("AUTOMATION - Wait Update Global Config")
        var waitRequests = self.countRequests()
        let time = NSDate().timeIntervalSince1970
        while (waitRequests == false) {
            waitRequests = self.countRequests()
            if (RestAPI().waitingTime(time: time)){
                waitRequests = true
                NSLog("AUTOMATION - Time Out Waiting Update Global Config")
            }
        }
    }

}
```

Retornar a contenido: «GlobalConfigStub.swift»

A.4. Integración con el SDK-DID

A.4.1. Android

Configuración del Gradle principal

build.gradle: repositories & dependencies

```
buildscript {

    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.5.3'
        classpath 'com.google.gms:google-services:4.3.3'
    }
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}
```

Retornar a contenido: «build.gradle»

Configuración del Gradle de la aplicación

build.gradle: android

```
android {
    compileSdkVersion 29
    buildToolsVersion "29.0.2"
    defaultConfig {
        minSdkVersion 21
        targetSdkVersion 29
        multiDexEnabled true
    }
}
```

Retornar a contenido: «build.gradle»

build.gradle: dependencies

```
dependencies {
    def latestSDKVersion = "8.2.0"
    implementation "net.easysol:liveness:$latestSDKVersion@aar"
    implementation "com.cyxtera.did.sdk:data:$latestSDKVersion@aar"
    implementation "com.cyxtera.did.sdk:encryptor:$latestSDKVersion@aar"
    implementation "com.cyxtera.did.sdk:tokens:$latestSDKVersion@aar"
    implementation "com.cyxtera.did.sdk:faceid_detector:$latestSDKVersion@aar"
    implementation
    "com.cyxtera.did.sdk:faceid-authenticator-sdk:$latestSDKVersion@aar"
    implementation "com.cyxtera.did.sdk:sdk:$latestSDKVersion@aar"
    implementation "com.cyxtera.did.sdk:offline:$latestSDKVersion@aar"
    implementation 'com.android.support:multidex:1.0.3'
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'com.google.android.material:material:1.1.0'
    implementation 'androidx.legacy:legacy-support-v4:1.0.0'
    implementation 'androidx.mediarouter:mediarouter:1.1.0'
    implementation 'androidx.vectordrawable:vectordrawable-animated:1.1.0'
    implementation 'androidx.cardview:cardview:1.0.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'com.google.firebase:firebase-core:17.2.3'
    implementation 'com.google.code.gson:gson:2.8.6'
    implementation 'com.squareup.retrofit2:retrofit:2.6.2'
    implementation 'com.squareup.retrofit2:converter-gson:2.6.2'
    implementation 'com.squareup.okhttp3:logging-interceptor:4.2.2'
    annotationProcessor 'com.google.dagger:dagger-compiler:2.25.2'
    annotationProcessor 'com.google.dagger:dagger-android-processor:2.25.2'
    implementation('com.google.dagger:dagger-android-support:2.25.2', {
        exclude group: 'com.android.support'
    })
}
```

Retornar a contenido: «build.gradle»

Configuración Manifiesto

AndroidManifiest.xml: application

```
<service android:name=
"did.services.UpdateDeviceInformationService" />
```

Retornar a contenido: «AndroidManifiest.xml»

AndroidManifiest.xml: permission

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.GET_ACCOUNTS" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

<uses-feature
    android:name="android.hardware.telephony"
    android:required="false" />
<uses-feature
    android:name="android.hardware.wifi"
    android:required="false" />
```

Retornar a contenido: «AndroidManifiest.xml»

A.5. Implementación de los pasos a ejecutar en cada escenario

A.5.1. Android

StepDefinitions.java

```
package glue.automation.steps;

import org.junit.Assert; // Librera para ejecutar los assert
import cucumber.api.java.*;
import cucumber.api.java.en.*;

public class StepDefinitions {

    @Before
    public void before() {
        // ACCIONES A EJECUTAR ANTES DE UN ESCENARIO
    }

    @Given("^paso Given$")
    public void prepareStepGiven() {
        // IMPLEMENTACIN DEL PASO GIVEN DEL ESCENARIO
    }

    @When("^paso When$")
    public void executeStepWhen() {
        // IMPLEMENTACIN DEL PASO WHEN DEL ESCENARIO
    }

    @Then("^paso Then$")
    public void validateStepThen(){
        // IMPLEMENTACIN DEL PASO THEN DEL ESCENARIO
    }

    @After
    public void after() throws InterruptedException {
        // ACCIONES A EJECUTAR AL FINALIZAR UN ESCENARIO
    }

}
```

Retornar a contenido: «StepDefinitions.java»

A.5.2. iOS

StepDefinitions.swift

```
import Foundation
import Cucumberish
import XCTest //Asserts: XCTAssert

class StepDefinitions {

    func stepDefinition(){

        Given("paso Given") { (args, userInfo) -> Void in
        // IMPLEMENTACIN DEL PASO GIVEN DEL ESCENARIO
        }

        When("paso When") { (args, userInfo) -> Void in
        // IMPLEMENTACIN DEL PASO WHEN DEL ESCENARIO
        }

        Then("paso Then") { (args, userInfo) -> Void in
        // IMPLEMENTACIN DEL PASO THEN DEL ESCENARIO
        }

    }

}
```

Retornar a contenido: «StepDefinitions.swift»

A.6. Ejecución de las pruebas

A.6.1. Android

Para la ejecución de los escenarios de prueba se requiere tener instalado **Android Studio**, y por tanto **adb**, **sdkmanager**, **avdmanager** y **emulator**.

→ **Listar dispositivos**

Para conocer los dispositivos de **Google** candidatos de instalación, ejecute el comando:

Dispositivos Google a instalar:

```
<PATH_ANDROID>/avdmanager list device
```

Donde <PATH_ANDROID> es el path donde se encuentran el **sdkmanager** y el **avdmanager**. Así mismo para conocer los dispositivos de **Google** instalados en la máquina, ejecute el comando:

Dispositivos Google instalados:

```
PATH_EMULATOR>/emulator -list-avds
```

Donde: <PATH_EMULATOR> es el path donde se encuentra el **emulator**. Para conocer los dispositivos de **Genymotion** instalados en la máquina, ejecute el comando:

Dispositivos Genymotion instalados:

```
VBoxManage list vms
```

→ **Instalación de dispositivos**

La instalación de cada dispositivo de **Google**, se lleva a cabo mediante la ejecución de los comandos:

Instalación dispositivos Google

```
<PATH_ANDROID>/sdkmanager --install
    "system-images;android-<IMAGE>;google_apis;x86"

echo "no" | <PATH_ANDROID>/avdmanager --verbose create avd --force --name
    "<NAME_DEVICE>" --package "system-images;android-<IMAGE>;google_apis;x86"
    --tag "google_apis" --abi "x86"
```

Donde <IMAGE> es el número de la imagen y <NAME_DEVICE> es cualquier nombre para identificar el dispositivo. Para más información acerca de la instalación de dispositivos emuladores de Google:
https://gist.github.com/mrk-han/66ac1a724456cadf1c93f4218c6060ae

→ **Ejecución de dispositivos**

La ejecución de cada dispositivo de Google se lleva a cabo mediante los comandos:

Ejecución dispositivos Google en modo no-headless

```
<PATH_EMULATOR>/emulator -avd <NAME_DEVICE> -port <PORT_DEVICE> &
```

Con el comando `emulator` los dispositivos corren con la interfaz gráfica, mientras que con el comando `emulator-headless`, los dispositivos corren sin la Interfaz Gráfica:

Ejecución dispositivos Google en modo headless

```
<PATH_EMULATOR>/emulator-headless -avd <NAME_DEVICE> -port <PORT_DEVICE> &
```

`<PORT_DEVICE>` es el puerto donde el dispositivo será ejecutado. Es recomendable que el puerto sea un número par definido en el rango `5554-5584` para un mejor seguimiento del dispositivo. Para más información acerca de la ejecución de dispositivos de Google: https://developer.android.com/studio/run/emulator-commandline?hl=es.

La ejecución de cada dispositivo de **Genymotion** se lleva a cabo mediante el comando:

Ejecución dispositivos Genymotion en modo no-headless

```
~/.genymotion/player --vm-name "<NAME_DEVICE>"
~/.genymotion/player --vm-name <VM-UUID>
```

Donde `<NAME_DEVICE>` es el nombre del dispositivo, o también se puede ejecutar mediante el ID del dispositivo `<VM-UUID>`.

Finalmente para conocer los dispositivos que se están ejecutando en la máquina:

Dispositivos ejecutados

```
adb devices
```

O para obtener información específica de cada dispositivo que se está ejecutando en la máquina:

Información de los dispositivos ejecutados

```
adb -s <EMULATOR> shell getprop
```

Donde `<EMULATOR>` es el identificador del dispositivo asignado en la lista anterior por medio de **abd**.

→ **Ejecución de las pruebas**

Para correr las pruebas de aceptación en todos los dispositivos ejecutados:

Ejecución de las pruebas en Android

```
./gradlew connectedCheck
```

Retornar a contenido: ejecución de las pruebas de aceptación

A.6.2. iOS

→ **Listado de dispositivos**

Para obtener información acerca de los dispositivos emuladores como físicos (conectados a la máquina), se ejecuta el siguiente comando:

Listado de dispositivos

```
instruments -s devices
```

→ **Ejecución de las pruebas**

Las pruebas se pueden ejecutar de diferentes maneras, desde Xcode: Product - Test, o:

Ejecución de las pruebas en iOS desde Xcode

```
command + U
```

Desde consola en múltiples dispositivos emulados ('destination' por dispositivo): (https://www.mokacoding.com/blog/running-tests-from-the-terminal/).

Ejecución de las pruebas en iOS desde consola para emuladores

```
xcodebuild \
-workspace PROJECT_NAME.xcworkspace \
-scheme PROJECT_NAME \
-destination 'platform=iOS Simulator,name=iPhone 11 Pro Max,OS=13.3' \
-destination 'platform=iOS Simulator,name=OTHER DEVICE,OS=13.3' \
test
```

Desde consola con dispositivo físico:

Ejecución de las pruebas en iOS desde consola para dispositivos físicos

```
xcodebuild \
-workspace automationCucumberish.xcworkspace \
-scheme automationCucumberish \
-destination 'name=NAME_DEVICE' \
test
```

Retornar a contenido: ejecución de las pruebas de aceptación

A.7. Generación de reportes en iOS

En primer lugar se debe instalar **xcpretty** para generar reportes y **xcpretty-json-formatter** para generar un reporte en formato json (https://github.com/xcpretty/xcpretty,https://github.com/marcelofabri/xcpretty-json-formatter):

Instalación xcpretty

```
sudo gem install xcpretty
sudo gem install xcpretty-json-formatter
```

Para generar los reportes de la ejecución de las pruebas ejecutar por dispositivo:

Ejecución pruebas y generación de reportes

```
xcodebuild \
-workspace PROJECT_NAME.xcworkspace \
-scheme PROJECT_NAME \
-destination 'platform=iOS Simulator,name=iPhone 11 Pro Max,OS=13.3' \
test| xcpretty -f `xcpretty-json-formatter`  \
-r html -r junit
```

De esta manera en el directorio raíz se crea la carpeta **build/reports** con los archivos:

- junit.xml
- tests.html
- erros.json

Si se desea seleccionar un PATH y nombres de archivos diferentes por dispositivo:

Ejecución pruebas y generación de reportes en ruta específica

```
xcodebuild \
-workspace PROJECT_NAME.xcworkspace \
-scheme PROJECT_NAME \
-destination 'platform=iOS Simulator,name=iPhone 11 Pro Max,OS=13.3' \
test|XCPRETTY_JSON_FILE_OUTPUT=PATH/NAME.json xcpretty -f
    `xcpretty-json-formatter`  \
-r html -o PATH/NAME.html  \
-r junit -o PATH/NAME.xml
```

Retornar a contenido: generación de reportes en iOS

A.8. Implementación sistema de pruebas automatizado en un entorno de integración continua

A.8.1. Android

Jenkinsfile Android

```
pipeline {
    agent { label 'Agent' }
    stages {
        stage('Clear gradle cache') {
                steps {
                    sh 'rm -rf ${MAIN_DIR}/.gradle/caches/modules-2/files-2.1/com.cyxtera.did.sdk/'
                }
            }
        stage('Build') {
          steps {
            sh 'chmod 755 gradlew'
            echo 'Building project'
            sh './gradlew clean'
            echo 'Run Devices'
             sh '''
             ${MAIN_DIR}/Library/Android/sdk/emulator/emulator-headless -avd pixel_8.0 -port 5554 &
             ${MAIN_DIR}/Library/Android/sdk/emulator/emulator-headless -avd pixel_6.0 -port 5556 &
             sleep 10
             /usr/local/bin/adb devices'''
          }
        }
        stage('Testing') {
            steps {
             echo 'Proceeding to grant permissions'
             sh './gradlew grantPermissions'
             echo 'Proceeding to execute instrumentation tests with cucumber'
             sh './gradlew connectedCheck'
            }
        }
    }
    post {
        always {
             echo 'Export Cucumber Reports'
             sh './gradlew exportCucumberReports'
             cucumber buildStatus: 'UNSTABLE',
                                     fileIncludePattern: '*cucumber.json',
                                     jsonReportDirectory: 'app/build/reports/cucumber/'
        }
    }
}
```

Retornar a contenido: pipeline de Android

A.8.2. iOS

Jenkinsfile iOS

```
pipeline {
agent { label 'Agent'}
environment {  BRANCH_LIB = "development" }
 stages {
  stage('Copy Artifacts') {
   steps {
    script {
        echo 'Copy ANY Library'
        step ([$class: 'CopyArtifact',
        projectName: "ANY/${BRANCH_LIB}",
        filter: "Framework/*.zip",
        target: 'Libraries/ANY',
        selector: lastSuccessful()]);
        sh 'unzip Libraries/ANY/Framework/ANY.zip -d Libraries'
        }
     }
  }
  stage('Start Mockup Server') {
   steps {
    dir('wiremock'){
      sh 'java -jar wiremock-standalone-2.25.1.jar -port 9090 --root-dir stubs &'}
        }
  }
  stage('Enable POD to Cucumberish ') {
   steps {
        echo 'Installing POD'
        sh '/usr/local/bin/pod install'
        }
  }
  stage('Executing escenarios'){
   steps {
     sh'''export LANG=en_US.UTF-8
     export LANGUAGE=en_US.UTF-8
     export LC_ALL=en_US.UTF-8
     instruments -s devices | grep "^iPhone.*Simulator)$"  | cut -f 1 -d "(" >> simulatorList.txt
     instruments -s devices | grep "^QA" | cut -f 1 -d "(" >> connectedDevicesList.txt
     rm -rf automationCucumberish/DID-SDK-Libraries/*
     cp -r Libraries/Devices/*.framework automationCucumberish/DID-SDK-Libraries/
     while read device; do
      file=`echo $device | tr -d ' '`
       set -o pipefail && xcodebuild -workspace automationCucumberish.xcworkspace -scheme
     automationCucumberish -destination "name=${device}" test | /usr/local/bin/xcpretty -r junit -o
     build/reports/${file}.xml
      done < connectedDevicesList.txt'''
    }
  }
}
    post {
        always {
            junit 'build/reports/*.xml'
            sh '''rm simulatorList.txt connectedDevicesList.txt Libraries'''
        }
    }
}
```

Retornar a contenido: pipeline de iOS

B. Anexo: Implementación CI

B.1. Pipeline Administrador

Jenkinsfile: Pipeline Administrador

```
pipeline {
 agent { label 'Agent'}
  triggers {
    GenericTrigger( genericVariables: [
         [key: 'object_kind', value: '$.object_kind'],
         [key: 'user_name', value: '$.user_name'],
         [key: 'user_email', value: '$.user_email'],
         [key: 'repository_name', value: '$.repository.name'],
         [key: 'total_commits_count', value: '$.total_commits_count'],
         [key: 'commit_messages', value: '$.commits[*].message'],
         [key: 'ref', value: '$.ref']],
            printContributedVariables: true, printPostContent: true, silentResponse: false)
  }
  stages {stage('Verify Job') { steps { getJob() } } }
    post { failure { notifyEmail()
                     notifyTeams() } }
}
def getJob(){
    switch (repository_name) {
        case "PROYECTO1": executeJob('Job_Proyecto_1')
        break
      default: echo 'Any Job to Execute'
        break}
}
def executeJob(def job){ def job_build = job + ref.substring(10)
    def params=[]
    params.add(string(name:'object_kind', value:object_kind))
    params.add(string(name:'user_name', value:user_name))
    params.add(string(name:'user_email', value:user_email))
    params.add(string(name:'total_commits_count', value:total_commits_count))
    params.add(string(name:'commit_messages', value:commit_messages))
    script {build job : job_build, parameters:params, quietPeriod: 2, wait: false}
}
def notifyEmail() { emailext ( subject: '${BUILD_STATUS}: ${JOB_NAME} Build #${BUILD_NUMBER}',
                    body: 'MENSAJE', to: user_email)
}
def notifyTeams(){ office365ConnectorSend webhookUrl: 'URL_TEAMS',
  factDefinitions: [[name: "Request Type", template: '${object_kind}'],
                    [name: "By the user", template: '${user_name}'],
                    [name: "With the commits:", template: '${commit_messages}']]
}
```

Retornar a contenido: pipeline Administrador

B.2. Implementación JaCoCo

build.gradle: Integración Plugin JaCoCo

```
plugins {
    id 'org.barfuin.gradle.jacocolog' version '1.0.1'
    id 'org.jacoco:org.jacoco.core' version '0.8.5'
}
apply plugin: 'jacoco'
jacoco {
    toolVersion = "0.8.5"
     reportsDir = file("$buildDir/reports")
}

tasks.withType(Test) {
    jacoco.includeNoLocationClasses = true
    jacoco.excludes = ['jdk.internal.*']
}

task jacocoTestReport(type: JacocoReport, dependsOn: ['testDevDebugUnitTest',
    'createDevDebugCoverageReport'], group: 'GROUP') {
    reports {
        xml.enabled = true
        html.enabled = true
    }

    def fileFilter = ['**/R.class',
                      '**/R$*.class',
                      '**/BuildConfig.*',
                      '**/Manifest*.*',
                      '**/*Test*.*',
                      'android/**/*.*']
    def debugTree = fileTree(dir:
    "$project.buildDir/intermediates/javac/devDebug/classes", excludes: fileFilter)
    def mainSrc = "$project.projectDir/src/main/java"
    sourceDirectories = files([mainSrc])
    classDirectories = files([debugTree])
    executionData = fileTree(dir: project.buildDir, includes: [
            'jacoco/testDevDebugUnitTest.exec',
            'outputs/code_coverage/devDebugAndroidTest/connected/*coverage.ec'
    ])
}
```

Retornar a contenido: «build.gradle»

B.3. Implementación Sonarqube

B.3.1. Android

En primer lugar, para lograr la integración con SonarQube con un proyecto Android, se debe agregar el plugin de SonarQube, dentro del `build.gradle` de la aplicación:

build.gradle: Integración Plugin SonarQube

```
plugins {
    id "org.sonarqube" version "2.8"
}
apply from: 'sonarqube.gradle'
```

Como se observa, se debe agregar un archivo dentro del directorio de la aplicación: `sonarqube.gradle`, donde se encuentra la configuración de SonarQube, como se muestra a continuación:

sonarqube.gradle

```
sonarqube {
    properties {
        property "sonar.host.url", "http://URL_SONARQUBE:PORT_SONARQUBE"
        property "sonar.projectKey", "IDENTIFICADOR_PROYECTO"
        property "sonar.projectName", "NOMBRE_PROYECTO"
        property "sonar.coverage.jacoco.xmlReportPaths",
    "build/reports/jacocoTestReport/jacocoTestReport.xml"
        }
}
```

En la configuración de SonarQube, se indica el host donde se encuentra el servidor de Sonar, el nombre e identificador del proyecto en específico dentro del servidor de Sonar y finalmente, se indica el directorio donde se encuentran los reportes `XML` generados por JaCoCo.

Retornar a contenido: integración con Sonarqube

B.3.2. iOS

En primer lugar se debe instalar **sonar-scanner** en la máquina de ejecución, dado que esta herramienta de SonarQube, es la que ejecuta el análisis de código estático.

Sin embargo, para poder ejecutar el análisis, **sonar-scanner** requiere del informe de cobertura generado por Xcode, en un archivo con extensión **.xcresult**; dado que **sonar-scanner**, maneja archivos con extensión **.xml**, se debe ejecutar el script `xccov-to-sonarqube-generic.sh` que convierte archivos .xcresult a .xml, dicho script se almacena dentro del directorio de instalación del **sonar.scanner**:

```
bash $SONAR_SCANNER/xccov-to-sonarqube-generic.sh PATH_XCRESULT/*.xcresult>coverage.xml
```

Donde, `SONAR_SCANNER`, es el directorio donde se encuentra instalado **sonar-scanner**. Para ejecutar el análisis con **sonar-scanner**, se pueden incluir las propiedades o configuración de SonarQube dentro de la misma ejecución, o se puede crear un archivo con toda la configuración: `sonarqube.properties`, como se muestra a continuación, para proyecto Swift como ObjectiveC:

sonarqube.properties

```
sonar.host.url=http://URL_SONARQUBE:PORT_SONARQUBE
sonar.projectKey=IDENTIFICADOR_PROYECTO
sonar.projectName=NOMBRE_PROYECTO
sonar.sources=PATH_SOURCES
sonar.tests=PATH_TESTS
sonar.exclusions=ARCHIVOS_A_EXCLUIR
sonar.coverageReportPaths=coverage.xml
```

Si se trata de un proyecto ObjectiveC se debe adicionar:

sonarqube.properties - ObjectiveC

```
sonar.cfamily.build-wrapper-output=objCwrapper
sonar.cfamily.cache.enabled=false
sonar.cfamily.threads=1
```

De esta manera, la ejecución de **sonar-scanner**, se realiza de la siguiente manera:

```
$SONAR_SCANNER/bin/sonar-scanner -Dproject.settings=sonar.properties
```

En el archivo de configuración de SonarQube, se indica el host donde se encuentra el servidor de Sonar, el nombre e identificador y nombre del proyecto en específico dentro del servidor de Sonar, la ruta donde se encuentran los archivos fuente a analizar (sources), los archivos de prueba (tests), aquellos archivos que se desean excluir del análisis y finalmente, se indica el directorio donde se encuentran los reportes `XML` generados a partir de la cobertura dada por Xcode.

Retornar a contenido: integración con Sonarqube

Referencias bibliográficas

[1] Niknil Pathania, “Learning Continuous Integration with Jenkins”. Packt, Birmingham - Mumbai, Second , 2017

[2] Dorothy Graham, Erik van Veenendaal, Isabel Evans, Rex Black, “Foundations of Software Testing ISTQB Certification”.

[3] International Software Testing Qualifications Board (ISTQB), “Probador Certificado. programa de estudio de nivel básico”. Versión 2010

[4] Dr. Hermann Steffen, “El equipo de Software Testing: organización, metodología, técnicas y herramientas”. Fundación para el Desarrollo de Nuevas Tecnologías. Seminario FUNTEC. Buenos Aires, Abril, 2010.

[5] Marianne Hollier, Allan Wagner, “Continuous Testing for dummies”. Compliments of IBM, IBM Limited Edition. 2017.

[6] Paul Duvall with Steve Matyas, Andrew Glover, “Continuous Integration, Improving Software Quality and Reducing Risk”. Pearson Education, Inc. 2007.

[7] Martin Fowler, “Continuous Integration”. May, 2006.

[8] AppGate “DetectID®”. [online] Tomado de: https://www.easysol.net/eng/tfp/strong-authentication. Abril, 2020.

[9] Software Testing Magazine, “Millisecond Full Stack Acceptance Tests”, October 28, 2019. [online] Tomado de: https://www.softwaretestingmagazine.com/videos/millisecond-full-stack-acceptance-tests/. Abril 2020.

[10] Mario Linares Vásquez y Camilo Escobar Velásquez, “Pruebas Automáticas de Software”. Universidad de los Andes, Bogotá, Colombia, 2018. [online] Tomado de: https://miso-4208-labs.gitlab.io/book/chapter1/pruebas-automaticas-y-automatizadas.html. Abril 2020.

[11] Universidad de lo Andes, “El futuro de las Pruebas Automáticas de Software en Colombia”, Mayo de 2019. [online] Tomado de: https://sistemas.uniandes.edu.co/foro/miso/2019/. Abril 2020.