Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Framework de Automatización de Tests Unificado para Web y Aplicaciones Móviles

Autor: Alejandra Alvarado Tirado

Tutor(a): Guillermo Román Diez

Madrid, enero 2025

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: Framework de Automatización de Pruebas para Aplicaciones Web y Móviles
Enero 2025

Autor: Alejandra Alvarado Tirado

Tutor:
Guillermo Román Diez
Lenguajes y Sistemas Informáticos e Ingeniería de Software
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

En este trabajo abordamos el desarrollo de un framework de automatización de pruebas para aplicaciones web y móviles, diseñado con el propósito de simplificar la elaboración de pruebas para páginas web y aplicaciones Android o nativas. El proyecto logra esto proporcionando un solo framework que logre utilizar el mismo código para ambos tipos de pruebas, permitiéndole a los desarrolladores la realización de dichas pruebas en diferentes entornos.

Para tener un concepto más claro sobre la automatización de pruebas, en el Trabajo de Fin de Grado se abordan conceptos básicos sobre la calidad de software y las características que la componen, para entender de donde nace la necesidad de comprobar la calidad en los productos software. También se profundiza en definiciones especificas al testing, para su posterior entendimiento a lo largo de la lectura del trabajo; se discuten los tipos de pruebas de automatización de test más relevantes en el desarrollo de proyecto y se subraya la importancia de la automatización en ese contexto, destacando herramientas como Selenium y Appium.

Se tocan también temas como la metodología del Desarrollo Guiado por Comportamiento (BDD), importante en el desarrollo del framework, ya que es la metodología utilizada y permite el uso de lenguajes comunes en los casos de pruebas.

Una vez descritos todos los componentes que ayudan al entendimiento del framework y sus objetivos, pasamos a la descripción del proyecto y los pasos tomados para su realización.

La creación del framework responde a la necesidad de la industria de garantizar la calidad de los productos software, donde la automatización resulta esencial para mantener la fiabilidad de las pruebas. El framework realizado se caracteriza por la configuración dinámica de variables de entorno, permitiendo a los desarrolladores ajustar el contexto de las pruebas sin modificar el código, y adaptarse así a aplicaciones web y móviles.

Estas configuraciones definen los principales parámetros del entorno de los tests, como la plataforma en que se realiza la prueba, la selección de navegadores, resolución de pantalla y la configuración de idiomas. La configuración flexible de las variables de entorno asegura su fácil integración en equipos de desarrollo que se encuentren trabajando con productos tanto web como móviles. La arquitectura del framework incluye componentes para la gestión de drivers y capacidades de navegación, que permiten la ejecución de pruebas automatizadas en distintos navegadores y sistemas móviles mediante la integración de Selenium para web y Appium para dispositivos móviles.

Un aspecto destacado de este proyecto es la integración de Allure Reports, que ofrece una interfaz visual y personalizable para el análisis de resultados. Allure facilita el monitoreo de las pruebas, permitiendo visualizar de manera detallada el estado de cada prueba, los posibles fallos y los datos relevantes del entorno en que fueron ejecutadas. Este sistema de reportes agrega valor en términos de trazabilidad y análisis de errores.

Adicional a esto, para la implementación de la metodología del Desarrollo Guiado por Comportamiento (BDD), el framework incorpora el uso de

Cucumber y Gherkin. Estos permiten que las pruebas sean escritas en lenguaje natural y sean comprensibles para todos los miembros del equipo.

Finalmente, el proyecto incorpora la posibilidad de ejecutar pruebas en la nube a través de SauceLabs. Esta plataforma permite a los usuarios ejecutar pruebas de manera remota en una extensa variedad de dispositivos y navegadores. Al ofrecer esta posibilidad aseguramos que los productos software pasen los criterios requeridos de calidad en diferentes plataformas y dispositivos.

En conjunto, este framework de automatización de tests representa una herramienta flexible y de fácil adaptabilidad, que contribuye de manera significativa a la mejora del desempeño de equipos multidisciplinarios.

Abstract

In this work, we address the development of a test automation framework for web and mobile applications, designed to simplify the creation of tests for websites and Android or native applications. The project achieves this by providing a single framework capable of using the same code for both types of tests, enabling developers to perform these tests across different environments.

To gain a clearer understanding of test automation, the Final Degree Project delves into basic concepts of software quality and its defining characteristics, to understand the need to verify quality in software products. It also explores specific definitions related to testing to enhance comprehension throughout the reading of the work; the most relevant types of test automation for project development are discussed, and the importance of automation in this context is highlighted, with tools like Selenium and Appium taking center stage.

Topics such as Behavior-Driven Development (BDD) methodology are also addressed. This methodology is significant in the framework's development, as it provides the foundation for using common languages in test cases.

Once all components that facilitate understanding of the framework and its objectives are described, we move on to the project's description and the steps taken for its implementation.

The creation of the framework responds to the industry's need to ensure the quality of software products, where automation is essential to maintain the reliability of tests. The framework developed is characterized by the dynamic configuration of environment variables, allowing developers to adjust the testing context without modifying the code, and thereby adapt to both web and mobile applications.

These configurations define the main parameters of the testing environment, such as the platform on which the test is performed, browser selection, screen resolution, and language settings. The flexible configuration of environment variables ensures easy integration into development teams working on both web and mobile products. The framework's architecture includes components for driver management and navigation capabilities, enabling automated testing across various browsers and mobile systems through the integration of Selenium for web and Appium for mobile devices.

A key feature of this project is the integration of Allure Reports, which offers a visual and customizable interface for results analysis. Allure facilitates test monitoring, allowing detailed visualization of each test's status, possible failures, and relevant data about the environment in which they were executed. This reporting system adds value in terms of traceability and error analysis.

Additionally, to implement the Behavior-Driven Development (BDD) methodology, the framework incorporates the use of Cucumber and Gherkin. These tools allow tests to be written in natural language, making them comprehensible to all team members.

Finally, the project includes the ability to execute tests in the cloud through SauceLabs. This platform enables users to run tests remotely on a wide range

of devices and browsers. By offering this capability, we ensure that software products meet the required quality criteria across different platforms and devices.

In summary, this test automation framework represents a flexible and easily adaptable tool that significantly contributes to improving the performance of multidisciplinary teams.

Tabla de contenidos

Lista de figuras

1. Introducción

Desde el uso masivo de productos software para la vida cotidiana, la calidad de estos se ha ido convirtiendo en una parte cada vez más importante para las empresas y la sociedad. Si bien el uso de este tipo de sistemas ya ha supuesto una total revolución, la extensión de su uso ha seguido creciendo a pasos agigantados a lo largo de la historia.

Esta constante evolución de la tecnología, que no da señales de detenerse, ha abierto las puertas al área del testing, que asegura que dichos productos se entreguen asegurando su funcionamiento.

1.1. Motivación

Con el auge del uso de sistemas software para todo tipo de actividades, desde profesionales hasta cotidianas, la necesidad de comprobar el funcionamiento de software complejo cada vez es mayor [86]. Con el fin de definir y contabilizar su calidad se han desarrollado numerosos estándares a lo largo de los años, pero igualmente sigue siento de gran dificultad valorar su estado en cuanto a la calidad, ya que esta no es algo tangible.

Hoy en día, existe una gran cantidad de tecnologías: múltiples dispositivos, sistemas operativos, etc, por lo que la comprobación de los productos software bajo dichas configuraciones se hacen complicado cuando se hace de forma manual. Por esta razón muchas compañías optan por la utilización de frameworks para la automatización de sus pruebas. Los frameworks además de presentar grandes beneficios en cuanto al tiempo de ejecución, también pueden ser configurados para realizar pruebas bajo diferentes entornos, lo que los hace perfectos para la situación actual.

Sin embargo, a pesar de todas sus ventajas, los frameworks también presentan un elevado costo de desarrollo, no solo económico, sino también en cuestión de recursos y tiempo. Además, estos suelen ser diseñados solo para determinados proyectos y en plataformas específicas, lo que limita proyectos que tengan que comprobar el funcionamiento de un sistemas que se desarrolló en diferentes plataformas, como productos que pueden ser visualizados en web y además en aplicaciones móviles.

1.2. Objetivos

El objetivo de principal de este trabajo es diseñar e implementar un framework de automatización de pruebas web y móviles, que permita realizar pruebas que comprueben el funcionamiento de sistemas que se presenten en dichas plataformas, reutilizando el mismo código y ahorrando la necesidad de implementar dos frameworks.

Este desarrollo conlleva la resolución de varios objetivos. El primero sería la investigación de las herramientas a utilizar y comprobar su funcionamiento en un framework conjunto.

El segundo, sería implementar un framework de pruebas automatizadas, que sea capaz de comprobar diferentes funcionamientos bajo múltiples configuraciones de entornos. Para esto se deberá realizar un entorno modificable y amigable para el usuario, que permita su modificación sin tener que reescribir código.

Y, el tercero y último, sería la creación de pruebas que verifiquen las funcionalidades del framework, para esto se tendrá que probar una aplicación que tenga tanto aplicación móvil como web.

1.3. Planificación

Para lograr dichos objetivos, el desarrollo del trabajo se repartió en 6 tareas:

- Investigación y análisis: investigación de las herramientas y plataformas a utilizar en el proyecto.
- Configuración del entorno de desarrollo: instalación de las herramientas necesarias para el desarrollo del framework.
- Desarrollo del framework base: implementación de los métodos encargados de modificar las variables de entorno, por ende los métodos encargados de gestionar el lanzamiento de los test en diferentes configuraciones.
- Implementación de pruebas para aplicaciones web: desarrollo de scripts automatizados para aplicaciones web.
- Implementación de pruebas para aplicaciones móviles: creación de scripts automatizados para aplicaciones móviles.
- Reporte de resultados: configuración de herramienta de reportes y generación de informes que reflejen el resultado de los tests.

Para la planificación de dichas tareas se ha previsto un marco de tiempo de 12 semanas. El reparto de estas se puede observar en la figura 1.1..

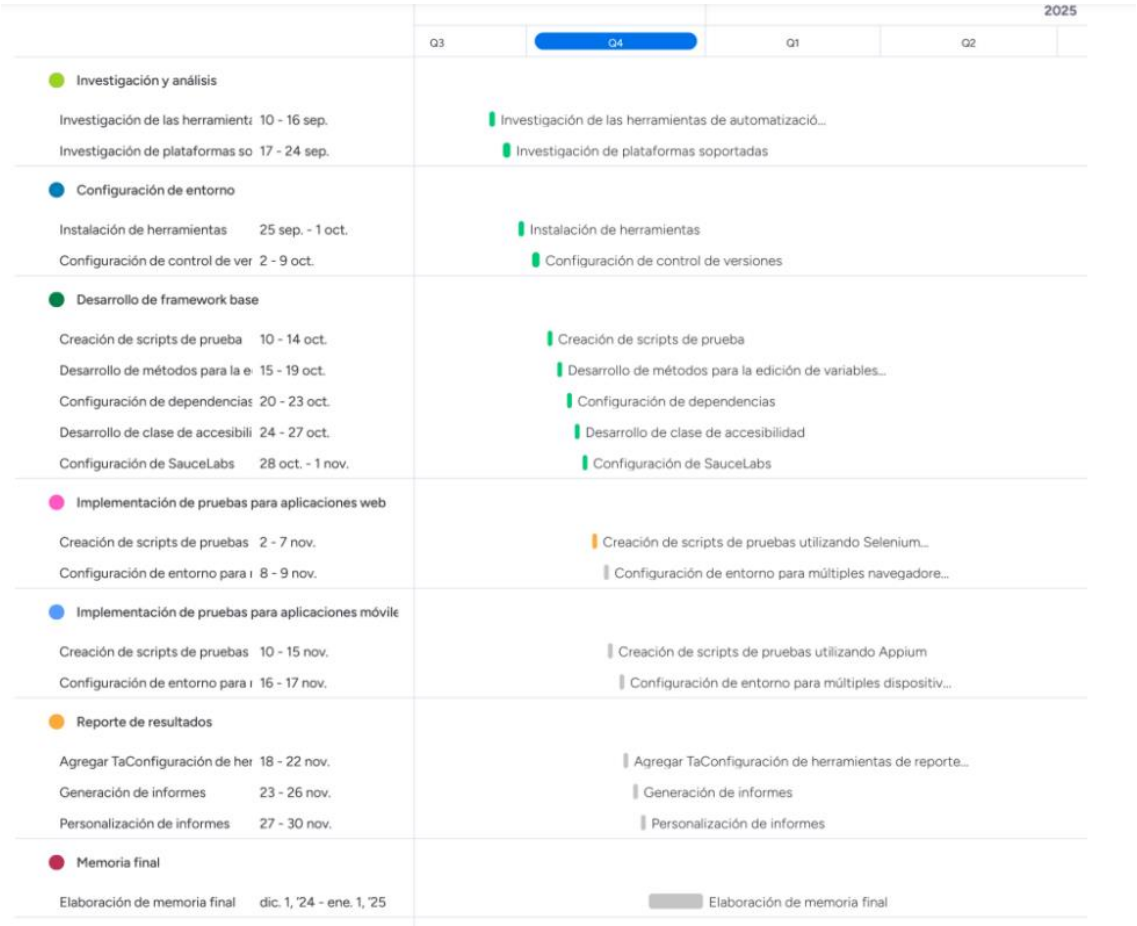


Figura 1- 1.1. Diagrama de Gantt de la planificación del TFG

1.4. Planificación del documento

En este proyecto se presenta una descripción detalla del proceso de comprobación de calidad de los proyectos software, comenzando por las bases de su origen, los distintos tipos y la herramienta de automatización realizada para comprobarlo. El documento se estructura de la siguiente manera:

- Capítulo 2: en este capítulo se ofrece una introducción sobre la calidad de software, sus características y definiciones importantes en el área de testing. También se explica la importancia de las pruebas en los productos software y en las compañías actuales, se describen los tipos de pruebas que se utilizan hoy en día y se destaca la automatización como el método más utilizado.
- Capítulo 3: en este capítulo se describen las herramientas utilizadas para el desarrollo del framework de automatización de pruebas. Se detallan la razones para su uso y por último se demuestra las distintas funcionalidades del framework con ejemplos de pruebas.
- Capítulo 4: se describen las conclusiones del trabajo y se exponen las dificultades de este.
- Capítulo 5: contiene el análisis de impacto.

2. Background

2.1. Calidad de software

2.1.1. Definición de calidad

La Real Academia Española de la Lengua define el concepto de "calidad" como [1]:

1. f. Propiedad o conjunto de propiedades inherentes a algo, que permit en juzgar su valor. *Esta tela es de buena calidad.*

2. f. Buena calidad, superioridad o excelencia. *La calidad de ese aceite h a conquistado los mercados.*

3. f. Adecuación de un producto o servicio a las características especific adas. *Control de la calidad de un producto.*

4. f. Carácter, genio, índole.

5. f. Condición o requisito que se pone en un contrato.

6. f. Estado de una persona, naturaleza, edad y demás circunstancias y condiciones que se requieren para un cargo o dignidad.

7. f. Nobleza del linaje.

8. f. Importancia o gravedad de algo.

9. f. pl. Prendas personales.

10. f. pl. Condiciones que se ponen en algunos juegos de naipes.

Mientras que la norma UNE-EN-ISO 9000:2005 la define como:

"Grado en que un conjunto de características inherentes cumple con los requisitos (UNE-EN-ISO 9000:2005) [2]."

Como podemos observar, existen muchas organizaciones e instituciones que poseen definiciones diferentes para el concepto de "calidad", y cada una se adapta a necesidades y objetivos específicos. Sin embargo, en el contexto de este trabajo es necesario adoptar una definición específica que se adapte al tema de estudio. En este caso, se tomará como adecuada el concepto de "calidad de software".

Se considerará como referencia la definición propuesta por la norma ISO 9000:2000.

En el ámbito del desarrollo de software, la calidad no se diferencia, en cuanto a objetivos, de la calidad de cualquier producto o servicio ofrecido por la industria tradicional. Según la ISO 9000:2000, la calidad se define por el grado en que un conjunto o características inherentes a un producto, servicio o proceso cumple con los requisitos establecidos [3]. Este enfoque resulta adaptable al desarrollo de software, ya que posee la necesidad de satisfacer un conjunto de requisitos específicos que son impuestos por las necesidades de los clientes o del contexto de uso.

Sin embargo, la calidad en el software presenta ciertas peculiaridades que la hacen compleja de definir y medir. A diferencia de otros productos de la industria tradicional, donde los estándares de calidad son más tangibles, los requisitos para determinar la calidad de programas informáticos suelen ser altamente subjetivos y difíciles de concretar [6]. Esto sucede porque los criterios

de evaluación de software incluyen aspectos técnicos y de experiencias de usuario, que pueden variar significativamente según las expectativas y necesidades del público.

Por esto, medir la calidad del software se traduce, en gran medida, en evaluar hasta qué punto los requisitos definidos para una aplicación específica son satisfechos por el producto final. Estos requisitos pueden abarcar dimensiones como funcionalidad, fiabilidad, usabilidad, eficiencia, mantenibilidad, portabilidad, etc.

2.1.2. Dimensiones de la calidad

El concepto de calidad en el contexto del desarrollo de software tiene un origen complejo, pero para verlo de una forma más sencilla, se puede visualizar la calidad como tres dimensiones que interactúan entre sí: calidad necesaria, calidad programada y calidad realizada. Estas dimensiones son necesarias al momento de decidir si se pueden satisfacer las necesidades del cliente y optimizar el desarrollo del trabajo.

La calidad necesaria es aquella que el cliente solicita o espera recibir [7]. Representa sus necesidades y expectativas, aunque a menudo estas no son del todo claras. Lograr una calidad adecuada requiere comprender las expectativas del cliente mediante la recolección de información. Sin embargo, estas expectativas no siempre coinciden con la calidad esperada, que es la percepción subjetiva que el cliente tiene sobre el producto o servicio antes de usarlo.

La calidad programada se puede definir como el estándar a seguir durante la planificación del producto inicial [7]. Este nivel es el objetivo que el desarrollador intenta alcanzar y se puede documentar con especificaciones técnicas. Para que esta dimensión sea alcanzada correctamente, es necesario que la documentación mencionada anteriormente sea lo más clara posible, para facilitar el trabajo de los desarrolladores y para que sirva como una guía para las personas responsables del proceso.

Por último, la calidad realizada es el nivel de calidad que se puede apreciar una vez el proyecto ha finalizado [7]. Este resultado depende completamente del trabajo realizado por los responsables del desarrollo y de los procesos del proyecto, ya que en ellos están las habilidades técnicas y los recursos y herramientas disponibles para este. Para mejorar esta dimensión en cualquier proyecto software es necesario potenciar las capacidades de los equipos, optimizar los medios utilizados para el desarrollo y mejorar los procesos que se encargan de mantener la calidad del producto.

Figura 2 - 2.1. Dimensiones de la calidad [4]

Las dimensiones antes descritas se pueden representar gráficamente como tres círculos que se interceptan, como se puede observar en la figura 1.1.. El objetivo de cualquier equipo de calidad es hacer que dichos círculos se unan, haciendo que el área común entre la calidad necesaria, la calidad programada y la calidad realizada sea la mayor posible, como se puede observar en la figura 1.2.. Que esto ocurra significaría que los objetivos de calidad establecidos por el cliente al principio del proyecto, son los estándares con los que se ha regido el proyecto durante todo su ciclo de vida. Fomentar este tipo de trabajo evita inconvenientes con el cliente y gastos innecesarios y al final añaden valor a la reputación de cualquier compañía y proyecto.

Figura 3 - 2.2. Objetivo de la gestión de calidad [5]

También, además de las dimensiones anteriormente mencionadas, existe la calidad percibida. La calidad percibida es una dimensión totalmente subjetiva, que depende principalmente de la opinión del cliente y, en la mayoría de los casos, está altamente alejada de la calidad realizada. Para reducir esta discrepancia y alinear la calidad percibida con la calidad realizada es necesaria una comunicación efectiva con el cliente y usuarios para así gestionar las expectativas de manera efectiva.

El gran desafío de los equipos que gestionan la calidad de los productos software es hacer que están dimensiones coincidan. Falta de comunicación con el cliente, especificaciones mal interpretadas y procesos poco eficientes pueden hacer que esto no suceda, por lo que es importante asegurar que un proyecto cumpla con los estándares de calidad establecidos y maximice la eficiencia de los equipos,

para así cumplir con las expectativas del cliente. Estos tres factores son esenciales para determinar el éxito en cuanto a la calidad de cualquier proyecto.

2.1.3. Características de la calidad de software

Como se comentó anteriormente, la calidad en el mundo del desarrollo software es un concepto difícil de definir. A pesar de esto, podemos coincidir en la pérdida de confianza que los nuevos productos software han sufrido desde que su uso empezó a ser un aspecto común en la vida cotidiana. En un estudio hecho en 2024, se reflejó que el nivel de confianza en productos de innovación software ha bajado una media de 8 puntos en un periodo de 5 años. Como podemos apreciar en la figura 1.3. se ha determinado que los nuevos productos software del mercado no cumplen con las expectativas de los usuarios y han creado una creciente falta de confianza en el sector [8].

	Global 24	Argentina	Australia	Brazil	Canada	China	Colombia	France	Germany	India	Indonesia	Ireland	Italy	Japan	Malaysia	Mexico	Netherlands	Saudi Arabia	Singapore	S. Africa	S. Korea	Spain	UAE	UK	U.S.
2019	62	63	41	74	52	87	71	47	46	79	79	38	65	56	74	73	51	70	66	51	69	63	74	48	50
2020	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2021	57	66	42	65	39	82	61	38	42	80	76	39	55	49	72	77	39	75	62	49	63	49	67	32	45
2022	58	63	42	66	34	89	62	39	40	82	82	39	57	49	73	68	45	78	62	47	68	54	78	37	41
2023	54	56	34	58	36	88	60	34	36	77	76	37	51	49	73	69	41	72	59	51	61	46	71	29	43
2024	54	49	35	57	31	86	57	35	34	78	77	30	50	52	78	63	38	72	60	50	69	44	76	33	35
5-yr change	-8*	-14*	-6	-17*	-21*	-1	-14*	-12*	-12	-1	-2	-8	-15*	-4	4	-10	-13*	2	-6	-1	0	-19*	2	-15*	-15*

Figura 4 - 2.3. Niveles de Confianza en Empresas Tecnológicas [8]

Para continuar con la definición de calidad se explicarán las características de esta, abordando la norma ISO 9126, que define 6 características fundamentales para su entendimiento: funcionalidad, fiabilidad, usabilidad, eficiencia, mantenibilidad y portabilidad.

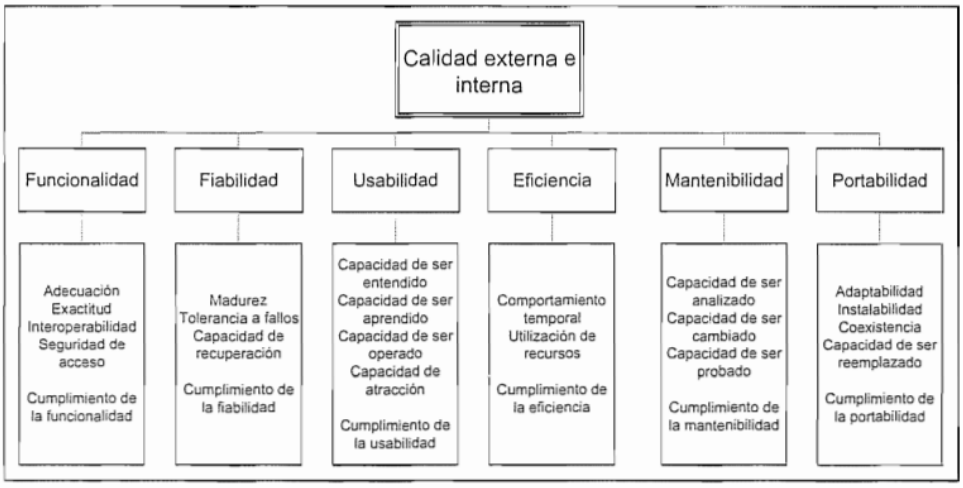


Figura 5 - 2.4. Modelo para la calidad interna y externa [9]

2.1.3.1. Fiabilidad

Al momento de adquirir un nuevo producto software, se espera que este cumpla con ciertos criterios. Cuando se habla de fiabilidad en este contexto, se refiere a que el software debe ser capaz de funcionar siempre que sea necesario [10]. Este es un aspecto crucial en cuanto al valor de un producto, ya que la incapacidad de su funcionamiento puede llevar a la falta de confianza en este o perdidas financiera u operativas. Según la norma ISO 9126 esta característica incluye:

- Madurez: indica la capacidad que tiene el software de minimizar fallos por defectos.
- Tolerancia a fallos: se refiere a la capacidad que permite que el sistema continúe funcionando, incluso cuando se presentan fallos parciales.
- Capacidad de recuperación: es la garantía de la recuperación de datos tras un incidente y la continuación del funcionamiento del sistema.
- Cumplimiento normativo: se asegura que se respeten las normas relacionadas con la fiabilidad.

2.1.3.2. Funcionalidad

La fiabilidad se refiere a la capacidad que posee el software de realizar las tareas que se supone que debe realizar [10]. Esto no solo implica su correcto funcionamiento, también incluye la entrega de la información requerida por el usuario, y la garantía de que su funcionamiento no va a ser comprometido. Es necesario que el sistema sea de fácil uso y que incluya manuales para facilitar el entendimiento para los nuevos usuarios. Dentro de esta característica existe:

- Adecuación: el software debe tener funciones relevantes a su objetivo.
- Exactitud: los resultados deben ser confiables y entendibles para el usuario.
- Interoperabilidad: se debe asegurar que el software es capaz de integrarse con diferentes dispositivos y funcionar en diversos ambientes.
- Seguridad: es crucial proteger los datos de los usuarios y restringir los accesos no autorizados.
- Cumplimiento normativo: el software debe adherirse a los estándares establecidos al principio del proyecto.

2.1.3.3. Usabilidad

La usabilidad garantiza la facilidad de entendimiento que tiene un producto software, con esto se refiere a la adecuación al sistema que tienen los nuevos usuarios y si en general es un producto agradable de usar [10]. Esta característica va enfocada principalmente a la interfaz de usuario, se asegura que esta sea intuitiva, accesible y que cumpla con las necesidades del cliente y usuario. Al asegurar todos estos aspectos, se mejora la percepción que tiene usuario, aumentando la confianza y disminuyendo el esfuerzo requerido para utilizarlo. Los componentes de esta característica son:

- Comprensibilidad: es la capacidad que tiene el sistema de transmitirle al usuario el propósito del software y su forma de uso.
- Facilidad de aprendizaje: se refiere a la facilidad que transmite el sistema. El proceso de aprendizaje debe de ser rápido e intuitivo.
- Operatividad: facilidad del usuario en el control y uso del sistema.
- Atractivo: inclusión de elementos visuales y de diseño que mejoren la experiencia general del usuario.
- Cumplimiento normativo: el software debe adherirse a los estándares establecidos al principio del proyecto.

Las características restantes, la eficiencia, la mantenibilidad y la portabilidad, son características secundarias que complementan a las anteriores. Estas no tienen un carácter prioritario a la hora del desarrollo de un programa, pero aumentan la calidad y el valor de cualquier producto.

2.1.3.4. Eficiencia

La eficiencia mide el rendimiento del software a través del uso de recursos y tiempos de respuestas [10]. Las características que acompañan a la eficiencia son:

- Desempeño temporal: es la medida que evalúa la rapidez con la que el software responde a las solicitudes.
- Optimización de recursos: garantiza el uso eficiente de recursos como memoria, CPU y energía.
- Cumplimiento normativo: el software debe adherirse a los estándares establecidos al principio del proyecto.

2.1.3.5. Mantenibilidad

La mantenibilidad se refiere a la facilidad que poseen los desarrolladores del proyecto a la hora de modificar el código. Se puede modificar el código por diferentes razones, ubicación de errores, adaptabilidad, la implementación de nuevos requerimientos o la mejora de su rendimiento [10]. Esta característica incluye:

- Capacidad de análisis: es la facilidad para la identificación de problemas o áreas de mejora.
- Capacidad de modificación: es la capacidad de realizar cambios especificos sin la necesidad de modificar el código completo.
- Estabilidad: minimiza los cambios secundarios tras una modificación.
- Facilidad para pruebas: simplifica la verificación de los cambios implementados.
- Cumplimiento normativos: el software debe adherirse a los estándares establecidos al principio del proyecto.

2.1.3.6. Portabilidad

La portabilidad significa la facilidad con la que el sistemas es capaz de funcionar en diferentes entornos, como plataformas, sistemas operativos, etc [10]. Sus características son:

- Adaptabilidad: permite que el software funcione en diferentes entornos sin modificaciones significativas.
- Facilidad de instalación: asegura que el proceso de instalación sea sencillo.
- Coexistencia: se refiere a la garantía de que el software funcione junto a otras aplicaciones en un entorno compartido.
- Cumplimiento normativo: el software debe adherirse a los estándares establecidos al principio del proyecto.

Este modelo de calidad es uno de los más utilizados en el desarrollo de software y conociendo e implementado las características mencionada se facilita su evaluación y proporciona una guía a la hora de su desarrollo.

2.2. Testing

2.2.1. Concepto de testing

El testing es una de las actividades principales en el ciclo de vida del desarrollo de software, ya que se encarga de garantizar que los sistemas desarrollados funcionen correctamente y según lo especificado. En esta etapa se detectan y corrigen errores antes de que el producto llegue al usuario final. Esta etapa ha sufrido cambios a lo largo de los años y ha evolucionado en cuanto a técnicas y herramientas.

Según el autor del libro The Art of Software Testing [11], Glenford J. Myers, el testing no debe ser entendido como un proceso que se dedica a demostrar que un sistema funciona correctamente. Por el contrario, el autor indica que los conceptos tradicionales del testing, como "el proceso de demostrar que no hay errores presentes" o "establecer confianza en que un programa realiza sus funciones correctamente", son incorrectas o, por lo menos, no reflejan completamente las funciones del testing. Estas definiciones dirigen la atención a la validación del software, en lugar de fomentar un análisis más crítico del código.

Por otro lado, Myers define el testing como "el proceso de ejecutar un programa con la intención de encontrar errores". Este enfoque parte de la premisa de que todo programa tiene defectos y que el propósito del testing es descubrir y corregir la mayor cantidad posible de ellos, elevando así la calidad y fiabilidad del producto. Esta definición, aunque parezca simple, ha ayudado a ver desde una nueva perspectiva el testing de software. Si los testers se enfocan únicamente en validar que el programa funciona, podrían seleccionar subconscientemente casos de prueba menos probables a generar fallos. Con esta nueva definición, el objetivo es encontrar errores, por lo que se crearan pruebas que se enfoquen en detectarlos.

El ISTQB (International Software Testing Qualification Board), una organización que se encarga de crear estándares de calidad en el testing de software, ofrece también una definición complementaria. Según esta organización, las pruebas son "el proceso que abarca actividades estáticas y dinámicas relacionadas con la planificación, preparación y evaluación de productos de software para determinar si cumplen con los requisitos especificados, si son aptos para su propósito y para identificar defectos". Como se mencionó antes, esta definición complementa a las antes vistas, ya que añade aspectos como la planificación y destaca la importancia de crear pruebas que se alineen con los objetivos del cliente.

En este contexto, el testing juega un papel fundamente en el desarrollo de cualquier producto software, ya que asegura la ausencia de fallos en el proyecto y garantiza su calidad en diferentes aspectos. El testing también actúa como un mecanismo de retroalimentación continuo, que permite verificar el cumplimiento de requisitos y la alineación del producto con las expectativas del cliente.

En conclusión, el testing no es solo una herramienta para verificar la funcionalidad de un producto software; es un proceso que busca agregar valor, mejorar la calidad y reducir riesgos. Adoptar una mentalidad critica enfocada a la detección de errores, como propone Myers, junto a las metodologías propuestas por organizaciones como ISTQB, permite maximizar el alcance del testing de productos software.

2.2.2. Términos del testing

En el contexto de testing debemos saber diferenciar múltiples términos, pero entre ellos, los más importantes son: error, defecto y fallo, que, aunque estén relacionados, tienen diferentes significados en el desarrollo y evaluación de software. Según la organización de ISTQB, estos conceptos están conectados en el siguiente flujo: (mostrado en la figura 2.1.) un error humano es capaz de producir un defecto en el código o en la documentación de un software y, si este defecto se ejecuta puede producir un fallo en el sistema. Este es el flujo que normalmente siguen los sistemas, aunque es verdad que no todos los defectos producen fallos necesariamente [13].

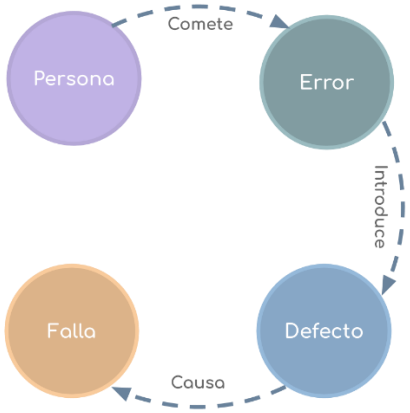


Figura 6 - 2.5. Flujo de fallos en productos software [14]

2.2.2.1. Error

Un error se puede definir como una acción humana incorrecta. Esta puede tener diferentes razones, como una mala interpretación de requisitos, falta de detalle en el diseño del sistema o simplemente un descuido al momento de escribir el código. Por ejemplo, la implementación de un método en el sistema que tenga una lógica que difiere a las establecidas en los requisitos, puede afectar el comportamiento del software. Si este error no es detectado puede convertirse en el origen de muchos otros problemas del sistema.

2.2.2.2. Defecto

Los defectos, también conocidos como bugs, es el resultado de los errores humanos. Los defectos no solo pueden ser encontrados en el código, también puede hallarse en la documentación, en las especificaciones del cliente o en cualquier parte del proyecto que comprometa el funcionamiento del sistema. Aunque no en todos los casos los defectos producen un fallo visible, en todos los casos significan un desvío en el comportamiento esperado, si no se detecta en el momento oportuno puede llegar a causar problemas significativos en un futuro.

2.2.2.3. Fallo

Un fallo es un defecto que no ha sido identificado y al momento de ser ejecutado ha desatado un fallo en el sistema. Se traduce en un comportamiento inesperado del programa y se puede presentar como resultados incorrectos o como acciones no deseadas. Por ejemplo, si en un sistema crítico como una central eléctrica el defecto no se detecta y corrige, el fallo resultante podría tener consecuencias considerables. Por otro lado, si el fallo proviniera de un sistema

de uso cotidiano, este puede derivar a la disminución de confianza en el producto y opiniones negativas sobre la compañía de origen.

Aprender a diferenciar estos términos en el área del testing es fundamental, ya que, aunque parezca que tienen la misma definición, sus significados marcan la diferencia en proyectos de testing. Saber definirlos e identificarlos es fundamental para evitar y corregir defectos antes de que se conviertan en fallos.

2.2.3. Importancia del testing

En la actualidad vivimos rodeados de la presencia de sistemas de software, ya sea en automóviles, aplicaciones, teléfonos inteligentes, entre otros. Este nivel de dependencia destaca la importancia del testing. Este tiene un papel fundamental en la prevención de fallos que puedan tener consecuencias graves, tanto económicas como humanas. A lo largo de la historia, la falta de un proceso de testing exhaustivo ha llegado a causar incidentes que sirven de ejemplo para demostrar las consecuencias que tiene ignorar esta etapa del ciclo de vida de un sistema. Entre los más populares se encuentran:

2.2.3.1. Therac-25 (1985)

El Therac-25 fue un sistema de radioterapia producido por Atomic Energy of Canada Limited (AECL). Este sistema, en el año 1985, administró de manera errada una sobredosis de radiación a al menos 6 pacientes, causándoles lesiones tan graves, que en algunos casos causaron la muerte. El testing realizado en el sistema se limitó solo a la parte del software, ya que desde ahí se determinaban las dosis de radiación para cada paciente. No se tuvieron en cuenta los errores provenientes del hardware de la máquina, por lo que no se creó un sistema que gestionara ese tipo de fallos. Cuando el Therac-25 empezó a suministrar dosis altamente peligrosas a los pacientes, no existía un bloqueo físico que lo impidiera, y como no se había considerado este fallo en el testing del programa, tampoco había forma de comunicar el problema a los operadores de la máquina.

Este ejemplo destaca la importancia del testing en sistemas que integran tanto software como hardware. Si un proyecto software depende de una parte hardware, hay que extender el proceso de testing a ambas áreas, ya que en ambas se puede originar un fallo que afecte el funcionamiento del sistema [15].

2.2.3.2. Ariane 5, vuelo 501 (1996)

El Ariane 5, designado como vuelo 501, fue un cohete que despegaba con el propósito de transportar una carga útil valorada en 370 millones de dólares a la atmósfera. El 4 de junio de 1996, un fallo causó la destrucción del cohete y de su carga 40 segundos después de su despegue. El incidente fue provocado por un fallo en el software del Sistema de Referencia Inercial (SRI), que generó una excepción que no había sido contemplada en el proceso de testing. Este fallo, originado por la reutilización del código utilizado en un modelo anterior (Ariane 4), llevo a la perdida de los datos de orientación y control del lanzador, lo que causo finalmente la destrucción del cohete.

El caso del Ariane 5, destaca los riesgo de la reutilización de código sin un previo proceso de testing. El testing de dicho código en el nuevo entorno hubiera sido suficiente para evitar el desastre [16].

2.2.3.3. Boeing 737 MAX

Entre 2018 y 2019, ocurrieron dos accidentes fatales que involucraron el Boeing 737 MAX, y resultaron en la muerte de 346 personas. Las investigaciones realizadas después de los accidentes determinaron que el fallo se encontraba en Sistema de Aumento de las Características de Maniobra (MCAS). Este sistema es el encargado de mejorar las maniobras mediante ajustes automáticos del estabilizador horizontal, estos ajustes se hacen con los datos enviados por el sensor de ángulo de ataque (AoA). En el caso de los Boeing 737 MAX, este sensor se activaba de manera errónea y enviaba los datos al MCAS. Como respuesta el MCAS forzaba la nariz del avión hacia abajo provocando la caída de este, sin importar los esfuerzos realizados por el piloto. Después de los accidentes y de que error del sensor fuese localizado, las flotas de Boeing 737 MAX fueron inmovilizadas a nivel mundial. Boeing implementó una actualización del software utilizado para el MCAS, añadió un nuevo control de errores para la lectura de datos del sensor AoA y mejoró la capacitación de los pilotos [17].

Este ejemplo fue un caso conocido a nivel mundial y el error le costó a Boeing su reputación como aerolínea confiable, además de haberle costado millones de dólares en pérdidas materiales y las 346 muertes que estos accidentes causaron.

Todos los ejemplos mencionados demuestran como la falta de pruebas de sistemas software puede perjudicar de múltiples maneras nuestras vidas. Hoy en día la mayoría de las empresas respaldan sus productos y servicios en sistemas de software, por eso es importante asegurarse de que estos cumplan con sus funcionamientos y evaluar los riesgos que los fallos en estos puedan provocar.

2.3. Pruebas de Software

El objetivo principal de las pruebas de software es identificar defectos antes de que estos se conviertan en fallos y garantizar que el software cumpla con los requisitos establecidos. Para realizar las pruebas, se diseñan y ejecutan los casos de pruebas definidos por el cliente y desarrolladores al principio del proyecto, los cuales permiten evaluar y comprobar el funcionamiento del sistema bajo diferentes condiciones. Estos casos de pruebas se elaboran de manera que se maximice la eficiencia en la detecciones de errores.

Una de las cuestiones que se presenta más frecuentemente a la hora de iniciar el proceso de pruebas de software es: ¿es posible encontrar todos los errores de un programa?. Según el autor del libro "The Art of Software Testing" [18], Glenford J. Myers y el autor del libro "Ingeniería de Software: Un Enfoque Practico" [19], Roger Pressman, la respuesta es no. Intentar identificar todos los fallos o defectos de un sistemas en inviable, debido a que hacerlo requeriría muchísimo tiempo y recursos. Además, con la creciente complejidad de los sistemas modernos, sería irrealizable.

Las pruebas de software exhaustivas, que implica evaluar todas las entradas, combinaciones y cambios posibles de un programa, son simplemente imprácticas. El número de combinaciones posibles en sistemas complejos crece exponencialmente, por lo que sería trabajoso y costoso la realización de estas pruebas.

Además de esto, se suma la necesidad de considerar múltiples escenarios, como entradas válidas e inválidas, configuraciones de entorno e interacciones con otros sistemas. Por eso, el objetivo de las pruebas no es garantizar que el software esté libre de defectos, si no identificar las áreas de riesgo y reducirlas a un nivel aceptable y seguro, para así lograr un producto confiable para el usuario.

Para poder identificar dichos riesgo es necesario establecer las prioridades del proyecto. Según Gerald Everett y Raymond McLeod, autores del libro "Software Testing: Testing Across the Entire Software Development Life Cycle" [20], estas prioridades se basan en:

1. Identificar los riesgos del proyecto que pueden reducirse mediante la realización de pruebas.
2. Diseñar pruebas enfocadas a reducir dichos riesgos.
3. Determinar criterios que definan cuando las pruebas han alcanzado un nivel aceptable.
4. Gestionar las pruebas del proyecto como parte esencial del desarrollo del software.

Para maximizar la eficacia de las pruebas, es necesario seguir los pasos mencionados y asignar los recursos necesarios a ellas, ya sea tiempo o herramientas. Para esto, el Principio de Pareto o "Regla del 80/20" [21] resulta especialmente útil: el 80% de los fallos del software suele ser causado por solo el 20% del código.

Este principio es una herramienta visual que le permite a los equipos de testing representar los defectos más frecuentes durante el desarrollo de programas. En el diagrama se organiza la información de la siguiente manera: cada barra representa un categoría de defecto, como por ejemplo, errores de lógica, problemas de interfaz o fallos de rendimiento. El diagrama también incluye una línea acumulativa que representa la categoría que genera más problemas.

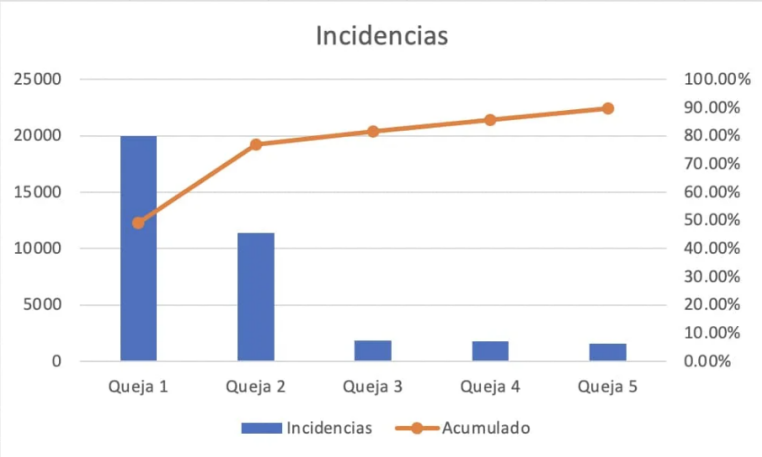


Figura 7 – 2.6. Ejemplo de Diagrama de Pareto [22]

En la imagen podemos observar como un pequeño grupo de categorías genera la mayoría de los incidentes. Por ejemplo, en un sistema de gestión empresarial, los defectos relacionadas con el módulo de variables de entorno y el módulo de drivers (Queja 1 y 2) representan el 80% de los problemas, aunque solo comprenden el 40% de las categorías totales. Según lo establecido en el principio de Pareto, la idea es enfocar los esfuerzos en estas áreas, ya que tendría el mayor impacto en la mejora del proyecto, mientras que el resto de áreas, se pueden gestionar, pero con menor urgencia [21].

Este principio además de ayudar a ubicar los recursos en las áreas en las que se necesitan, también puede aportar valor a la hora de seleccionar los tipos de pruebas que pueden aplicarse en cada caso.

2.3.1. Tipos de pruebas de software

Existen diferentes tipos de pruebas de software, cada una creada para probar diferentes aspectos del sistema. Según el nivel de evaluación, se pueden aplicar pruebas que prueben componentes individuales y su funcionamiento hasta sistemas completos y su interacción. Las pruebas se pueden clasificar en:

- Pruebas según el enfoque.
- Pruebas según el nivel.

2.3.1.1. Pruebas según el enfoque

Pruebas funcionales

Las pruebas funcionales en el contexto del desarrollo de software, se encargan de garantizar que el sistema cumpla con las especificaciones requeridas por el cliente. Este tipo de pruebas cambia las entradas del programa y verifica que la salida sea la correcta para verificar que el sistema funcione correctamente bajo cualquier configuración.

Las pruebas funcionales, además de lo mencionado anteriormente, tienen como objetivo comprobar el funcionamiento de los requisitos documentados. Estas comprobaciones dependen principalmente de la calidad de dichos requisitos, ya que una mala implementación de estos, puede llevar a pruebas que no comprueben las especificaciones correctas.

Según la norma ISO/IEC 25010, la funcionalidad de un producto software se define como la capacidad que tiene el producto para proporcionar funciones que

satisfagan las necesidades declaradas e implícitas de los usuarios. En la norma se reflejan las siguientes características de la funcionalidad del software [23]:

- Completitud funcional: mide hasta qué punto las funcionalidades del sistema cubren todos los objetivos especificados.
- Corrección funcional: evalúa si el software proporciona resultados exactos y correctos.
- Pertinencia funcional: analiza si las funciones definidas son las indicadas para las tareas de los usuarios.

Todas están características sirven como base para el diseño de las pruebas funcionales. Una vez terminada esta etapa, se puede pasar a la planificación de las pruebas. Para garantizar que el software cumple con lo establecido se pueden seguir los siguientes pasos:

- Identificación de requisitos: identificar todos los requisitos funcionales documentados en las especificaciones y aquellos que se deduzcan de las expectativas del cliente.
- Diseño de casos de prueba: crear casos de pruebas que cubran todas las funciones principales y sus posibles combinaciones. Para cada caso se deben incluir flujos específicos, las entradas válidas e inválidas y los resultados esperados.
- Ejecución de pruebas: evaluar cada funcionalidad del sistema, comparando los resultados obtenidos con los esperados y registrando cualquier defecto que se encuentre.

Este tipo de pruebas se pueden aplicar en cualquier ciclo de pruebas, desde pruebas unitarias (para validar funcionalidades individuales), hasta pruebas de integración (para validar funciones que dependen de diferentes módulos) [24].

Aunque son bastante multifacéticas, el reto principal de las pruebas funcionales es asegurar que la prueba tiene una cobertura completa de la funcionalidad a evaluar (sin que esto resulte un esfuerzo desproporcionado), sobre todo cuando se está trabajando con sistemas complejos, con múltiples combinaciones de entradas e infinitas salidas. Para este tipo de casos es mejor utilizar criterios como la frecuencia de uso, riesgos asociados o el Principio de Pareto y dar prioridad a las partes del código que presentan más problemas.

A pesar de esto, también garantizan la confianza en cuanto al cumplimiento de requisitos, aumentan la calidad del producto y ayudan a identificar errores en el funcionamiento del código, evitando fallos en etapas más avanzadas del proyecto

Otro aspecto importante de estas pruebas es que no son una actividad aislada, de hecho están estrechamente relacionadas con la gestión de requisitos, el diseño de la arquitectura del sistema y la validación final del producto.

Pruebas no funcionales

Las pruebas no funcionales, evalúan aspectos del sistema que no están relacionados con la funcionalidad del código, en cambio comprueban aspectos como la calidad y el comportamiento en general del sistema. Este tipo de pruebas se centra en cómo funciona el sistema, evaluando temas como el rendimiento, la usabilidad, la fiabilidad y la seguridad.

En las especificaciones del producto deben estar establecidas tanto las características funcionales como las no funcionales. La norma ISO/IEC 25010

establece un modelo para guiar la calidad de un producto software, que además de incluir la adecuación funcional, contiene otras siete características que ayudan a moldear las especificaciones de las pruebas no funcionales [23]:

- Eficiencia del desempeño: es la capacidad del sistema para proporcionar un rendimiento adecuado en relación con los recursos dados.
- Compatibilidad: se refiere a la capacidad que tiene el sistema de operar con diferentes entornos.
- Usabilidad: es la facilidad con la que los usuarios pueden aprender a utilizar el sistema.
- Fiabilidad: es la capacidad que posee el sistema para mantener un nivel de desempeño bajo determinadas condiciones.
- Seguridad: es la protección que proporciona el sistema contra accesos no autorizados y la garantía de la integridad de los datos de los usuarios.
- Mantenibilidad: se refiere a la facilidad con la que el sistema puede ser modificado para corregir defectos u otras características.
- Portabilidad: indica la capacidad del sistema para ser utilizado en un entorno u otro.

Dentro del ámbito de las pruebas no funcionales, se destacan varios tipos específicos [25].

- Pruebas de carga: evalúan el comportamiento del sistema al aumentar la carga de, por ejemplo, el número de usuarios simultáneos o la cantidad de transacciones procesadas.
- Pruebas de rendimiento: miden la velocidad de procesamiento y el tiempo de respuesta del sistema.
- Pruebas de volumen: analizan la capacidad del sistema para manejar grandes cantidades de datos.
- Pruebas de esfuerzo: determinan cómo el sistema se comporta bajo condiciones de sobrecarga y su capacidad de recuperación.
- Pruebas de seguridad: verifican la protección del sistema contra accesos no autorizados, ataques u otras amenazas.
- Pruebas de estabilidad y robustez: evalúan la respuesta del sistema ante fallos.
- Pruebas de compatibilidad: Aseguran que el sistema funcione correctamente en diferentes entornos.
- Pruebas de usabilidad: miden la facilidad de uso y satisfacción de los usuarios al interactuar con el sistema.

La implementación de todos los aspectos mencionados, mejora la calidad del producto y reduce los riesgos asociados a su operación en condiciones reales.

Es importante mencionar, que este tipo de pruebas complementan a las pruebas funcionales, ya que con ambas se puede dar una visión integral del estado del proyecto y se asegura que el sistema haga lo que debe hacer de manera efectiva.

Pruebas estructurales

Las pruebas estructurales son las encargadas de realizar un análisis del código fuente del sistema, para así verificar que cada ruta posible de este se ejecute y funcione según los requisitos establecidos. A diferencia de las pruebas funcionales, que se centran mayormente en comprobar las funcionalidades del sistema, el objetivo de las pruebas estructurales es asegurarse que todas las

estructuras del programa funcionen correctamente y a través de esta verificación identificar errores de lógica [12].

Para llevar a cabo las pruebas estructurales, se emplean diferentes técnicas para la evaluación del código.

- Cobertura de sentencias: verifica que cada línea de código se ejecuta al menos una vez durante las pruebas.
- Cobertura de decisiones: garantiza que todas las decisiones lógicas (como las condiciones if o switch) se evalúen en ambas direcciones (verdadero o falso).
- Cobertura de Condiciones: analiza cada condición individual dentro de una condición compuesta.
- Cobertura de Caminos: examina todas las rutas posibles que se pueden tomar a través del código.

La adopción de estas técnicas para la realización de pruebas estructurales durante el desarrollo de software contribuye al aumento de la calidad del producto y a su sostenibilidad en el tiempo. Uno de los principales aportes es la detección temprana de errores, lo que evita que estos se propaguen a fases más avanzadas del desarrollo, reduciendo los costos asociados a su resolución.

La realización de este tipo de pruebas puede dar una visión detallada del estado del código por su naturaleza estructurada y por su gran abarque. Además, fomenta la implementación de código limpio e indirectamente facilita el mantenimiento del proyecto.

2.3.1.2. Pruebas según el nivel

Pruebas unitarias

Las pruebas unitarias representan el nivel más básico de las pruebas de software según el nivel, ya que se centran en verificar unidades individuales de código, como funciones, métodos o clases. En el libro "The Art of Software Testing", estas pruebas son fundamentales para la identificación de errores en etapas tempranas, justo cuando resulta menos costosa y más eficiente [11].

Estas pruebas se realizan para garantizar que cada pieza de código individual funciona según lo establecido y valida que los resultado generados a partir de esta sean los esperados según su entrada. Estas pruebas suelen realizarse utilizando herramientas como Junit para Java o PyTest para Python, que permite la ejecución repetitiva de los casos de pruebas.

Una de las principales ventajas es que fomentan la calidad del código promoviendo buenas prácticas de programación, como la modularidad y el desacoplamiento. Cuando se enfoca el desarrollo de una manera modular y se crean los componentes individualmente, los desarrolladores pueden aislar y solucionar defectos de manera más sencilla. También al momento de introducir cambios, facilita su verificación.

Las pruebas unitarias sirven como base para las siguientes etapas de pruebas, como las pruebas de integración y sistemas, ya que garantiza que cada bloque individual funcione correctamente, antes de realizar pruebas que comprueben su funcionamiento en conjunto. Es importante, como en todas las pruebas, que las pruebas unitarias estén bien diseñadas y alineadas con los requisitos, ya que de esto depende su eficacia y la eficacia de las pruebas posteriores a esta.

Pruebas de integración

Las pruebas de integración evalúan cómo interactúan y funcionan en conjunto los diferentes módulos o componentes del sistema. Con esto, se busca identificar defectos en las interfaces y conexiones entre los módulos, asegurando que trabajen de manera conjunta como un sistema cohesivo [11].

El enfoque principal de las pruebas de integración es descubrir problemas derivados de la interacción entre módulos previamente probados de manera individual. Para lograr esto, existen varias formas de realizar las pruebas, entre ellas están:

- Integración ascendente: se comienza probando los módulos más básicos y se continua con los módulos más complejos.
- Integración descendente: se inicia con los módulos de más complejos del sistema y posteriormente se agregan los más básicos.
- Integración incremental: los módulos se prueban gradualmente, en grupos pequeños o uno por uno.
- Integración en grandes bloques: todos los módulos se prueban juntos desde el principio de las pruebas. Esta forma no es recomendada, debido a que puede dificultar la ubicación de errores.

El desafío que presentan este tipo de pruebas es que a medida que se agregan módulos, crece la complejidad de estas. Para manejar esta complejidad es necesario tener un planificación y casos de pruebas detallados y unas herramientas y entornos de pruebas que simulen las condiciones reales en las que se utilizará el sistema.

El éxito de las pruebas de integración radical en garantizar que los módulos probado cumplan con su objetivo y colaboren para cumplir los objetivos globales del proyecto.

Pruebas de Sistema

Las pruebas de sistema son cruciales a la hora de validar un producto software final, estas pruebas se enfocan en evaluar el sistema completo y verifican que cumpla con todos los requisitos funcionales y no funcionales. Estas pruebas analizan el software como un todo, comprueba que las interacciones entre módulos sean correctas y que las funciones globales del sistemas operen según lo establecido en un entorno que simule uno real [11].

Estas pruebas podrían considerarse una unión de las dos pruebas anteriores, pero de una manera más global, ya que verifican aspectos funcionales y no funcionales y también comprueba el rendimiento, la seguridad y la usabilidad.

Las pruebas de sistema abarcan varios aspectos:

- Escenarios reales de uso: se diseñan casos de prueba que repliquen las condiciones operativas en las que el software será utilizado.
- Cobertura amplia: a diferencia de las pruebas unitarias o de integración, las pruebas de sistema evalúa la funcionalidad global del software, estas pruebas verifican cómo interactúan todos los componentes entre sí y con el entorno operativo.
- Validación de requisitos no funcionales: estas pruebas incluyen pruebas de rendimiento, estabilidad, seguridad y compatibilidad para asegurar que el sistema funcione bajo diferentes condiciones.

El gran desafío de estas pruebas es la creación de casos de pruebas detallados que cubran todos los escenarios posibles y los diferentes entornos en los que puede ser ejecutado el programa. Crear y probar todos esos casos de pruebas puede conllevar un gran esfuerzo en términos de recursos y tiempo.

A pesar de esto, su realización puede ofrecer diversas ventajas al proyecto, como:

- Garantizan que el software funcione correctamente como un todo.
- Ayudan a identificar problemas que no suelen ser detectados en pruebas unitarias o de integración.
- Mejoran la confianza del cliente al asegurar que el producto final cumple con los estándares de calidad y los requisitos especificados.

Pruebas de aceptación

Las pruebas de aceptación son las últimas pruebas realizadas en el ciclo de pruebas software. Su propósito es asegurar que el producto cumple con las expectativas del cliente y del usuario final. A diferencia de las pruebas mencionadas anteriormente, estas se centran en comprobar que el software es funcional y que está listo para su uso en un entorno real [11].

Estas pruebas no se enfocan en detectar fallos, sino en garantizar que el producto le entregue valor al usuario. Básicamente, es la confirmación final que se le hacen al proyecto antes de ser entregado. Las pruebas de aceptación incluyen verificaciones como, validación de resultados esperados, verificación de procesos y comprobación de que el sistema cumple como con los criterios establecidos.

Estas pruebas normalmente son realizadas por el cliente, el usuario final o un equipo que los represente. En muchas ocasiones, el desarrollo de los casos de prueba se realiza en colaboración con los usuarios, para asegurar que se desarrollen casos que prueben que sus expectativas se cumplen.

La ejecución incluyen:

- Validación de procesos clave
- Evaluación de resultados obtenidos frente a resultados esperados
- Identificación de discrepancia en la implementación en comparación con las necesidades reales.

Las pruebas de aceptación pueden adoptar varias formas dependiendo del contexto del proyecto, entre ellas existe:

- Aceptación de Usuario (UAT): enfocadas en comprobar que el software satisface los requerimiento funcionales y operativos del usuario final.
- Aceptación Operativa (OAT): diseñadas para validar que el sistema es adecuado para el entorno de producción.
- Pruebas Alfa y Beta: las pruebas alfa son pruebas que se realizan en un entorno controlado y simulando condiciones reales, mientras que las pruebas beta son ejecutadas por un grupo de usuarios finales en un entorno real, pero antes del lanzamiento oficial.

2.3.2. Ciclo de vida del software

Muchas veces se ha mencionado el ciclo de vida de software y tras analizar los diferentes tipos de pruebas, se considera importante ubicarlas dentro de este.

El ciclo de vida es una estructura que se sigue al momento de la creación de cualquier producto software. Según el autor del libro “Software Engineering in the UNIX/C Environment” [77], Charles Fox, el ciclo de vida del software se puede definir como las distintas fases por las que pasa el software desde que nace la necesidad de mecanizar un proceso, hasta que deja de utilizarse.

Figura 8 – 2.7. Ciclo de vida del software [78]

Este ciclo se compone de varias etapas [79]:

- Fase de planificación: en esta etapa se establecen los objetivos y necesidades del proyecto. Se realizan estudios que analizan posibles riesgos y se planifican las fases siguientes.
- Fase de análisis: se definen las funciones específicas que el software debe ejecutar y sus características.
- Diseño y estructura: con la información recopilada, se diseña la estructura de la base de datos, la lógica de datos y la interfaz de usuario.
- Fase de desarrollo: se inicia la programación del software.
- Fase de pruebas: se realizan las pruebas que identifiquen y corrijan errores o defectos no detectados. En esta fase se garantiza la calidad del producto antes de su implementación final.
- Fase de mantenimiento: con el producto software en funcionamiento, se monitorea su desempeño para identificar limitaciones o áreas de mejora.

2.4. Automatización de pruebas

2.4.1. Definición y propósito

La automatización de pruebas es el proceso de emplear herramientas y scripts para ejecutar pruebas de software de forma automática, con el objetivo de verificar que una aplicación funcione correctamente. Las pruebas automatizadas complementan a las pruebas manuales al permitir una ejecución rápida y repetitiva de los casos de pruebas, lo que funciona especialmente bien en entornos ágiles [26].

La automatización de pruebas mejora la eficiencia de los equipos y asegura la efectividad de los procesos que aseguran la calidad del producto. Al automatizar tareas repetitivas y propensas a sufrir errores, se elimina por completo la posibilidad de defectos provocados por errores humanos, se logra una detección más temprana, en caso de que se presenten y se reduce el tiempo de ejecución de las pruebas, facilitando la ejecución de estas cuando se introduce un cambio en el código [27]. Este tipo de pruebas acelera el ciclo de pruebas y desarrollo y aumenta la calidad del producto final.

La automatización también permite la ejecución de pruebas bajo diferentes entornos y configuraciones y asegura que el sistema funcione en múltiples plataformas. Esto es una gran ventaja, ya que los defectos provocados por cambios en el entorno no son tan visibles en pruebas manuales y tardarían mucho en ser identificados [28].

Aunque la automatización ofrezca varios beneficios, no es una opción eficiente en todos los casos. La selección de herramientas, la creación y mantenimiento de scripts de pruebas y la integración de estos en el desarrollo requieren una planificación y recursos especializados [26]. Por lo que es imprescindible evaluar si las pruebas automatizadas son aptas para el proyecto en cuestión y si estas aportan el máximo valor en el desarrollo de software.

Para maximizar los beneficios de la automatización de pruebas, es fundamental considerar lo siguiente:

- Viabilidad técnica y económica: la automatización es más efectiva en casos de pruebas repetitivos, como las pruebas de regresión o las de rendimiento. Para tareas puntuales o de baja frecuencia, las pruebas manuales pueden ser mejores [29].
- Selección de herramientas: para la elección de las herramientas a utilizar es preciso fijarse en aspectos como la compatibilidad con las herramientas que ya se utilizan en el proyecto [30].
- Mantenimiento: los scripts deben tener un continuo mantenimiento, lo que significa que cualquier cambio en el software también debe verse reflejado en los scripts [29].

La automatización de pruebas puede llegar a ser una opción atractiva para muchos proyecto, por su garantía de aumentar los niveles de calidad y fomentar la detección de errores en etapas tempranas del desarrollo, sin embargo, es sumamente importante planificar su implementación, ya que la automatización de pruebas no siempre es la mejor opción para todos los casos.

2.4.2. Comparación entre pruebas manuales y pruebas automatizadas

2.4.2.1. Pruebas manuales

Las pruebas manuales consisten en la evaluación del software siguiendo los casos de pruebas especificados con anterioridad y explorando el sistema para identificar errores. Las pruebas manuales son ideales para evaluar aspectos subjetivos que requieren del juicio humano, como la usabilidad, la experiencia de usuario y la estética de la interfaz [26].

Las ventajas de estas pruebas son las siguientes:

- Flexibilidad: permite adaptar los casos de pruebas en tiempo real en función de los resultados obtenidos.
- Evaluación subjetiva: analiza características no técnicas, como la facilidad de uso o la satisfacción del usuario.
- Baja inversión inicial: no requiere herramientas de automatización, lo que facilita su implementación en proyectos pequeños o con recursos limitados.

Sin embargo, las pruebas manuales también presentan desafíos, como:

- Mayor susceptibilidad a errores humano: la repetición de pruebas puede derivar a omisiones o errores.
- Tiempo y esfuerzo: son más lentas y requieren un gran esfuerzo en pruebas repetitivas.

2.4.2.2. Pruebas automatizadas

Las pruebas automatizadas son pruebas que se respaldan en herramientas y scripts para su realización de manera automática. Este tipo de pruebas son más adecuadas para casos de pruebas repetitivos o pruebas a gran escala, como las de rendimiento o seguridad [26].

Entre las ventajas de estas pruebas, se encuentran:

- Velocidad y eficiencia: permite ejecutar una gran cantidad de pruebas en poco tiempo.
- Consistencia: elimina el riesgo de errores humanos y garantiza resultados repetibles.
- Escalabilidad: facilita la ejecución de pruebas en múltiples entornos y configuraciones.

Por otra parte, también tiene limitaciones:

- Inversión inicial alta: la creación de scripts y herramientas necesarias para desarrollarlas requiere tiempo y recursos financieros.
- Mantenimiento de scripts: los scripts deben actualizarse constantemente ante los cambios realizados en el software, lo que genera costos adicionales.
- Menor adaptabilidad: no puede evaluar aspectos subjetivos como la experiencia de usuario, lo que la hace menos efectiva en áreas como la usabilidad.

2.4.3. Herramientas de automatización de pruebas

Existen muchas herramientas que ayudan a la automatización de pruebas y cada una sirve para diferentes propósitos. A continuación, se describen las herramientas más utilizadas:

2.4.3.1. Selenium

Selenium es una herramienta de código abierto utilizada para la automatización de pruebas de páginas web. Es capaz de trabajar con múltiples navegadores y sistemas operativos, y permite escribir scripts en diversos lenguajes de programación, como Java, C# y Python. Es la opción más popular para la automatización de páginas web [31].

2.4.3.2. Appium

Appium es una herramienta de automatización de código abierto diseñada para aplicaciones móviles. Permite realizar pruebas en aplicaciones nativas, hibridas y web en plataformas iOS y Android. Al igual que Selenium, soporta diversos lenguajes de programación y es capaz de adaptarse en diferentes entornos de desarrollo [31][32].

2.4.3.3. Tricentis Tosca

Tricentis Tosca es una herramienta comercial para la automatización de pruebas. Utiliza un enfoque de pruebas basado en modelos, que permite una creación y mantenimiento más eficiente de los casos de pruebas. El enfoque basado en modelos se basa en la generación de casos de prueba a partir de modelos en el sistema, como diagramas o especificaciones. Al actualizar el modelo, las pruebas se ajustan automáticamente, reduciendo el tiempo de mantenimiento [33]. Además, soporta pruebas en aplicaciones web, móviles y de escritorio y se integra con diversas herramientas de gestión de pruebas y desarrollo [34].

2.4.3.4. UFT One

UFT One, herramienta desarrollada por OpenText y conocida anteriormente como QTP, es una herramienta de automatización de pruebas funcionales que soporta gran variedad de aplicaciones, como web, móviles, SAP y Oracle. Ofrece la capacidad de reconocer objetos en la interfaz de usuario y facilita la creación y mantenimiento de los scripts de pruebas. Su diseño está enfocado a la validación funcional de las aplicaciones, ya que simula las interacciones del usuario. Esta herramienta es una de las más utilizadas para pruebas funcionales y de regresión [31].

2.4.3.5. QF-Test

QF-Test es una herramienta que se centra en la automatización de pruebas para aplicaciones Java y web. Soporta una gran cantidad de entornos y frameworks, incluyendo Swing, JavaFX y SWT, también soporta aplicaciones web basadas en HTML y JavaScript. Esta ofrece funcionalidades como grabación y reproducción de pruebas y se integra con herramientas de gestión de pruebas.

2.4.3.6. Cypress

Cypress es una herramienta de automatización de pruebas de código abierto enfocada en aplicaciones web modernas. Ofrece una arquitectura que permite funcionalidades como recarga en tiempo real y depuración. Es especialmente útil para pruebas de extremos a extremo [35].

Al ser de código abierto, Cypress permite a los desarrolladores acceder y modificar el código fuente, facilitando su adaptación con diferentes herramientas [36].

2.4.3.7. TestComplete

TestComplete es una herramienta comercial que permite la automatización de pruebas para aplicaciones de escritorio, web y móviles [37]. Soporta diversos lenguajes para la elaboración de scripts y ofrece reconocimiento de objetos, una funcionalidad que permite identificar y manipular los elementos de la interfaz de usuario. Además, se integra fácilmente con otras herramientas de desarrollo y gestión de pruebas [38].

2.4.3.8. Katalon Studio

Katalon Studio es una herramienta de automatización de pruebas que soporta aplicaciones web, móviles y API. Fue construido sobre el marco de Selenium y ofrece un entorno de desarrollo que permite la creación y ejecución de pruebas. Es una herramienta que ofrece grabación de pruebas, generación de scripts y reportes integrados. Esta soporta diversos lenguajes para la elaboración de scripts, incluyendo Groovy y Java y se integra con herramientas de gestión de pruebas como Jenkins, Jira y Git [39].

2.4.3.9. Robot Framework

Robot Framework es un framework de automatización de pruebas de código abierto que utiliza una sintaxis basada en palabras clave, facilitando la creación de casos de pruebas. Es extensible y se integra con diferentes herramientas y bibliotecas. Estas bibliotecas amplían sus capacidades, ya que permiten la integración con diferentes herramientas. Además, genera reportes en formato HTML de las pruebas ejecutadas.

2.4.3.10. Ranorex

Ranorex es una herramienta de automatización de pruebas que permite crear, ejecutar y gestionar pruebas para aplicaciones de escritorio, web y móviles. Ofrece una interfaz sencilla que permite tanto a los testers manuales como a los desarrolladores la creación de pruebas. Soporta diferentes lenguajes de scripting, como C# y VB.NET y se integra con herramientas de desarrollo y gestión de pruebas como Jenkins, Jira y TestRail.

Como mencionamos antes, la elección de las herramientas de pruebas depende exclusivamente del tipo de proyecto a probar. También hay otros factores como los recursos disponibles, pero principalmente se toman en cuenta las necesidades del proyecto. Herramientas como Ranorex o TestComplete pueden ser una solución ideal para equipos con poca experiencia en el área de la automatización, en cambio, Katalon Studio es una herramienta flexible que puede contribuir a las pruebas de múltiples plataformas y desempeñar un mejor papel en equipos con experiencia. En general, las herramientas más utilizadas en el mercado son Selenium, para aplicaciones web y Appium, para aplicaciones móviles, ya que tienen un alto nivel de flexibilidad y una gran popularidad. Por dichas razones, este proyecto se ha desarrollado utilizando Selenium y Appium como herramientas de automatización. Se profundizara en estas herramientas más adelante en el trabajo. En conjunto, esas razones hicieron a Selenium y Appium la elección para el desarrollo del framework.

2.4.4. Scripts de automatización de pruebas

Los scripts de automatización de pruebas son conjuntos de instrucciones detallas que se ejecutan para comprobar el comportamiento y funcionalidad de una aplicación. Estos scripts marcan los pasos a seguir para los test, las entradas necesarias para comprobar todos los casos y los resultados esperados.

Es común confundir los scripts con los casos de pruebas, y en el contexto de la automatización de pruebas es esencial conocer la diferencia. Un caso de prueba es un documento que define los pasos a seguir para evaluar una funcionalidad especifica del proyecto, mientras que un script de prueba es el conjunto de instrucciones que ejecuta los pasos definidos en los casos de prueba. Básicamente, es el código que hace capaz la automatización de los casos de prueba [40].

Los scripts de pruebas son los que permiten la ejecución repetitiva de las pruebas sin intervenciones manuales. Para crear dichos scripts generalmente se siguen los siguientes pasos [40]:

- Definir objetivos: establecer las funcionalidades que se evaluaran en la prueba, cuáles son las entradas necesarias y resultados esperados.
- Identificar casos de pruebas: seleccionar los escenarios específicos que se probarán, en este paso se ponen a prueba las áreas críticas de la aplicación.
- Seleccionar las herramientas adecuadas: elegir herramientas de automatización que sean compatibles con los objetivos del proyecto, con las tecnologías utilizadas en él y con sus necesidades.
- Desarrollar los scripts de pruebas: escribir los scripts utilizando el lenguaje decidido, asegurando que cada paso este correctamente definido y que los resultados esperados estén especificados.
- Validar y depurar los scripts: ejecutar los scripts en un entorno controlado para verificar su funcionamiento y realizar ajustes en según sea necesario.
- Mantener y actualizar los scripts: revisar y modificar los scripts cada vez que haya una actualización en el código o cambio en los requisitos de pruebas.

Además de estos pasos, existen reglas que se motiva a seguir al crear scripts de prueba automatizados [47]:

- Modularidad: los scripts deben ser modulares para facilitar su reutilización en diferentes escenarios.
- Claridad: es importante escribir scripts bien documentados, con comentarios que expliquen cada sección del código.
- Evitar la redundancia: los scripts deben ser concisos, evitando pasos innecesarios que aumenten la complejidad y el tiempo de ejecución.
- Pruebas independientes: cada script debe poder ejecutarse de manera independiente, sin depender de otros scripts.
- Validación robusta: incorporar verificaciones claras en los scripts para validar los resultados esperados de forma confiable.

También, existe diversas herramientas que ayudan en la creación de scripts de pruebas, tanto comerciales como de código abierto. Algunas de las más utilizadas son [41]:

- Selenium (libre): es un conjunto de herramientas para automatizar los navegadores web a través de muchas plataformas que nos permiten crear conjuntos de pruebas sobre aplicaciones web. Permite la creación de scripts en diferentes lenguajes de programación como Java, Ruby, Phyton, etc [42].
- Junit (libre): Un marco de pruebas para aplicaciones Java que facilita la creación y ejecución de pruebas unitarias [43].
- QTP – Quick Test Professional (comercial): es el módulo de automatización de la empresa HP. Permite la automatización de pruebas funcionales, soporta una amplia gama de aplicaciones y entornos y los scripts son programados en Visual Basic Scripts [44].
- Cucumber (libre): permite la automatización de pruebas de aceptación para aplicaciones web o móviles. Está basada en BDD (Behaviour Driven Development) y el lenguaje de generación de scripts es Ruby [45].
- IBM Rational Automation Framework (comercial): es el módulo para la generación de scripts y automatización de pruebas de la empresa IBM que se puede integrar con los módulos de gestión de pruebas. Permite la generación de scripts con varios lenguajes de programación [46].

La elección de esta herramientas depende de factores como el tipo de aplicación a probar, el entorno de desarrollo, el presupuesto y la experiencia del equipo de testing [40].

2.4.5. Ambientes de pruebas

Los ambientes de pruebas son entornos configurados específicamente para evaluar el comportamiento de los productos software antes de su implementación en producción. Estos entornos permiten identificar y corregir defectos, para así garantizar que el software funcione según los especificado en condiciones controladas [48].

Existen diversos ambientes para estas pruebas:

- Entornos reales: utilizan el hardware y software que se utilizará en el entorno de producción o al menos se utiliza uno muy similar. Este ambiente proporciona una representación real y precisa del comportamiento del sistema en condiciones reales, para evaluar el comportamiento de este y realizar pruebas que no se podrían hacer en otros entornos, como las de rendimiento y aceptación [49].
- Entornos virtualizados: utilizan tecnologías virtuales para simular diferentes sistemas operativos y configuraciones en un solo hardware. A diferencia de los entornos reales, estos ofrecen flexibilidad y escalabilidad en cuanto a la creación rápida de múltiples entornos de pruebas con diferentes configuraciones [49].
- Simuladores y emuladores: herramientas que replican el comportamiento de sistemas operativos o hasta dispositivos completos. Estos son de especial utilidad en casos en donde el hardware real no está disponible o es costos. Los simuladores y emuladores nos proporcionan la posibilidad de ejecutar las pruebas en diferentes dispositivos y diversas configuraciones [49].

Los ambientes de prueba se emplean en diversas fases del ciclo de vida del desarrollo de software:

- Desarrollo: en esta etapa se prueba el software que aún está siendo desarrollado. En el equipo de testing se utilizarían entornos locales o virtualizados para hacer pruebas unitarias o de integración [49].
- Integración: se combinan los componentes que se han desarrollado individualmente y se prueban en conjunto. Las pruebas de integración se intentan hacer en ambientes que simulen lo mejor posible el entorno de producción [49].
- Pruebas funcionales y no funcionales: en esta fase su utilizan entornos que replican las condiciones de producción para realizar las pruebas [50].
- Pre-producción (Staging): un entorno que refleja fielmente el entorno de producción, en esta fase se realizan las pruebas finales y las validaciones correspondientes antes del despliegue definitivo a cliente [49].

Es recomendable la utilización de ambientes de pruebas en los proyectos de desarrollo de software, pero antes de decidirse es necesario tener ciertas consideraciones para la definición de este:

- Aislamiento: asegurar que los distintos entornos de pruebas establecidos estén separados del entorno de producción para evitar interferencias y asegura la fiabilidad e integridad de los datos finales [49].
- Representatividad: es necesario configurar los entornos de pruebas para que sean lo más similares posible al entorno d producción, incluyendo las configuraciones de hardware, software, bases de datos y redes [50].
- Gestión de datos de pruebas: utilizar datos seleccionados de escenarios reales sin comprometer la privacidad o seguridad de los datos relacionados [50].
- Automatización y control de versiones: implementar herramientas que faciliten la configuración y mantenimiento de los entornos de pruebas [49].

2.4.6. Desarrollo Guiado por Comportamiento (BDD)

El Desarrollo Guiado por Comportamiento (Behavior-Driven Development, BDD) es una metodología de desarrollo de software que se enfoca en la colaboración entre los diferentes roles involucrados en un proyecto de desarrollo software. BDD se encarga de utilizar un lenguaje común que habilite el entendimiento y la práctica entre todas las partes del equipo [51].

Características de BDD

- Lenguaje común: fomenta la utilización de un lenguaje natural, como el formato Given-When-Then, para describir el comportamiento esperado de un sistema. Este lenguaje es entendible para las personas técnicas y no técnicas.
- Colaboración: aumenta la colaboración entre los equipos técnicos y los stakeholders para definir los criterios de aceptación y para definir las funcionalidades del software desde el punto de vista del usuario.
- Pruebas como documentación: ya que las pruebas están escritas en lenguaje natural, estas actúan como documentación, ya que muestran en todo momento que hace el software y por qué.
- Trabaja en iteraciones rápidas y pequeñas: se aumenta la cantidad de entregas para tener una visión del estado del producto software clara, también con esto se aumenta la retroalimentación y flujo de valor para el cliente.

Este modelo de desarrollo proporciona principalmente claridad, ya que proporciona un lenguaje común en el equipo, que fomenta el entendimiento entre todos los miembros y se asegura que todos los integrantes comprendan los requisitos, aumentando la comunicación efectiva [51].

También, este contribuye a mejorar la calidad del software, ya que desde un principio, desarrolladores y empleados no funcionales, conocen el funcionamiento y el comportamiento esperado de las pruebas. Esto facilita la reducción de defectos, ya que se aumenta el número de participantes en la creación y revisión de las pruebas [51].

Además del Desarrollo Guiado por Comportamiento (BDD) existen también otras metodologías, como Desarrollo Guiado por Pruebas (TDD) y Desarrollo Guiado por Pruebas de Aceptación (ATDD). Aunque todas tienen el mismo objetivo de garantizar que el software cumple con los requisitos establecidos, poseen diferentes enfoques y etapas.

El Desarrollo Guiado por Pruebas (Test-Driven Development, TDD), se enfoca en el desarrollo de pruebas unitarias, antes de la creación del código. Estas pruebas suelen ser específicas, ya que se utilizan para guiar a los desarrolladores en la escritura del código [80].

Por otro lado, el Desarrollo Guiado por Pruebas de Aceptación (Acceptance Test-Driven Development, ATDD), como su nombre indica, se enfoca más en las pruebas de aceptación [80].

Muchos piensan que la metodología de BDD reemplaza los procesos ágiles existentes, pero no es el caso, este lo complementa. BDD actúa como un valor añadido, que permite a los equipos cumplir con los objetivos Agile, como entregas confiables de software funcional que satisfagan las necesidades del cliente [81].

Las actividades de los equipos que implementan esta metodología se basan en [81]:

- Descubrimiento: analizar ejemplos de nuevas funcionalidades (historias de usuario) para detallar y acordar los detalles que se esperan lograr.
- Formulación: documentar dichos ejemplos de forma que puedan ser automatizados.
- Automatización: implementar el comportamiento escrito en el ejemplo documentado.

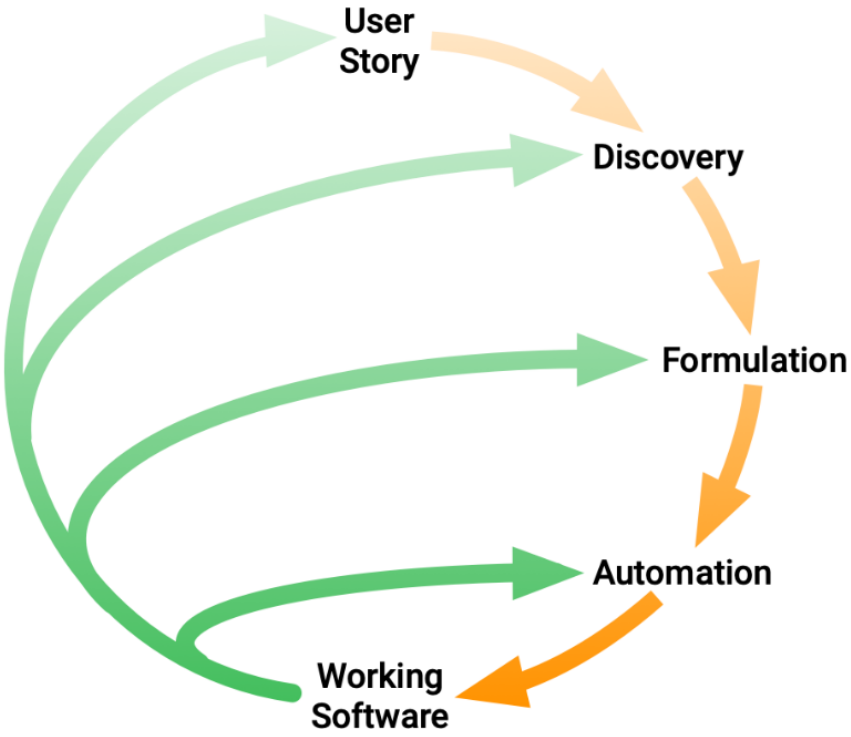


Figura 9 – 2.8. Descubrimiento, Formulación y Automatización

Entrando más a fondo, en lo que cada tarea se basa [81]:

2.4.6.1. Descubrimiento: qué podría ser

La parte más compleja de iniciar un proyecto software es el inicio, determinar que crear y como crearlo. El objetivo principal es desarrollar software valioso y funcional, por lo que en esta etapa se inician conversaciones entre las personas involucradas en el desarrollo y entrega de las pruebas. En estas conversaciones, se analizan ejemplos reales del sistema desde la perspectiva del usuario y se especifican las necesidades y reglas que rigen el funcionamiento del sistema. Estas conversaciones ayudan a todos los integrantes del equipo a tener una visión más amplia y completa de las necesidades del usuario y las acciones a tomar para satisfacerlas.

2.4.6.2. Formulación: qué debería hacer

La formulación implica documentar los ejemplos antes mencionados a un formato que pueda ser automatizado. Esta documentación sirve como especificación y documentación del sistema. Los ejemplos se toman individualmente y se formulan de tal manera que se tenga múltiples ejemplos documentados. A diferencia de la documentación tradicional, BDD proporciona un medio que puede ser leído por personal funcional y no funcional, por lo que:

- Se puede obtener retroalimentación de ambas partes del equipo.
- Se puede automatizar estos ejemplos para guiar el desarrollo.

2.4.6.3. Automatización: qué hace

La automatización convierte las especificaciones en pruebas automatizadas que verifican que el sistema se comporte según lo esperado. Estas pruebas proporcionan retroalimentación inmediata y aseguran que el código cumple con los criterios de aceptación.

Una de las herramientas más populares de la metodología de Desarrollo Guiado por Comportamiento es Cucumber, este proporciona un marco para escribir

especificaciones en lenguaje natural, y las vincula con el código del sistema. Por todas las razones anteriormente mencionadas, se ha decidido utilizar esta metodología en el proyecto de desarrollo.

2.5. Selenium

Selenium es una suite de herramientas para la automatización de pruebas de aplicaciones, está la integran varios componentes que están diseñados para colaborar en los diferentes aspectos de las pruebas [51].

El componente WebDriver es la base de Selenium y se utiliza principalmente para la automatización de pruebas de aplicaciones web de escritorio y de móviles. WebDriver interactúa con los navegadores a través de las APIs de automatización proporcionadas por los propios navegadores. Esto asegura que las pruebas simulen de manera realista las interacciones de un usuario, realizando las acciones como si un humano estuviera utilizando el navegador [51].

El selenium IDE es una herramienta diseñada para el desarrollo rápido de casos de pruebas. Se presenta como una extensión para navegadores como Chrome y Firefox, que permite grabar sus acciones en el navegador y convertirlas automáticamente en scripts de prueba. Estos scripts utilizan comandos de Selenium ajustados al contexto de los elementos interactuados.

El IDE no solo acelera el proceso de desarrollo de pruebas, sino que también es una excelente herramienta para aprender la sintaxis de los scripts de Selenium, lo que lo convierte en una alternativa perfecta para usuarios principiantes [51].

El componente Selenium Grid permite la ejecución de pruebas en múltiples máquinas y plataformas de forma distribuida. Grid facilita la ejecución simultánea de pruebas en diferentes combinaciones de navegadores y sistemas operativos, de tal manera que se optimice el tiempo de ejecución y mejorando la cobertura de las pruebas.

El control de la ejecución se realiza desde un nodo local, mientras que las pruebas son ejecutadas automáticamente por nodos remotos. Este enfoque es útil en proyectos que requieren verificar la compatibilidad de la aplicación con diferentes entornos, navegadores o sistemas operativos [51].

2.5.1. Componentes

Para desarrollar una test suite utilizando WebDriver hay que entender el uso de los diferentes componentes.

El componente WebDriver se comunica con el navegador a través de un driver. Esta comunicación es mutua: el WebDriver se comunica con el browser a través del driver y también recibe información a través de la misma vía (como se muestra en la figura 5.1.).

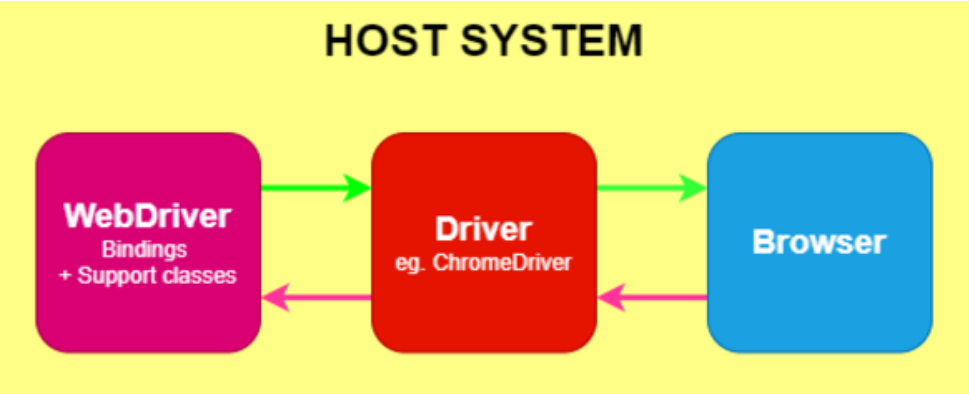


Figura 10 – 2.9. Comunicación entre WebDriver y navegador [52]

Se utiliza un driver especifico por cada navegador, por ejemplo: se utiliza ChromeDriver para el navegador Chrome o Chromium, GeckoDriver para Mozilla´s Firefox, etc. El driver se ejecuta en el mismo sistema que el browser y este puede o no puede ser en el mismo sistema en el que se ejecutan los tests.

El ejemplo utilizado anteriormente hace referencia a la comunicación directa entre el driver y el navegador, pero la comunicación entre estos también puede ser remota a través de Selenium Server o RemoteWebDriver. RemoteWebDriver, que se observa en la figura 5.2., se ejecuta en el mismo sistema en el que se ejecuta el driver y el navegador.

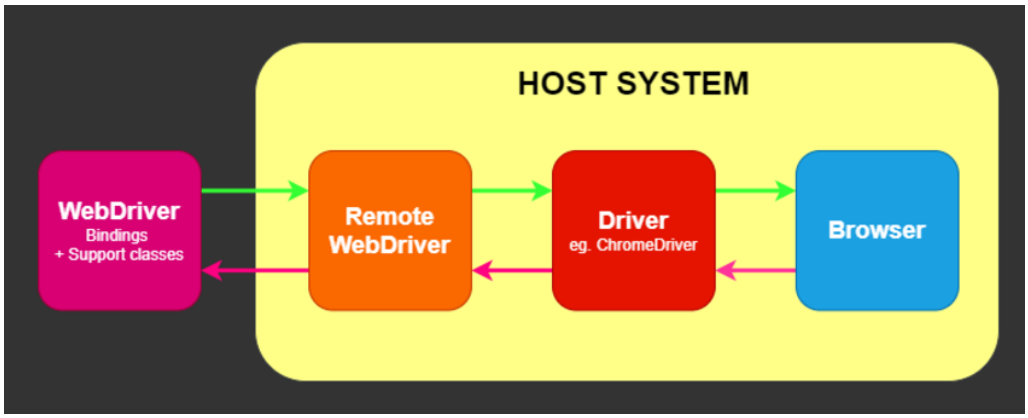


Figura 11 – 2.10. Comunicación remota entre WebDriver y navegador [52]

La comunicación remota también puede ser utilizada con Selenium Server o Selenium Grid.

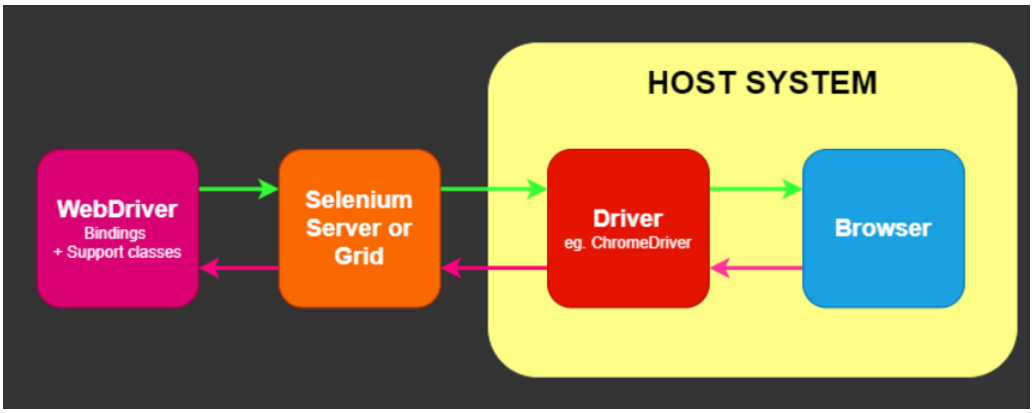


Figura 12 – 2.11. Comunicación remota entre WebDriver y navegador utilizando Selenium Server o Grid [52]

El único trabajo de WebDriver es la comunicación con el navegador a través de las vías mencionadas anteriormente, por lo que es necesario un framework de automatización de pruebas a la hora de realizar los tests. Se necesitará un framework que comparta el lenguaje de las pruebas, como: NUnit para .NET, Junit para Java, RSpec para Ruby, etc.

El framework, como se ve en la figura 5.4., es el responsable de ejecutar el WebDriver y las pruebas correspondientes.

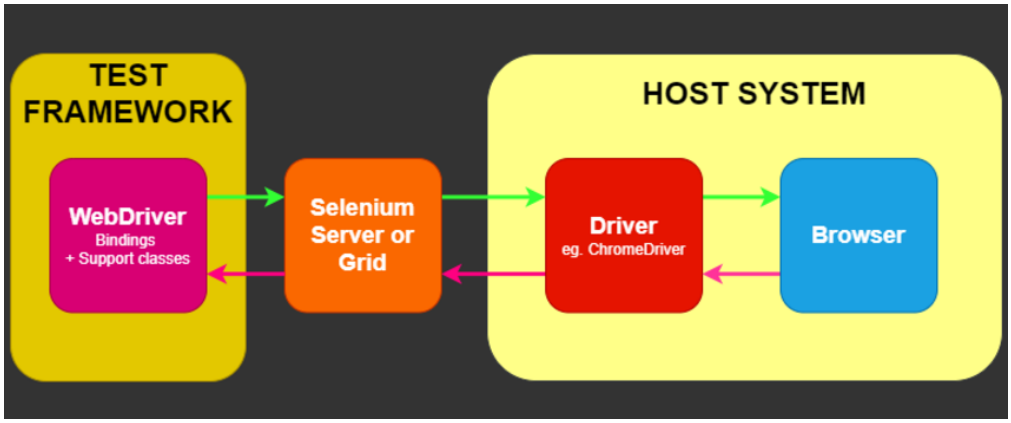


Figura 13 – 2.12. Comunicación entre Framework y navegador [52]

También se pueden utilizar herramientas de lenguaje natural como Cucumber en Frameworks, como en el caso del proyecto desarrollado, pero todo depende de cómo se quiera enfocar este.

2.5.2. WebDriver

Selenium WebDriver es uno de los componentes principales de Selenium, diseñado para controlar navegadores, ya sea en una máquina local o de forma remota. Selenium WebDriver se refiere tanto a las bibliotecas que permiten la automatización como a las implementaciones específicas que controlan cada navegador, y es comúnmente conocido como “WebDriver” [53].

Su enfoque orientado a objetos permite interactuar con el navegador, lo que facilita la creación de pruebas y su escalabilidad. En este proyecto, de todos los componentes de Selenium, se ha seleccionado WebDriver como la herramienta principal para la automatización de pruebas web [53].

Selenium ofrece una serie de recursos diseñados para facilitar el aprendizaje y la adopción de su marco de automatización, especialmente cuando se trata de nuevos usuarios. Este marco soporta la automatización de los principales navegadores disponibles en el mercado a través del uso de WebDriver. Cada navegador está respaldado por una implementación especifica de WebDriver, denominada "driver". Como se vió antes, este driver actúa como un intermediario entre Selenium y el navegador, gestionando la comunicación y delegando las acciones necesarias para ejecutar las pruebas automatizadas.

Una características distintiva de Selenium es la separación entre su API y los controladores de los navegadores, un diseño intensional que asigna a los proveedores de navegadores la responsabilidad de implementar sus propios drivers. Aunque Selenium utiliza estos drivers proporcionados por terceros siempre que es posible, el proyecto también desarrolla y mantiene sus propios controladores para los casos en los que no se dispone de drivers ofrecidos por los fabricantes de navegadores [54].

El marco de Selenium integra todos estos elementos mediante una interfaz orientada al usuario, que permite el uso transparente de distintos backends de navegadores. Esta integración facilita la automatización multiplataforma y en múltiples navegadores.

A diferencia de otras herramientas comerciales de automatización, la configuración de Selenium requiere la instalación de bibliotecas específicas, conocidas como bindings, para el lenguaje de programación elegido, así como el navegador y el driver correspondiente [54].

2.5.2.1. Drivers

En Selenium WebDriver, los drivers son componentes que facilitan la comunicación entre el código de pruebas y los navegadores web. Cada navegador cuenta con un driver específico que traduce las instrucciones del WebDriver en acciones que el navegador puede ejecutar, permitiendo así la automatización de tareas como la navegación y la interacción con elementos de la página.

Creación de sesiones

Para iniciar una sesión con un navegador, es necesario crear una instancia de la clase correspondiente al driver del navegador deseado. Este proceso establece una nueva sesión de WebDriver, que se asocia con una instancia del navegador en la que se ejecutarán las pruebas automatizadas. La creación de una nueva sesión se alinea con el comando "New Session" definido en la especificación W3C para WebDriver [55]. En lenguajes de programación como Java, esto se logra inicializando un nuevo objeto de la clase driver, por ejemplo:

```
WebDriver driver = new ChromeDriver();
```

Finalización de Sesiones

Una vez completadas las pruebas, es fundamentan cerrar la sesión del driver para liberar los recursos asociados. Esto se realiza mediante el método quit(), que cierra todas las ventanas del navegador abiertas durante la sesión y finaliza la instancia del WebDriver [55]. Por ejemplo:

```
driver.quit();
```

Es importante destacar que, al trabajar con sesiones de WebDriver, se deben manejar adecuadamente las excepciones y asegurar que las sesiones se cierren correctamente, incluso en fallos durante la ejecución de las pruebas [55].

2.5.2.2. Opciones del navegador

Las opciones del navegador son configuraciones necesarias a la hora de determinar el comportamiento de las sesiones de pruebas automatizadas. Estas opciones permiten especificar las diferentes capacidades del navegador, y especificando las pruebas según las necesidades del proyecto.

Capacidades comunes a todos los navegadores

A partir de Selenium 4, las capacidades del navegador se definen utilizando clases de opciones específicas para cada navegador, en lugar de las clases de "Desired Capabilities" utilizadas en versiones anteriores. Esta transición mejoro la claridad en la configuración de las sesiones de WebDriver.

Las opciones compartidas por todos los navegadores son las siguientes [56]:

- browserName: especifica el nombre del navegador que se utilizará en la sesión de prueba. Al instanciar una clase de opciones para un navegador particular, este valor se establece automáticamente.

```
ChromeOptions chromeOptions = new ChromeOptions();
String name = chromeOptions.getBrowserName();
```

- pageLoadStrategy: determina la estrategia que el WebDriver seguirá al cargar las páginas. Las opciones disponibles son:
    - normal (por defecto): espera a que todos los recursos de la página se hayan descargado completamente.

```
ChromeOptions chromeOptions = new ChromeOptions();
chromeOptions.setPageLoadStrategy(PageLoadStrategy.NORMAL);
WebDriver driver = new ChromeDriver(chromeOptions);
```

    - Eager: considera que la página está lista cuando el DOM está accesible, aunque otros recursos, como imágenes, aún estén cargándose.

```
ChromeOptions chromeOptions = new ChromeOptions();
chromeOptions.setPageLoadStrategy(PageLoadStrategy.EAGER);
WebDriver driver = new ChromeDriver(chromeOptions);
```

    - None: no espera a que se complete la carga de la página, permitiendo que las pruebas continúen sin bloqueos.

```
ChromeOptions chromeOptions = new ChromeOptions();
chromeOptions.setPageLoadStrategy(PageLoadStrategy.NONE);
WebDriver driver = new ChromeDriver(chromeOptions);
```

- Timeouts: permite establecer tiempos de espera predeterminados para diversas operaciones, como la carga de scripts, la búsqueda de elementos y la navegación entre páginas.
    - Script: tiempo de espera para la ejecución de scripts.

```
ChromeOptions chromeOptions = new ChromeOptions();
Duration duration = Duration.of(5, ChronoUnit.SECONDS);
chromeOptions.setScriptTimeout(duration);
```

    - pageLoad: tiempo de espera para la carga de la página.

```
ChromeOptions chromeOptions = new ChromeOptions();
Duration duration = Duration.of(5, ChronoUnit.SECONDS);
chromeOptions.setPageLoadTimeout(duration);
```

    - implicit: tiempo máximo para la búsqueda de elementos antes de lanzar un excepción.

```
ChromeOptions chromeOptions = new ChromeOptions();
Duration duration = Duration.of(5, ChronoUnit.SECONDS);
chromeOptions.setImplicitWaitTimeout(duration);
```

Opciones específicas para cada navegador

Además de las opciones comunes, cada navegador ofrece opciones específicas. Por ejemplo, en Chrome, es posible añadir argumentos de líneas de comandos para modificar su comportamiento, como indicar el navegador en modo incognito o deshabilitar extensiones. De manera similar, Firefox permite especificar perfiles de usuario personalizados o definir la ubicación del ejecutable del navegador [56].

Para configurar estas opciones en una sesión de WebDriver, se instancian las clases de opciones correspondientes y se establecen las propiedades deseadas. Posteriormente, estas opciones se pasan al crear la instancia del driver. Por ejemplo:

```
ChromeOptions options = new ChromeOptions();
options.addArguments("--incognito");
options.setAcceptInsecureCerts(true);
WebDriver driver = new ChromeDriver(options);
```

2.5.2.3. Localizadores

La identificación de los elementos dentro del Document Object Model (DOM) es necesaria a la hora de interactuar con la página web. Para este propósito, Selenium ofrece diversas estrategias de localización que permiten seleccionar elementos específicos en función de sus atributos y relaciones en el DOM [58].

Estrategias de Localización tradicionales

Selenium proporciona soporte para las siguientes estrategias de localización en WebDriver [58]:

- Class name: localiza elementos cuyo atributo de clase contiene el valor de búsqueda especificado. Es importante destacar que no se permiten nombres de clase compuestos en esta estrategia.

```
WebDriver driver = new ChromeDriver();
driver.findElement(By.className("information"));
```

- Css selector: permite la localización de elementos que coinciden con un selector CSS determinado, y así seleccionar elementos basándose en sus estilos y jerarquías.

```
WebDriver driver = new ChromeDriver();
driver.findElement(By.cssSelector("#fname"));
```

- Id: identifica elementos cuyos atributo ID coincide exactamente con el valor de búsqueda proporcionado. Como los IDs deben ser únicos dentro de una página HTML, esta estrategia es altamente confiable.

```
WebDriver driver = new ChromeDriver();
driver.findElement(By.id("lname"));
```

- Name: selecciona atributos cuyo NAME coincide con el valor de búsqueda, siendo útil en formularios y controles de entrada donde este atributo es comúnmente utilizado.

```
WebDriver driver = new ChromeDriver();
driver.findElement(By.name("newsletter"));
```

- Link text: orienta elementos de anclaje (<a>) cuyo texto visible coincide exactamente con el valor de búsqueda, facilitando la interacción con enlaces específicos en la página.

```
WebDriver driver = new ChromeDriver();
driver.findElement(By.linkText("Selenium Official Page"));
```

- Partial link text: similar al anterior, pero permite la localización de enlaces cuyo texto visible contiene el valor de búsqueda, sin necesidad de una coincidencia exacta, lo que es útil para enlaces con textos dinámicos o variables.

```
WebDriver driver = new ChromeDriver();
driver.findElement(By.partialLinkText("Official Page"));
```

- Tag name: localiza elementos que coinciden con un nombre de etiqueta HTML específico, permitiendo la selección de todos los elementos de un tipo particular, como , <input>, etc.

```
WebDriver driver = new ChromeDriver();
driver.findElement(By.tagName("a"));
```

- Xpath: utiliza expresiones XPath para localizar elementos basándose en su estructura y relaciones dentro del DOM, ofreciendo una gran flexibilidad para seleccionar elementos complejos o basados en condiciones específicas.

```
WebDriver driver = new ChromeDriver();
driver.findElement(By.xpath("//input[@value='f']"));
```

Utilizando Localizadores

Selenium también introduce otros mecanismos para la búsqueda de elementos [58]:

- By(): Es una clase base que contiene métodos estáticos para todas las estrategias anteriormente mencionadas, como By.id, By.name, By.xpath, entre otros.

```
import org.openqa.selenium.By;
WebDriver driver = new ChromeDriver();
driver.findElement(By.className("information"));
```

- ByAll(): permite combinar múltiples estrategias de localización. Un elemento se considera encontrados si cumple al menos una de las estrategias especificadas.

```
By example = new ByAll(By.id("password-field"), By.id("username-
field"));
List<WebElement> login_inputs = driver.findElements(example);
```

2.5.2.4. Interacciones

En WebDriver solo hay 5 comandos básicos que se pueden ejecutar en un elemento [60]:

- Click (aplica a cualquier elemento)
- sendKeys (solo aplica a campos de texto y elementos de contenido editable)
- clear (solo aplica a campos de texto y elementos de contenido editable)
- submit (solo aplica a elementos de forma)
- select (se mencionaran posteriormente)

Validaciones adicionales

1. si se determina que el elemento a interactuar esta fuera de la ventana, se desplazara hasta que el elemento este visible.
2. Se asegura de que el elemento sea interactuable antes de realizar cualquier acción.

Click

El método click() es ejecutado en el centro del elemento. Si dicha área esta obstruida, Selenium generara un error de "element click intercepted" [60].

```
driver.get("https://www.selenium.dev/selenium/web/inputs.html");
        // Click on the element
        WebElement
checkInput=driver.findElement(By.name("checkbox_input"));
        checkInput.click();
```

SendKeys

El método sendKeys() permite ingresar texto en un elemento editable, como un campo de entrada de formulario o un elemento con el atributo content editable.

```
// Clear field to empty it from any previous data
WebElement emailInput=driver.findElement(By.name("email_input"));
emailInput.clear();
//Enter Text
String email="admin@localhost.dev";
emailInput.sendKeys(email);
```

Si el elemento no es editable, se lanzara un error de estado de elemento invalido [60].

Clear

El método clear() restablece el contenido de un elemento editable, dejándolo vacío. Si el elemento no es editable o no puede ser restablecido, se producirá un error de estado de elemento inválido [60].

```
//Clear Element
        // Clear field to empty it from any previous data
        emailInput.clear();
```

Submit

En Selenium 4, el método submit() ya no se implementa como un punto de acceso separado; en su lugar, se ejecuta mediante un script. Se recomienda no utilizar este método y, en su lugar, hacer click en el botón de envío del formulario correspondiente [60].

2.5.2.5. Información sobre los elementos

Hay una gran cantidad de detalles que se puede extraer de elementos específicos [61].

Visibilidad del Elemento

El método isDisplay() es un booleano que verifica si un elemento es visible en la interfaz de usuario. Un elemento se considera visible si ocupa espacio en el diseño de página y no está oculto mediante estilos CSS.

```
driver.get("https://www.selenium.dev/selenium/web/inputs.html");
// isDisplayed
 // Get boolean value for is element display
boolean isEmailVisible =
driver.findElement(By.name("email_input")).isDisplayed();
assertEquals(isEmailVisible,true);
```

Estado de habilitación

El método isEnabled() se utiliza para comprobar si un elemento es interactivo o no, es decir, si está habilitado para recibir instrucciones del usuario.

```
        //isEnabled
       //returns true if element is enabled else returns false
        boolean isEnabledButton =
driver.findElement(By.name("button_input")).isEnabled();
        assertEquals(isEnabledButton,true);
```

Estado de Selección

El método isSelected() verifica si un elemento está seleccionado o no. Este método es ampliamente utilizado para comprobar si las casillas de verificación, botones o elementos interactuables están seleccionado.

```
//isSelected
        //returns true if element is checked else returns false
        boolean isSelectedCheck =
driver.findElement(By.name("checkbox_input")).isSelected();
        assertEquals(isSelectedCheck,true);
```

2.5.2.6. Interacciones del navegador

A continuación se describen las principales interacciones que se pueden realizar [62]:

Información del navegador

- getCurrentUrl(): devuelve la URL de la página que se está visualizando.

```
String url = driver.getCurrentUrl();
```

Navegación en el navegador

- Get(): carga la página especificada.

```
//Convenient
driver.get("https://selenium.dev");

//Longer way
driver.navigate().to("https://selenium.dev");
```

- Navigate().back(): simula la acción de presionar el botón de retroceso del navegador.

```
//Back
driver.navigate().back();
```

- Navigate().forward(): emula la acción de avanzar en el historial del navegador.

```
//Forward
driver.navigate().forward();
```

- Navigate().refresh(): recarga la página actual.

```
//Refresh
        driver.navigate().refresh();
```

Impresión de pantallas

Selenium facilita la captura de pantallas mediante sus clases PrintOptions, PrintsPage y BrowsingContext, estas proveen una interfaz para la automatización de estas acciones. También, estas habilitan la posibilidad de configurar las preferencias de impresión, como márgenes, escalas, entre otros [65].

Una vez establecidos las opciones deseadas, podemos llamar a la función print, que generará un PDF de la página web.

```
public void PrintWithPrintsPageTest()
{
    driver.get("https://www.selenium.dev/");
    PrintsPage printer = (PrintsPage) driver;
    PrintOptions printOptions = new PrintOptions();
    Pdf printedPage = printer.print(printOptions);
    Assertions.assertNotNull(printedPage);
}
```

Manejo de ventanas

Aunque WebDriver no distingue entre ventanas y pestañas, Seleniun puede interactuar con ellas a través de un manejador de ventanas. Este contiene los siguientes métodos [66]:

- getWindowHandle(): cada ventana tiene un identificador único por cada sesión del navegador, este método devuelve el identificador de la ventana o pestaña de dicha sesión.

```
// Navigate to Url
driver.get("https://www.selenium.dev/selenium/web/window_switching_tes
ts/page_with_frame.html");
//fetch handle of this
String currHandle=driver.getWindowHandle();
assertNotNull(currHandle);
```

- switchTo(): el sistema operativo solo considera activa una pestaña por sesión, si se desea cambiar esta, es necesaria la utilización del método switchTo(). Este método devuelve todos los controladores de ventanas y los almacena en un array. Para encontrar el deseado se deberá iterar el array hasta encontrarlo.

```
//click on link to open a new window
driver.findElement(By.linkText("Open new window")).click();
//fetch handles of all windows, there will be two, [0]- default, [1] -
new window
Object[] windowHandles=driver.getWindowHandles().toArray();
driver.switchTo().window((String) windowHandles[1]);
//assert on title of new window
String title=driver.getTitle();
assertEquals("Simple Page",title);
```

- Close(): el método cierra la ventana actual. Es recomendable cerrar las ventanas que no se van a utilizar y cambiar con el manejador a la ventana activa.

```
//closing current window
driver.close();
//Switch back to the old tab or window
driver.switchTo().window((String) windowHandles[0]);
```

- newWindow(): el método crea una nueva ventana o pestaña.

```
//Opens a new tab and switches to new tab
driver.switchTo().newWindow(WindowType.TAB);
assertEquals("",driver.getTitle());

//Opens a new window and switches to new window
driver.switchTo().newWindow(WindowType.WINDOW);
assertEquals("",driver.getTitle());
```

- Quit(): cierra las ventanas y la sesión del navegador.

```
//quitting driver
driver.quit(); //close all windows
```

- TakeScreenshot(): toma una captura de pantalla de la ventana activa.

```
import org.apache.commons.io.FileUtils;
import org.openqa.selenium.chrome.ChromeDriver;
import java.io.*;
import org.openqa.selenium.*;

public class SeleniumTakeScreenshot {
    public static void main(String args[]) throws IOException {
        WebDriver driver = new ChromeDriver();
        driver.get("http://www.example.com");
        File scrFile =
((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);
        FileUtils.copyFile(scrFile, new File("./image.png"));
        driver.quit();
    }
}
```

- executeScript(): ejecuta un código JavaScript en la ventana del contexto actual.

```
 //Creating the JavascriptExecutor interface object by Type casting
 JavascriptExecutor js = (JavascriptExecutor)driver;
 //Button Element
   WebElement button =driver.findElement(By.name("btnLogin"));
 //Executing JavaScript to click on element
   js.executeScript("arguments[0].click();", button);
 //Get return value from script
   String text = (String) js.executeScript("return
arguments[0].innerText", button);
 //Executing JavaScript directly
   js.executeScript("console.log('hello world')");
```

2.5.2.7. Acciones

Además de las interacción entre elementos, WebDriver también proporciona un control sobre las acciones que pueden realizar los dispositivos de entrada y salida de datos. La interfaz de Selenium tiene tres tipos de entradas: teclado, ratón y rueda y cada una tiene la posibilidad de construir comandos individuales específicos [67].

Teclado

Solo existen dos acciones que se pueden realizar con el teclado; presionar una tecla y liberar una tecla ya presionada [68].

- sendKeys(): este método es la combinación de presionar y liberar una tecla en una sola acción. Simula el envío de un texto por teclado.

```
new Actions(driver)
                .sendKeys("abc")
                .perform();
```

Ratón

Para el ratón, solo hay tres acciones que se pueden realizar: pulsar un botón, liberar un botón pulsado y mover el ratón. Selenium proporciona métodos que combinan estas acciones para facilitar la realización de pruebas [69].

Existen en total cinco botones definidos para un ratón: botón izquierdo (0), botón del medio (1), botón derecho (3), botón de atrás (4), botón de adelante (5) [69].

- clickAndHold(): este método combina el movimiento del ratón a un punto especifico con el presionar el botón izquierdo de este.

```
WebElement clickable = driver.findElement(By.id("clickable"));
        new Actions(driver)
                .clickAndHold(clickable)
                .perform();
```

- clickAndRelease(): este método es igual al anterior, pero este presiona y libera el botón izquierdo del ratón.

```
WebElement clickable = driver.findElement(By.id("click"));
        new Actions(driver)
                .click(clickable)
                .perform();
```

- doubleClick(): pulsa y libera el botón izquierdo del mouse dos veces.

```
WebElement clickable = driver.findElement(By.id("clickable"));
        new Actions(driver)
                .doubleClick(clickable)
                .perform();
```

- moveToElement(): mueve el ratón a un elemento determinado.

```
WebElement hoverable = driver.findElement(By.id("hover"));
        new Actions(driver)
                .moveToElement(hoverable)
                .perform();
```

- dragAndDrop(): primero se presiona sobre un elemento, se mueve a un punto determinado y se libera el elemento.

```
WebElement draggable = driver.findElement(By.id("draggable"));
        WebElement droppable = driver.findElement(By.id("droppable"));
        new Actions(driver)
                .dragAndDrop(draggable, droppable)
                .perform();
```

Rueda

Hay 3 escenarios cuando nos desplazamos en una página web [70]:

- scrollToElement(): este método es el más común y solo puede ser utilizado cuando el elemento no está visible. Este se desplaza por la página hasta encontrar el elemento especificado.

```
WebElement iframe = driver.findElement(By.tagName("iframe"));
        new Actions(driver)
                .scrollToElement(iframe)
                .perform();
```

- scrollByAmount(): hace que la pantalla se desplace hasta cierto punto.

```
WebElement footer = driver.findElement(By.tagName("footer"));
        int deltaY = footer.getRect().y;
        new Actions(driver)
                .scrollByAmount(0, deltaY)
                .perform();
```

- Desplazarse desde un elemento por una cantidad especifica: es la combinación de los dos métodos anteriores. Primero utiliza un elemento determinado como punto de origen y a partir de ese punto se desplaza a las coordinadas previamente establecidas.

```
WebElement iframe = driver.findElement(By.tagName("iframe"));
WheelInput.ScrollOrigin scrollOrigin =
WheelInput.ScrollOrigin.fromElement(iframe);
new Actions(driver)
        .scrollFromOrigin(scrollOrigin, 0, 200)
        .perform();
```

2.6. Appium

2.6.1. Introducción

Appium es una herramienta de automatización de código abierto diseñada para facilitar la automatización de interfaces en diversas plataformas, como aplicaciones móviles (iOS, Android, Tizen), navegadores web (Chrome, Firefox, Safari), aplicaciones de escritorio (macOS, Windows), entre otras. También, soporta diferentes lenguajes a la hora de implementar el código de automatización [71].

Para lograr estas funcionalidades, Appium se estrutura en cuatro componentes [71]:

- Appium Core: define las APIs centrales que proporcionan las funcionalidades básica de automatización.
- Drivers: implementan la conectividad con plataformas específicas, permitiendo la interacción directa con diferentes entornos.
- Clients: proporcionan implementaciones de la API de Appium en diversos lenguajes de programación, como JavaScript, Java y Python.
- Plugins: permiten modificar o extender la funcionalidad central de Appium, adaptándolo a las integraciones específicas.

Como mencionamos antes, Appium es un proyecto de código abierto diseñado para facilitar la automatización de interfaces de usuario en múltiples plataformas. Con el lanzamiento de Appium 2, se han establecido los siguientes objetivos [72]:

- Proporcionar capacidades de automatización específicas de cada plataforma a través de una API estándar y multiplataforma.
- Permite el acceso a la API desde cualquier lenguaje de programación.
- Ofrece herramientas que permiten el desarrollo comunitario de Appium.

2.6.2. API de Appium

Appium se beneficia enormemente de los avances realizados previamente por Selenium. Como se comentó antes, gracias a su colaboración con los proveedores de navegadores y el grupo de estándares de W3C, Selenium logró convertir su API del proyecto WebDriver, en un estándar oficial denominado la especificación WebDriver. En la actualidad, todos los navegadores principales tienen implementados capacidades de automatización alineadas con estas especificaciones, eliminando la necesidad de agregar software adicional especifico a cada navegador.

Inicialmente, Appium se propuso crear un estándar de automatización para aplicaciones móviles, tanto en iOS como en Android, pero en lugar de eso, decidió adoptar las especificaciones de WebDriver como su API, con el fin de promover la estandarización. La interacción del usuario en sitios web y aplicaciones móviles difieren en algunos aspectos, por ello, los conceptos básicos de la especificación WebDriver (como encontrar elementos, interactuar con ellos y cargar páginas o pantallas) son aplicables a múltiples plataformas.

Appium reconoce las diferencias entre plataformas, y para abordarlas, aprovecha la extensibilidad de WebDriver y agrega algunas consideraciones especificas:

- Compatibilidad limitada: en ciertas plataformas, algunos comandos de WebDriver pueden ser no compatibles. Por ejemplo, la automatización de

aplicaciones móviles nativas no permite comandos relacionados con cookies.
- Extensión personalizadas: Appium amplía la funcionalidad más allá de los comandos disponibles en la especificación WebDriver, asegurando que estas cumplan con los estándares.

2.6.3. Plataforma de automatización

Appium es básicamente un programa de Node.js y utiliza un protocolo para mapear comportamientos de automatización en diversas plataformas. Este trabajo no lo hace Appium directamente, lo realiza un módulo conocido como driver de Appium, más adelante daremos una instrucción más detallada.

Un driver es un módulo que le otorga a Appium la capacidad de automatizar una plataforma específica (o un conjunto de plataformas). Generalmente, un driver realiza la automatización apoyándose en tecnologías propias de la plataforma utilizada. Por ejemplo, Apple ofrece una tecnología de automatización para iOS llamada XCUITest, este obtiene ese nombre, ya que básicamente convierte el protocolo WebDriver en llamadas a la biblioteca XCUITest.

Una de las razones por las que los drivers son módulos independientes es que cada uno funciona de manera completamente diferente. Las herramientas y los requisitos para crear y usar drivers varía según la plataforma. Por esto, Appium permite utilizar únicamente drivers específicos dependiendo de la tarea a realizar. La elección e instalación de estos drivers es tan importante que Appium cuenta con su propia interfaz de línea de comandos (CLI) para gestionarlos.

2.6.4. Acceso a lenguajes de programación universal

Como Appium es un programa basado en Node.js, podría haber sido diseñado para integrarse como una biblioteca, junto con sus driver, dentro de los programas propios de Node.js, pero como eso no cumplía con el objetivo de Appium de ofrecer capacidades de automatización a personas que utilizan cualquier lenguaje de programación popular, se decidió adoptar el enfoque de Selenium [72].

El haber adoptado el enfoque de Selenium desde el principio hizo que ya no existiera este problema. El protocolo de especificación de WebDriver es un protocolo basado en HTTP, lo que significa que está diseñado para ser utilizado a través de una red, en lugar de ejecutarse únicamente dentro de la memoria de un programa.

Esta arquitectura “cliente-servidor” permite separar al implementador de la automatización (el “servidor, encargado de ejecutar la automatización”), del ejecutor de la automatización (el “cliente”, encargado de definir qué debe hacerse y en qué pasos). Básicamente, toda la parte compleja (determinar cómo hacer que la automatización funcione en una plataforma específica) puede manejarse en el servidor, mientras que los más sencillos, pueden escribirse en cualquier lenguaje de programación. Estos clientes con necesidades más sencillas, solo necesitan codificar solicitudes HTTP al servidor de manera adecuada para su lenguaje.

Hay algunos puntos clave que se deben entender como usuario de Appium:

- Appium es un servidor HTTP: debe ejecutarse como un proceso en una computadora mientras se necesite para la automatización. Además, debe

ser accesible en la red desde cualquier computadora que se vaya a usar para ejecutar la automatización.
- Para usar Appium, se necesita un cliente Appium en el lenguaje elegido.
- El servidor Appium y el cliente Appium no necesita ejecutarse en la misma computadora. Solo es necesario que el cliente pueda enviar solicitudes HTTP al servidor a través de una red. Esto hace que sea muy fácil usar proveedores en la nube para Appium, ya que pueden alojar el servidor Appium junto con sus drivers y dispositivos relacionados.

Cabe destacar que no todo está relacionado con el desarrollo de pruebas, sino con el uso de Appium y sus bibliotecas con el propósito de automatización. Esta es una de las ventajas de la accesibilidad universal de Appium, ya que puede integrase con cualquier conjunto de herramientas que se consideren útiles para cada caso.

2.6.5. Clientes Appium

En este marco, el servidor Appium, interactúa directamente con los dispositivos bajo prueba para ejecutar tareas de automatización. El cliente, manejado por el autor de las pruebas, envía comandos al servidor a través de una red y procesa las pruebas del servidor para determinar el éxito de los comandos o para recuperar información específica sobre el estado de la aplicación [73].

El conjunto de comandos de automatización disponibles depende de los drivers y complementos utilizados durante la sesión. Los comandos estándar suelen incluir operaciones como encontrar elementos, hacer click, recuperar el código fuente de la página y realizar capturas de pantallas. Estos comandos, definidos anteriormente en la especificación de WebDriver, son independientes del lenguaje y se implementan como parte de una API HTTP, lo que permite acceder a ellos desde cualquier leguaje de programación. Por ejemplo, el método FindElement() corresponde a una solicitud HTTP POST dirigida al endpoint `/session/:sessionid/element`, donde `:sessionid` representa el identificador único de la sesión generado por el servidor al iniciar la sesión [73].

Para facilitar la interacción con esta API, se ha desarrollado una suite de bibliotecas cliente de Appium. Estas bibliotecas abstraen la complejidad de la comunicación HTTP directa con el servidor Appium, proporcionando comandos nativos adaptados a lenguajes de programación específicos. Esta abstracción permite a los autores de pruebas escribir scripts de automatización en lenguajes como Python, JavaScript o Java, sin necesidad de preocuparse por los protocolo HTTP subyacentes [73].

Cada biblioteca cliente de Appium se mantiene de forma independiente, por lo tanto, la disponibilidad de funciones específicas puede variar entre clientes. Algunos pueden ofrecer un conjunto completo de funciones auxiliares, mientras que otros no. Además, la frecuencia de actualizaciones puede diferir entre clientes. Por ello, al seleccionar una biblioteca cliente, es fundamental considerar tanto el lenguaje de programación como el conjunto de características y el estado de mantenimiento de la biblioteca.

2.7. Cucumber

Cucumber es una herramienta que facilita el Desarrollo Guiado por Comportamiento (BDD), que le permite a los equipos de desarrollo escribir especificaciones en leguaje natural que describe el funcionamiento del software [82]. Dichas especificaciones consisten en diferentes ejemplos o escenarios, conformados por Steps, como se observa en el siguiente ejemplo:

```
Scenario: Breaker guesses a word
  Given the Maker has chosen a word
  When the Breaker makes a guess
  Then the Maker is asked to score
```

Cada escenario es una lista de pasos con los que Cucumber trabaja. Cucumber verifica que exista un código asociado a cada paso y reporta si ha pasado o fallado cada escenario.

Estas especificaciones se escriben en un lenguaje llamado Gherkin, que es entendible tanto para personal técnico como no técnico.

Gherkin es un conjunto de reglas gramaticales que proporcionan una estructura para las especificaciones escritas en lenguaje natural. Permite que las descripciones sea interpretadas por humanos y por Cucumber.

Como podemos ver en la figura 7.1., Gherkin sirve diferentes propósitos [82]:

- Posee especificaciones ejecutables, de forma que se asegura que el código cumpla con los requisitos.
- Automatización de test utilizando Cucumber.
- Documentación que refleja como el sistema funciona realmente.

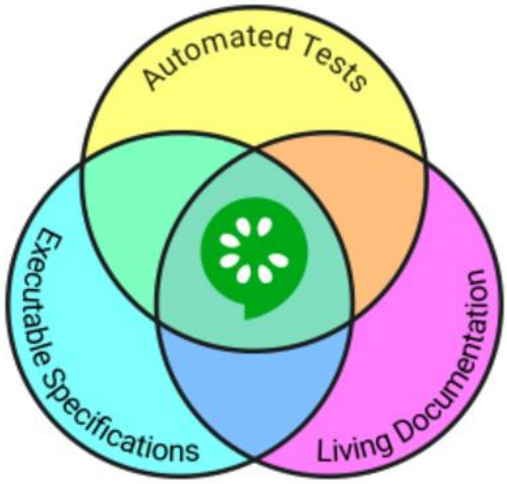


Figura 14 -2.13. Propósitos de Gherkin [82]

Los documentos con los escenarios y Steps se almacenan en archivos .feature y suelen versionarse junto con el código fuente del proyecto [82].

2.7.1. Definición de Steps

La definición de Steps es, básicamente, métodos con expresiones enlazadas a los Steps de Gherkin. Cuando Cucumber ejecuta un Step de Gherkin en un escenario, buscará una definición de Step que coincida para ejecutarlo [83].

Para ser más descriptivos, se creará el siguiente escenario:

```
Scenario: Some cukes
  Given I have 48 cukes in my belly
```

La parte “I have 48 cukes in my belly” del Step (la que sigue después de la palabra “Given”) debe coincidir con la siguiente definición del Step:

```
package com.example;
import io.cucumber.java.en.Given;

public class StepDefinitions {
    @Given("I have {int} cukes in my belly")
    public void i_have_n_cukes_in_my_belly(int cukes) {
        System.out.format("Cukes: %n\n", cukes);
    }
}
```

Cada Step puede tener un resultado diferente [84]:

- Success: cuando Cucumber encuentra una definición de Step correcta, la ejecuta y si este bloque de código no contiene ningún error, el Step es marcado como success (verde).
- Undefined: cuando Cucumber no encuentra ninguna definición de Step que coincida con el que trata de ejecutar, ese Step es marcado como Undefined y los Step siguientes se saltan.
- Failed Step: cuando una definición de Step es ejecutada y el bloque de código presenta un error, el Step es marcado como Failed (rojo).
- Skipped: los Steps consiguientes a los Steps marcados como undefined o failed nunca son ejecutados, por lo que son marcados como Skipped.

2.7.2. Hooks

Los Hooks son bloques de código que puede ejecutarse en diferentes puntos de una ejecución de Cucumber. Son normalmente usados para configurar o borrar entornos de desarrollo en cada ejecución [84].

Existen múltiples hooks, como:

- Before: se ejecuta antes de cada escenario.

```
@Before
public void doSomethingBefore() {
}
```

- After: se ejecuta después del último Step de cada escenario, incluso cuando el Step ha sido marcado como Failed, Undefined o Skipped.

```
@After
public void doSomethingAfter(Scenario scenario){
    // Do something after after scenario
}
```

- BeforeStep: se ejecuta antes de cada Step.

```
@BeforeStep
public void doSomethingBeforeStep(Scenario scenario){
}
```

- AfterStep: se ejecuta después de cada Step, independientemente del resultado de este.

```
@AfterStep
public void doSomethingAfterStep(Scenario scenario){
}
```

- BeforeAll: se ejecuta antes de todos los escenarios.

```
@BeforeAll
public static void beforeAll() {
}
```

- AfterAll: se ejecuta después de que todos los escenarios han sido ejecutados.

```
@AfterAll
public static void afterAll() {
}
```

2.7.3. Gherkin

Como mencionamos anteriormente, Gherkin es un lenguaje utilizado para estructurar especificaciones o requisitos en Cucumber, utilizando palabras claves que otorgan significado a dichas especificaciones. Cada palabra clave es capaz de traducirse a diferentes idiomas, permitiendo a Gherkin ser accesible en diversos lenguajes [85].

Un documento Gherkin típico comienza con la palabra clave Feature, seguida por una breve descripción de la funcionalidad a probar. Debajo de la palabra clave se puede extender la descripción, dando más contexto. Estas líneas son ignoradas por Cucumber durante la ejecución, pero son útiles para la documentación de la prueba [85].

Después, se define el escenario utilizando la palabra clave Scenario (o Example), que describe la situación específica a probar. Cada escenario contiene una secuencia de pasos (Steps) que comienza con las palabras clave Given, When, Then, And o But. Estos pasos detallas el contexto inicial, los eventos y los resultados del escenario.

Palabras clave principales [85]:

- Feature: describe una característica o funcionalidad del software.
- Example o Scenario: define un ejemplo concreto.
- Given, When, Then, And, But: definen los pasos dentro de un escenario.
- Background: proporciona contexto común para multiples escenarios.

Un ejemplo de esto, sería:

```
Feature: Adivinar la palabra

  El juego de adivinar la palabra es un juego por turnos para dos
jugadores.
  El Creador elige una palabra para que el Adivinador la adivine.
  El juego termina cuando el Adivinador adivina la palabra del
Creador.

  Scenario: El Creador inicia un juego
    When el Creador inicia un juego
    Then el Creador espera a que un Adivinador se una

  Scenario: El Adivinador se une a un juego
    Given el Creador ha iniciado un juego con la palabra "casa"
    When el Adivinador se une al juego del Creador
    Then el Adivinador debe adivinar una palabra de 4 caracteres
```

3. Framework de automatización de tests

El objetivo de este trabajo es el desarrollo de un framework de automatización de pruebas tanto web como móviles, que permita el uso de tecnologías de automatización web y móvil combinadas en una sola herramienta. La unión de estas dos herramientas hace posible la coexistencia de pruebas web y móviles en un mismo framework. Esta innovación permite el desarrollo de scripts comunes para pruebas de aplicaciones web y móviles (aunque también se pueden desarrollar scripts específicos para web y para móviles), lo que significa un gran avance en el desarrollo de framework, ya que normalmente se emplean frameworks separados para cada tipo de plataforma.

Este framework además de ahorrar costes, en cuanto a tiempo y recursos, permite la participación de todos los miembros del equipo, agregando herramientas que traduzcan el código de los scripts de pruebas en lenguaje natural y herramientas que muestres los resultados de las pruebas en forma de reportes.

Entrando en una parte más técnica, se combina el uso de Selenium para pruebas en aplicaciones web y Appium para la automatización en dispositivos móviles. Para la demostración del resultado de las pruebas se utiliza la herramienta Allure Reports y complementario a su ejecución en local, se utiliza SauceLabs para la ejecución de pruebas en la nube.

La elección de Selenium como herramienta base para la ejecución de las pruebas web, se debe a su soporte para múltiples navegadores web, lenguajes de programación y su creciente reputación entre herramientas de automatización. Específicamente se ha utilizado el protocolo WebDriver, que permite la ejecución de las pruebas a través de sus drivers y se integra fácilmente con cualquier framework.

Como Selenium no extiende esas capacidades al dominio móvil, se ha decidido utilizar Appium para la automatización de aplicaciones nativas e híbridas de dispositivos móviles Android. Su ventaja principal es el hecho de que, como esta es derivada del protocolo de WebDriver de Selenium, es capaz de reutilizar el código creado para las pruebas web, evitando el tener que reescribir código específico para las pruebas móviles. Esto es una gran innovación, ya que, normalmente se utilizan dos frameworks separados para la automatización de pruebas web y móviles. Con este framework se utilizaría el mismo código para la ejecución de pruebas móviles y web. Además de todo esto, Appium cuenta también con un sistema extensible que permite la personalización de plugins y drivers.

Para la definición de pruebas se ha utilizado Cucumber, una herramienta que permite la escritura de escenarios de pruebas en un lenguaje natural (Gherkin). Como las pruebas son escritas en lenguaje natural facilita el entendimiento y la colaboración entre equipos que no entienden los lenguajes de programación contemporales. Esto también asegura que las pruebas estén bien escritas y que cumplan con los requerimientos establecidos.

También, para la ejecución de las pruebas se ha elegido Junit, ya que se integra fácilmente con Cucumber y ofrece métodos convenientes para la aserción y organización de las pruebas.

Para la visualización de los resultados de las pruebas, se decidió que verlas en la terminar no era una opción adecuada, ya que el framework está diseñado

para ser utilizado por personas técnicas y no técnicas. Para solucionar esto se optó por utilizar Allure Reports, una herramienta de generación de reportes. Esta proporciona una interfaz para observar los resultados de las pruebas, que se ha personalizado para reflejar el entorno en el que se ha ejecutado el test y en caso de fallo, la razón de este, también se ha agregado una captura del momento en el que ha ocurrido.

Asimismo, el framework permite la ejecución de pruebas en SauceLabs, una plataforma de pruebas en la nube que soporta múltiples dispositivos. Con SauceLabs se es capaz de ejecutar pruebas en dispositivos reales y en tiempo real, lo que elimina la necesidad de tener infraestructura real local, reduce los costos y mantenimiento y permite asegurar que las pruebas funcionen en diferentes configuraciones y ambientes.

3.1. Entorno de desarrollo y pruebas

El desarrollo del framework de automatización se llevó a cabo utilizando el entorno de desarrollo IntelliJ IDEA, el lenguaje de programación Java, y el sistema de gestión de dependencias Maven. La elección de estas herramientas se basó en la facilidad de integración con las mencionadas anteriormente.

En cuanto al entorno de desarrollo de las pruebas, se diseñó de tal manera para que pudiera ser configurable, dependiendo de las necesidades de cada ejecución. Se logró utilizando las variables de entorno que proporciona IntelliJ IDEA antes de lanzar las pruebas. Estas variables se crearon para controlar el comportamiento del framework y las funcionalidades activas.

Esto permite que la misma prueba pueda ser ejecutada en diferentes configuraciones y entornos con cambios sencillos y fáciles de mantener. El proceso funciona de la siguiente manera:

- Configuración de variables de entorno en IntelliJ: antes de ejecutar las pruebas es necesario que el usuario defina las variables de entorno requeridas para las pruebas. Estas variables actúan como parámetros que el framework utiliza para ajustar su comportamiento.
- Lectura de las variables del framework: durante la ejecución, el framework accede las variables previamente creadas mediante System.getenv().
- Ejecución condicional: según las variables establecidas, el framework selecciona los drivers adecuados, define el entorno de ejecución y los recursos necesarios.

Las variables de entorno de las pruebas definidas en el proyecto son las siguientes:

- Platform:
    - Valores posibles: Android, Web.
    - Descripción: Especifica la plataforma para realizar las pruebas, ya sea Android o Web. Debe proporcionarse con valores válidos para que las pruebas funcionen correctamente.
- Browser:
    - Valores posibles: Chrome, Firefox, Edge, Safari.
    - Descripción: Define el navegador que se utilizará para las pruebas web. Si se establece como null, el navegador predeterminado será Chrome. Para usar Safari en macOS,

es necesario habilitar la automatización con el comando: safaridriver --enable.

- Application:
  - Descripción: Debe ser el nombre de la aplicación web que se está probando. No puede ser null y debe proporcionarse con valores válidos para que las pruebas funcionen correctamente.
- Resolution:
  - Valores posibles: Los valores permitidos para la variable de entorno están definidos en el anexo.
  - Descripción: Especifica la resolución de pantalla para las pruebas web. Si es null o está vacía, se establece una resolución predeterminada de 1024x768.
- Language:
  - Valores posibles: en-GB, en-US, es-ES, fr-FR, entre otros.
  - Descripción: especifica el idioma de ejecución de las pruebas. Si es null o está vacía, el idioma predeterminado es es-ES.
- App:
  - Descripción: especifica la ruta o el nombre del archivo APK o IPA que se instalará y probará en un dispositivo móvil. Las pruebas funcionarán si se establece el campo App, o si se proporcionan ambos campos, AppIdentifier y AppActivity.
- AppActivity:
  - Descripción: representa la actividad de la aplicación para Android que se está probando. Las pruebas funcionarán si se proporcionan tanto AppIdentifier como AppActivity.
- AppIdentifier:
  - Descripción: representa el paquete de la aplicación que se está probando. Las pruebas funcionarán si se proporcionan tanto AppPackage como AppActivity.
- UDID:
  - Descripción: Representa el identificador único del dispositivo (UDID) utilizado para pruebas móviles. Si la variable es null, el valor predeterminado será el UDID del emulador actual.
- Provider:
  - Valores posibles: Local, SauceLabs.
  - Descripción: Especifica el proveedor o entorno para ejecutar las pruebas. Puede ser en SauceLabs o en un entorno local.
- User:
  - Descripción: Representa el nombre de usuario de tu cuenta de SauceLabs.
- AccessToken:
  - Descripción: Representa el token de acceso API de tu cuenta de SauceLabs.

La gestión de todas estas variables se hace en una clase llamada LocalEnvironment(). En esta se crean métodos que almacenan los valores de las variables de entorno y, como se puede observar en el código de la clase, se hace la gestión del formato. En el caso de las variables “Browser” y “Language” también se gestiona la asignación de valores por defecto, en caso de que alguna

de estas sea establecida a “null”. Para la variable de “Browser” el valor por defecto es “Chrome” y para la variable “Language” es español.

Para la variable “Application” se tuvieron consideraciones especiales. Al iniciar el proyecto, la variable “Application” se llamaba “URL” y para activarla, básicamente, se le pasaba la URL de la aplicación a probar. Esto parecía un poco rudimentario, por lo que se optó a cambiar la variable y asignarle el nombre de la aplicación a probar. Cuando se llama a esta variable en el código con getApplication(), el método se encarga de buscar en un archivo yaml llamado webConfiguration.yaml (mostrado en el anexo) una coincidencia con el valor de la variable. En ese archivo yaml se encuentran las URL de las aplicaciones probadas y sus respectivos nombre. Con estos cambios se establece de una manera más sencilla la variable “Application” y facilita su modificación.

```
public class LocalEnvironment {

    public static String getPlatform() {
      return System.getenv("Platform");
    }
    public static String getProvider() {
      return System.getenv("Provider");
    }
    public static String getApplication() {
      return Objects.nonNull(System.getenv("Application"))
          ? System.getenv("Application").toLowerCase()
          : "";
    }
    public static String getBrowser() {
      if (!FrontEndOperation.isNullOrEmpty(System.getenv("Browser")))
{
        return System.getenv("Browser").toLowerCase();
      } else {
        return "chrome";
      }
    }
    public static String getResolution() {
      return System.getenv("Resolution");
    }
    public static String getUdid() {
      return System.getenv("Udid");
    }
    public static String getUser() {
      return System.getenv("User");
    }
    public static String getAccessToken() {
      return System.getenv("AccessToken");
    }
    public static String getApp() {
      return System.getenv("App");
    }
    public static String getAppIdentifier() {
      return System.getenv("AppIdentifier");
    }
    public static String getAppActivity() {
      return System.getenv("AppActivity");
    }
    public static boolean isMobile() {
      return !isWeb();
    }
    public static boolean isWeb() {
      return System.getenv("Platform").equalsIgnoreCase("Web");
    }
```

```
   public static boolean isAndroid() {
      return System.getenv("Platform").equalsIgnoreCase("Android");
    }
    public static boolean isWindows() {
      return
System.getProperty("os.name").toLowerCase().contains("win");
    }
    public static boolean isMac() {
      return
System.getProperty("os.name").toLowerCase().contains("mac");
    }
    public static String getApplicationUrl() throws
IllegalArgumentException {
      Map<String, Map<String, String>> environment =
DriverConfiguration.loadCapabilitiesWeb();
      Map<String, String> urls = environment.get("url");
      String url = null;
      if (!urls.containsKey(getApplication()) ||
getApplication().isBlank()) {
        throw new IllegalArgumentException("Application not found");
      }
      for (Map.Entry<String, String> entry : urls.entrySet()) {
        String application = entry.getKey().toLowerCase();
        String applicationUrl = entry.getValue();
        if (Objects.equals(application, getApplication())) {
          url = applicationUrl;
          break;
        }
      }
      return url;
    }
    public static String getLanguage() {
      String language = System.getenv("Language");
      if (FrontEndOperation.isNullOrEmpty(language)) {
        language = "es-ES";
      }
      if (!language.matches(LANGUAGE_REGEX)) {
        throw new IllegalArgumentException("Invalid language format.
It should be xx-XX");
      }
      return language;
    }
    public static String getLanguageCode() {
      return getLanguage().split("-")[0];
    }
    public static String getCountryCode() {
      return getLanguage().split("-")[1];
    }
  }
```

Con estas variables, somos capaces de que el framework se ajuste a las necesidades de cada ejecución. Con el enfoque modular de este, facilitamos las pruebas y aseguramos que estas funcionen en distintos escenarios y plataformas sin necesidad de cambiar el código.

3.2. Drivers

Dado que el framework utiliza los drivers de Selenium para las pruebas web y móviles (a través de Appium), se decidió centralizar la gestión de los drivers en una sola clase.

Para las pruebas web, se emplearon los drivers oficiales de Selenium para los navegadores más utilizados: Chrome, Firefox, Edge y Safari. Para esto se creó un método que se encargó de la creación de los drivers dependiendo del valor de la variable de entorno browser.

Para la gestión de los drivers se tuvieron en cuenta los diferentes aspectos:

- Compatibilidad con Versiones: Se comprobó través de la documentación de Selenium que los drivers oficiales estuvieran siempre sincronizados con los navegadores correspondientes. Esto facilitó el trabajo, no se tuvo que crea código que gestionara los casos de error por versiones no coincidentes, ya que cualquier discrepancia entre versiones puede ocasionar errores en la ejecución de las pruebas.
- Configuración dinámica: la selección del driver para cada prueba web depende de la variable de entorno Browser. Esta variable permite elegir el navegador en el que se ejecutaran las pruebas web.
- Gestión de drivers en SauceLabs: en las pruebas ejecutadas en SauceLabs, se asigna como el driver de ejecución el driver propio de SauceLabs, ya que este se encarga de proporcionar los navegadores y versiones correspondientes a sus entornos.

En el código siguiente se puede observar cómo se crea el driver de la sesión, dependiendo de si el valor de la variable “Provider” es “Local” o “SauceLabs”. Si el valor de “Provider” resulta ser “SauceLabs” se le asigna al driver creado el driver propio de SauceLabs. Si, en cambio, el valor de “Platform” es “Local” o su equivalente, “null” se asigna un driver especifico dependiendo de valor de la variable “Platform”. Si la variable “Platform” es “Web” se procede a llamar al método configureWebDriver(), se verá más adelante. Además de llamar a dicho método se gestionan configuraciones comunes a todos los WebDrivers como la resolución, que depende de la variable “Resolution” y se asigna la URL de la página a probar. Si el valor de la variable “Platform” resulta ser “Android” se le

asigna al driver creado una nuevo driver de tipo AndroidDriver(), el cual se crea llamando al método fillCapabilities(), que se detallará más adelante.

```
public static WebDriver getDriver() {
        if (currentDriver != null) {
            return currentDriver;
        }
        if (Objects.nonNull(getProvider()) &&
getProvider().equalsIgnoreCase("SauceLabs")) {
            currentDriver = SauceLab.getSauceDriver();
        } else {
            if (isWeb()) {
                Dimension windowResolution =
ScreenResolution.getResolutionFromEnv();
                String url = setURL();

                WebDriver driver = configureWebDriver();
                driver.manage().window().setSize(windowResolution);
                driver.get(url);
                currentDriver = driver;
            } else if (isAndroid()) {
                try {
                    URL url = new URL(DRIVER_URL);
                    currentDriver = new AndroidDriver(url,
fillCapabilities());
                } catch (MalformedURLException e) {
                    throw new RuntimeException(e);
                }
            }
        }
        return currentDriver;
    }
```

Para cada opción de navegador, en lugar de descargar un driver específico para cada uno, se utilizaron los drivers oficiales de Selenium y se establecieron las opciones deseadas, en nuestro caso, solo se establecieron como opciones el lenguaje de las pruebas, que también iba determinado por las variables de entorno de IntelliJ. Como se puede ver en el fragmento de código, se asigna un driver específico dependiendo del valor de la variable "Browser" y se establecen las opciones de manera correspondiente.

```
private static WebDriver configureWebDriver() {
    WebDriver driver;
    String browser = LocalEnvironment.getBrowser();

    switch (browser) {
        case "edge":
            EdgeOptions edgeOptions = new EdgeOptions();
            edgeOptions.addArguments("--lang=" +
LocalEnvironment.getLanguage());
            driver = new EdgeDriver(edgeOptions);
            break;
        case "firefox":
            FirefoxProfile profile = new FirefoxProfile();
            profile.setPreference("intl.accept_languages",
LocalEnvironment.getLanguage());
            FirefoxOptions firefoxOptions = new FirefoxOptions();
            firefoxOptions.setProfile(profile);
            driver = new FirefoxDriver(firefoxOptions);
            break;
        case "safari":
            driver = new SafariDriver();
            break;
        default:
            ChromeOptions chromeOptions = new ChromeOptions();
            chromeOptions.addArguments("--lang=" +
LocalEnvironment.getLanguage());
            driver = new ChromeDriver(chromeOptions);
            break;
    }
```

Para las pruebas móviles, se casteó el driver oficial de Selenium a un AndroidDriver , que extiende el protocolo WebDriver para plataformas móviles. En el caso de las pruebas móviles es necesario establecer en las opciones (llamadas capabilities en pruebas móviles) variables como la plaforma, la appActivity, el AppIdentifier, el idioma, entre otros, para su correcto funcionamiento. Para esto se creó un método llamado fillCapabilities(), que se encargó de establecer estas opciones y rellenar sus valores desde las variables de entorno. En ese mismo método se añadieron opciones adicionales para personalizar la ejecución de las pruebas, en nuestras pruebas de ejemplo se utilizaron las opciones de NoReset a true y NewCommandTimeout a 90. Se

implementó un método que cargara dichas opciones desde un archivo yaml "androidConfiguration.yaml" (mostrado en el anexo), para facilitar su edición y montaje.

```
private static MutableCapabilities fillCapabilities() throws
IllegalArgumentException {
    Map<String, Map<String, String>> environment;
    Map<String, String> capabilities;
    MutableCapabilities filledCapabilities = new DesiredCapabilities();

    filledCapabilities.setCapability("platformName",
LocalEnvironment.getPlatform());
    filledCapabilities.setCapability("udid",
LocalEnvironment.getUdid());
    environment = loadCapabilitiesMobile(Constants.ANDROID_CONFIG);
    capabilities = environment.get("capabilitiesAndroid");
    filledCapabilities.setCapability("appPackage",
LocalEnvironment.getAppIdentifier());
    filledCapabilities.setCapability("language",
LocalEnvironment.getLanguageCode());
    filledCapabilities.setCapability("locale",
LocalEnvironment.getCountryCode());
    String apk = LocalEnvironment.getApp();
     if (Objects.nonNull(apk) && !apk.isEmpty()) {
        filledCapabilities.setCapability(
               "app", Paths.get(Constants.RESOURCE_PATH +
apk).toAbsolutePath().toString());
     } else {
        filledCapabilities.setCapability("appActivity",
LocalEnvironment.getAppActivity());
     }

    if (Objects.isNull(capabilities)) {
        throw new IllegalArgumentException("Capabilities are not set");
    }
for (Map.Entry<String, String> entry : capabilities.entrySet()) {
        String capabilityName = entry.getKey();
        String capabilityValue = entry.getValue();
        if (Objects.nonNull(capabilityValue) &&
!capabilityValue.isEmpty()) {
            filledCapabilities.setCapability(capabilityName,
capabilityValue);
        } else {
            throw new IllegalArgumentException("Capabilities cannot be
blank");
        }
    }
return filledCapabilities;
}
```

3.3. Allure Reports

Allure Report es una herramienta de generación de reportes de pruebas que proporciona una interfaz visual detallada para el resultado de las pruebas. En este proyecto se utilizó Allure Reports con la finalidad de hacer más sencillo el entendimiento de los resultados para los miembros del equipo no técnicos [74].

Características de Allure Reports [75]

- Interfaz Visual: Allure Reports genera reportes en formato web que incluye métricas y diagramas que facilitan la interpretación de resultados. Para los usuarios es sencillo explorar los detalles de la ejecución, errores y tiempo de ejecución.
- Integración con framework: es compatible con muchos frameworks de automatización como Junit, TestNG y Cucumber.
- Personalización: Allure Report permite una alta personalización, lo que ayuda a que sea capaz de adaptarse a cualquier proyecto. Esto incluye la posibilidad de agregar etiquetas personalizadas e incluso redefinir como se presentan y organizan los detalles del reporte.

En nuestro proyecto, utilizamos la gran capacidad de personalización de Allure Reports para mostrar las variables de entorno con las que se ejecuta el test. En los casos de errores, también decidimos añadir capturas de pantalla para ubicar el momento en el que ocurren fallos. Para esto se implementó una clase dedicada a hacer las capturas de pantallas, luego en la clase AllureReport se llama a esta clase para mostrar las capturas en caso de error. En esta clase también se llaman a las variables de entorno activas para formar la descripción del reporte.

Al terminar el test, se abre automáticamente la página del reporte. En primer lugar se muestra el número de tests ejecutados, a su lado se observa el porcentaje de tests pasado y el porcentaje que han fallado. Si queremos ver en detalle la información de cada test ejecutado, debemos entrar a la página de cada test.

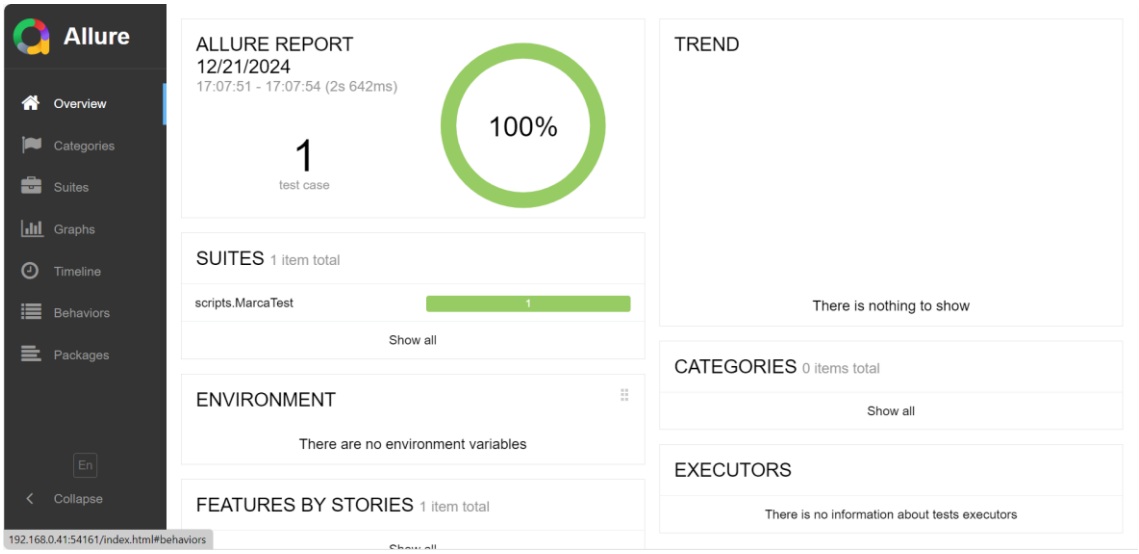


Figura 15 - 3.1. Página principal de Allure Report

En nuestro ejemplo comprobamos la existencia de una imagen en la página de

noticias Marca. El test ha salido exitoso, por lo que sale marcado como Passed. A la derecha se encuentra la descripción de las variables de entorno con las que se ejecutó el test, junto a la duración de este.

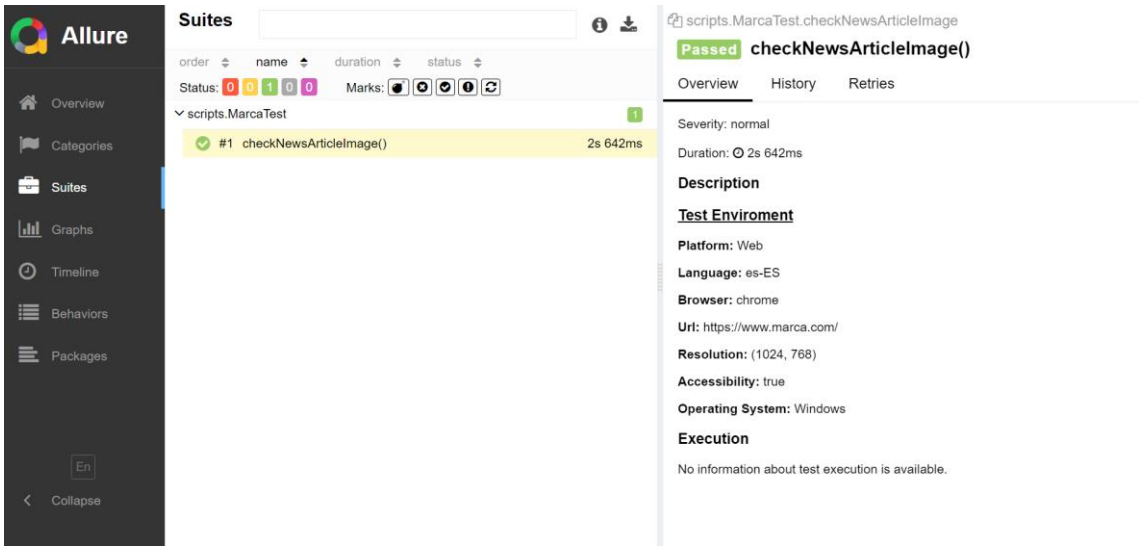


Figura 16- 3.2. Reporte de Allure test exitoso

En caso de error o bloqueo, la página principal de Allure es la misma, solo cambia el porcentaje de error en la ejecución. En cambio, cuando observamos un test en específico, podemos ver que se muestra la razón por la que el test no funciona, en nuestro caso, por un elemento no interactuable. También, tenemos anexado una captura del momento exacto en el que ha ocurrido el error.

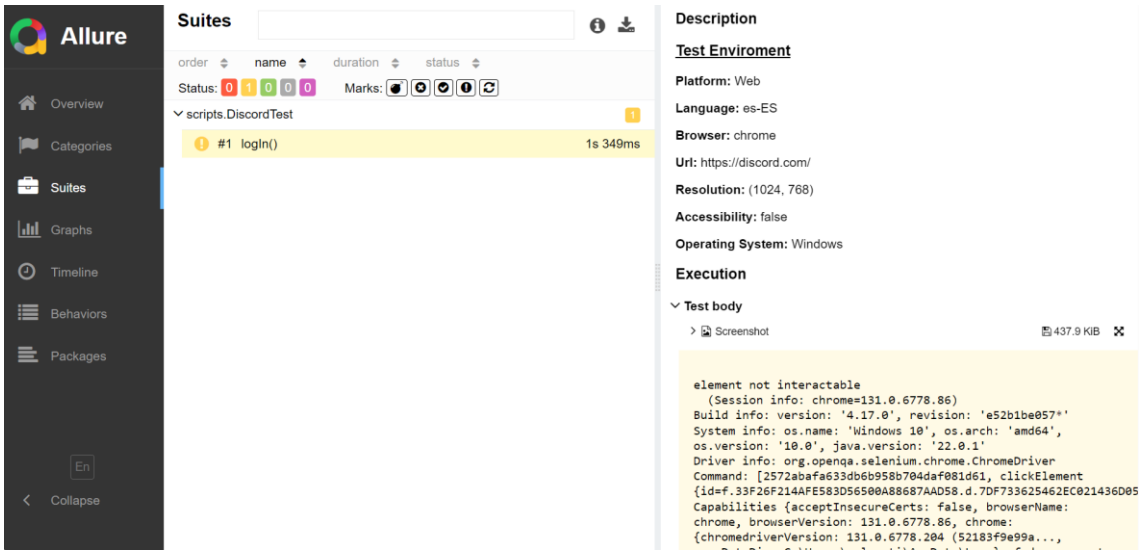


Figura 17 - 3.3. Reporte de Allure fallido

Para añadir todos los detalles en el apartado de descripción de los reportes de Allure se creó un método llamado setTestDescription() que tomaba de los valores de las variables de entorno y las asignaba a un StringBuilder vacío. Una vez se añadidas todas las variables de entorno activas se devuelve dicho StringBuilder. Posteriormente en el método fillReportInfo() se llama a la función setTestdescription() y se envía a la descripción en formato HTML a Allure.

```
private static String setTestDescription() {
        WebDriver driver = DriverConfiguration.getDriver();
        StringBuilder description = new StringBuilder();
        AppiumDriver driverMobile;
        String os = null;
        description.append("<h3 style=\"text-decoration:
underline;\">Test Enviroment</h3>");
        description.append("<p><b>Platform:</b>
").append(LocalEnvironment.getPlatform()).append("</p>");
        description.append("<p><b>Language:</b>
").append(LocalEnvironment.getLanguage()).append("</p>");
        if (LocalEnvironment.isMobile()) {
            driverMobile = (AndroidDriver) driver;
            String appActivity = LocalEnvironment.getAppActivity();
            String deviceName =
driverMobile.getCapabilities().getCapability("deviceName").toString();
            description.append("<p><b>Device Name:</b>
").append(deviceName).append("</p>");               String
platformVersion =
                    driverMobile.getCapabilities().getCapability("plat
formVersion").toString();

            description
                    .append("<p><b>Platform
Version:</b>".concat("Android "))
                    .append(platformVersion)
                    .append("</p>");
            if (!FrontEndOperation.isNullOrEmpty(appActivity)) {
                description.append("<p><b>App Activity:</b>
").append(appActivity).append("</p>");
            }
        description
                .append("<p><b>Udid:</b> ")
                .append(
                        driverMobile
                                .getCapabilities()
                                .getCapability(isAndroid() ?
"deviceUDID" : "udid")
                                .toString())
                .append("</p>");
```

```
description
                .append("<p><b>App Identifier:</b> ")
                .append(LocalEnvironment.getAppIdentifier())
                .append("</p>");
        String apk = LocalEnvironment.getApp();
        if (!FrontEndOperation.isNullOrEmpty(apk)) {
            description.append("<p><b>App:</b>
").append(apk).append("</p>");
        }

        if (LocalEnvironment.isWindows()) {
            os = "Windows";
        } else if (LocalEnvironment.isMac()) {
            os = "Mac";
        } else {
            os = "linux";
        }
        description.append("<p><b>Browser:</b>
").append(LocalEnvironment.getBrowser()).append("</p>");
        description
                .append("<p><b>Url:</b> ")
                .append(LocalEnvironment.getApplicationUrl())
                .append("</p>");
        description
                .append("<p><b>Resolution:</b> ")
                .append(ScreenResolution.getResolutionFromEnv())
                .append("</p>");
        description
                .append("<p><b>Accessibility:</b> ")
                .append(LocalEnvironment.getAccessibility())
                .append("</p>");
        description.append("<p><b>Operating System:</b>
").append(os).append("</p>");
    }
    description.append(checksHtml);
    checksHtml = "";
    return description.toString();
    }
```

```
public static void fillReportInfo() {
    descriptionHtml = setTestDescription();
    Allure.descriptionHtml(descriptionHtml);
    descriptionHtml = "";
  }
```

3.4. SauceLabs

SauceLabs es una plataforma de pruebas en la nube que ejecuta pruebas automatizadas en una multitud de dispositivos y navegadores. Su principal ventaja es la eliminación de la necesidad de mantener una infraestructura local, ya que proporciona desde la distancia una gran variedad de dispositivos reales o simulados [76].

Características de SauceLabs [76]

- Múltiples dispositivos y navegadores: SauceLabs ofrecen acceso a una gran selección de navegadores, versiones y dispositivos reales, así como emuladores y simuladores para pruebas móviles.
- Entorno escalable: la infraestructura en la nube permite escalar las pruebas fácilmente, ejecutándolas en paralelos, en diferentes dispositivos, para así reducir los tiempos de ejecución.
- Informes: SauceLAbs genera reportes detallados que incluyen logs, capturas de pantalla y grabaciones de las ejecuciones, para facilitar la identificación de problemas.
- Integración con diferentes herramientas: es compatible con Selenium, Appium y otras herramientas.

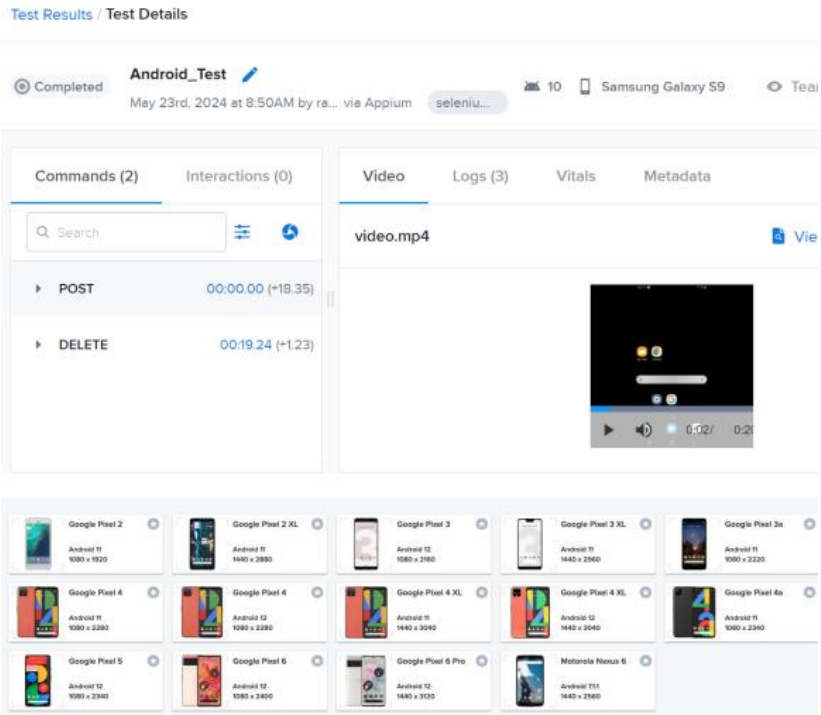


Figura 18 – 3.4. Entorno de pruebas SauceLabs

En este proyecto, SauceLabs fue utilizado como un entorno secundario para la ejecución de las pruebas en la nube, para aplicaciones web y móviles. Su implementación valió para comprobar el funcionamiento de las pruebas móviles en dispositivos reales.

El framework fue configurado para integrarse con SauceLabs de la siguiente manera:

- User y AccessToken: para poder ejecutar las pruebas en la nube, es necesario tener una cuenta creada en SauceLAbs y establecer las

variables de entorno “User” y “AccessToken” con el usuario y accessToken de la cuenta.

- Provider: la variable de “Provider” debe estar establecida con el valor “SauceLabs”.
- Resto de variables: las variables restantes se pueden establecer con las configuraciones deseadas para la prueba. Las pruebas remotas soportan pruebas móviles y pruebas web.

Para la configuración del driver de SauceLabs se creó una clase para la gestión de todas las variables de entorno de la pruebas en la nube. En esta se crearon métodos para aplicar las opciones determinadas (configureSauceAndroid() y configureSauceWeb()) que luego se agregaron a la hora de la creación del driver remoto. La gestión del driver remoto se puede observar en el código siguiente y su implementación se puede observar en el método de gestión de drivers centralizado getDriver() mostrado en el punto 2.1..

```
      public static WebDriver getSauceDriver() {
        if (isWeb()) {
            Dimension windowResolution =
ScreenResolution.getResolutionFromEnv();
            String url = setURL();
            WebDriver driver = configureSauceWeb();
            driver.manage().window().setSize(windowResolution);
            driver.get(url);
            return driver;
        } else {
            return configureSauceAndroid();

        return null;
    }
```

3.5. Pruebas

Para la comprobación de todas estas funciones se crearon diferentes pruebas. Para verificar que el framework funcionara correctamente lanzando pruebas en local y en la nube se creó una prueba que comprobaba la existencia de una noticia en la página de Marca. Esta prueba la utilizaremos como ejemplo para mostrar los pasos que hay que seguir para la creación de pruebas.

- Creación de elementos y método auxiliares: para poder interactuar con los elementos es necesarios almacenarlos en una variable de tipo WebElement utilizando las anotaciones de Selenium @FindBy y las de Appium @AndroidFindBy. Esta variable es interactuable tanto por los drivers de Selenium como los drivers de Appium.

```
@FindBy(id = "ue-accept-notice-button")
    WebElement acceptCookies;

    @FindBy(partialLinkText = "Noticia que existe")
    @AndroidFindBy(
            xpath =
                    "(//android.widget.LinearLayout[@resource-
id=\"com.iphonedroid.marca:id/portadilla_container\"])[3]")
    WebElement randomNotice;

    @FindBy(id = "buttonYes")
    WebElement ageButton;
```

Luego también es necesaria la creación de método auxiliares para la interacción con dichos elemento. En nuestro caso de ejemplo, hemos creado el método acceptCookies() que le da click al botón de aceptar del popup de las cookies.

```
public void acceptCookies() {
    if (isVisible(acceptCookies)) {
        acceptCookies.click();
    }
}
```

- Creación del escenario de Cucumber: creamos un archivo .feature en el que estará el escenario de la prueba con las anotaciones correspondientes. En nuestro caso hemos creado el siguiente.

```
Feature: Check if a new exist in Marca

  Scenario: Verify if a new exits
    Given I am on the Marca website
    When I look for a notice
    Then I should be able to see the notice
```

- Creación de métodos de prueba: para ejecutar las acciones de los escenarios es necesario crear métodos enlazados a los pasos de Cucumber. Para esto crearemos un método por paso y este se enlazaran con las anotaciones correspondientes.

```
@Given("I am on the Marca website")
    public void iAmOnTheMarcaWebsite() {
        DriverConfiguration configuration = new DriverConfiguration();
        driver = configuration.getDriver();
        controller = new Marca(driver);
    }

    @When("I navigate to a news article")
    public void iNavigateToANewsArticle() {
        controller.acceptCookies();
        controller.goToNotice();
        //controller.acceptAge();
    }

    @Then("I should be able to see if the article contains an image")
    public void iShouldBeAbleToSeeIfTheArticleContainsAnImage() {
        assertTrue("La imagen se muestra en pantalla", true);
    }
```

Siguiendo estos pasos se pueden crear todas las pruebas deseadas.

El framework es capaz de soportar diferentes tipos de pruebas, pero está más enfocado al desarrollo de pruebas funcionales.

- Pruebas funcionales: El framework permite la ejecución de pruebas funcionales con Selenium y Appium a través de la creación de pruebas que sigan flujos específicos. Con esto, se puede comprobar el funcionamiento de cualquier aspecto de una página web o aplicación móvil.
- Pruebas no funcionales: el framework también es capaz de medir aspectos no funcionales, como el rendimiento y la compatibilidad. Con las opciones y capabilities que proporciona Selenium y Appium podemos medir los tiempos de carga y respuesta, y con SauceLabs y las

configuraciones propias del framework podemos evaluar la compatibilidad de la aplicación con diferentes entornos.

- Pruebas unitarias: aunque el framework esta más orientado a pruebas de nivel superior, la integración con Junit permite la ejecución de pruebas unitarias. Se pueden crear pruebas que comprueben el funcionamiento individual de una funcionalidad y comprobar su correcto funcionamiento.
- Pruebas de integración: el framework permite comprobar la interacción entre los diferentes módulos mediante la creación de flujos completos de pruebas. Esto se puede realizar con Selenium y con Appium y comprobar el resultado con los reportes de SauceLabs y Allure.
- Pruebas de aceptación: las pruebas de aceptación aseguran que el sistema cumpla con las especificaciones del cliente. Los escenarios de pruebas escritos con Cucumber (en lenguaje natural), hace mucho más fácil la comprobación, por parte del cliente, de las funcionalidades que se están ejecutando

3.6. Ejemplo de prueba

Para entender mejor el funcionamiento del framework, se explicará cómo funciona, desde la creación de scripts de pruebas hasta su ejecución.

El objetivo del framework es comprobar el funcionamiento de una aplicación, para este ejemplo, comprobaremos el botón de iniciar sesión de la página de Discord.

Para esto debemos comenzar creando un archivo .feature de Cucumber. En este archivo guardaremos los pasos que va a realizar nuestro test. En este caso, el archivo se llamará check_login_button.feature y lucirá de la siguiente forma:

```
Feature: Check if login button works in Discord

  Scenario: Verify if login button works
    Given I am on the Discord main page
    When I click the login button
    Then I should go to the login page
```

En este archivo se utilizan los hooks vistos en el capítulo 2.7.. Se establece en el apartado de Feature una descripción general de lo que va a hacer el test y en los hooks siguientes establecemos las precondiciones y los resultados esperados. Este archivo es el que vamos a ejecutar una vez terminemos todos los pasos.

Lo siguiente sería la creación de una clase en la que se desarrollarán los método que gestionen la automatización de las pruebas. Para la creación de esta clase es necesario anclar los hook del archivo check_login_button.feature en un método especifico. Solo debe existir un método por cada hook.

Además de esto, para la creación de la clase Discord, hay que tener en cuenta diferentes aspectos. Como en el proyecto se sigue el modelo DOM (Document Object Model) es necesario identificar los elementos a utilizar en el test antes de implementar los métodos. En el caso de ejemplo se va a utilizar el botón de inicio de sesión y un texto de la página de inicio de sesión de Discord.

Para esto se utilizan los hooks @FindBy de Selenium (Visto en el capítulo 2.5.2.3.) que asignan el identificador utilizado a un objeto de tipo WebElement. Como Appium deriva de Selenium, tiene su propio hook que asigna de la misma manera un identificador a un objeto de tipo WebElement, por lo que al declararlo junto, si la prueba es Web se utiliza el identificador marcado por @FindBy y si es una prueba móvil, se utiliza el identificador marcado por @AndroidFindBy.

Una vez completada esta parte podemos pasar a la implementación de los métodos. En el caso de ejemplo, se iniciar el driver de la sesión, se hace click al botón guardado en el WebElement y luego se comprueba que se muestre el texto indicado en pantalla, lo que indica que se ha llegado a la página de inicio de sesión.

```
public class Discord {

    private WebDriver driver;

    @FindBy(className = "button-white login-button-js w-button hide-on-
mobile")
    @AndroidFindBy(
        xpath = "//android.widget.Button[@content-desc=\"Iniciar
sesión\"]/android.view.ViewGroup")
    WebElement iniciarSesion;

    @FindBy(className = "title_d10a58")
    @AndroidFindBy(xpath = "//android.view.View[@text=\"¡Espera! ¿Eres un
ser humano?\"]")
  WebElement textoConfirmacion;


    @Given ("I am on the Discord main page")
    public Discord() {
      this.driver = DriverConfiguration.getDriver();
    }

    @When ("click the login button")
    public void clickIniciarSesion() {
      iniciarSesion.click();
    }

    @Then("I should go to the login page")
    public void goLoginPage() {
        FrontEndOperation.checkThat(
            "El texto es igual",
            textoConfirmacion.getText(),
            is(TextTranslation.get("textoConfirmacion")));
      }
}
```

Con esto se tendría terminado el script de la prueba de ejemplo. Para ejecutar el test se deben seguir los siguientes paso.

Primero se deben configurar las variables de entorno con las que queremos lanzar el test. En este caso se comenzaran con las básicas:

- Platform: Web
- Application: Discord

Con estas dos variables establecidas, podemos lanzar cualquier test de web, ya que las demás variables se establecen por defecto. La variable de Browser estaría definida como Chrome, Resolution como 1024x768, Language como es-ES y Provider como Local.

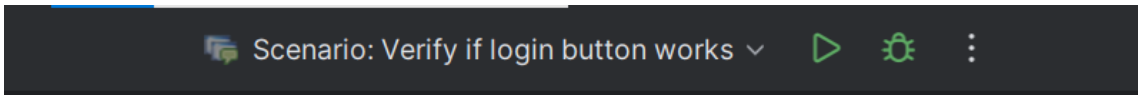
Una vez se llega aquí, ya se puede ejecutar el test. Para hacerlo hay diferentes opciones:

- Ubicar el archivo check_login_button.feature y dar click al play ubicado en la izquierda.

```
Feature: Check if login button works in Discord

  Scenario: Verify if login button works
    Given I am on the Discord main page
    When I click the login button
    Then I should go to the login page
```

- Dar click al play ubicado en la parte superior del IDE



Independientemente de la opción utilizada, cuando se lanza el test se utiliza una clase implementada en el framework llamada CucumberRunner, aquí se detecta que se está utilizando Cucumber por la anotación @RunWith(Cucumber.class), con esto Cucumber toma el proceso de ejecución.

```
@RunWith(Cucumber.class)
@CucumberOptions(
        features = "src/test/resources/features",
        glue = "tests/cucumber_steps"
)
```

Para ubicar los archivos Feature y los método anclados a ellos, se establecieron como CucumberOptions la ubicación de los archivos Feature y la ubicación de las clases en donde se encuentran los métodos, llamado cucumber_steps.

Por cada step del escenario se intenta coincidir con las anotaciones definidas en las clases de cucumber_steps. Si se encuentra una coincidente, se ejecuta la función correspondiente, sino la prueba falla.

Ahora, entrando más a fondo en lo que ocurre cuando se ejecuta un test, en el primer step del escenario se crea una instancia del driver, en este caso un ChromeDriver. Este driver actuara como puente entre Selenium y Chrome, para enviar comandos al navegador y obtener información.

Como se comentó antes, de todos los servicios que ofrece Selenium, en el framework se utiliza el protocolo WebDriver, por lo que al llamar al método click(), en el segundo step, se genera una solicitud HTTP que el driver envía al

navegador y este inyecta un código JavaScript que simula la acción de hacerle click al botón.

Por último, en el tercer step se llama al método checkThat que compara los dos Strings pasados como argumentos y devuelve true si son iguales, comprobando así que nos encontramos en la página de login de Discord.

Una vez se termina el test, se llama a Allure Reports. Allure, durante la ejecución de las pruebas, recolecta los resultados de cada step y genera un archivo JSON llamado allure-results. Al finalizar estas se construye una carpeta llamada allure-report que contiene los archivos HTML con la interfaz gráfica para la visualización de los resultados.

Ese sería el proceso que siguen los test básicos que se pueden hacer en el framework. Pero como este se desarrolló para que se pudieran hacer todo tipo pruebas, modificando las variables de entorno se pueden obtener resultados distintos.

Agregando la variable de entorno “Language = en-EN” se puede obtener los mismo resultado solo que ahora con la página de Discord en inglés.

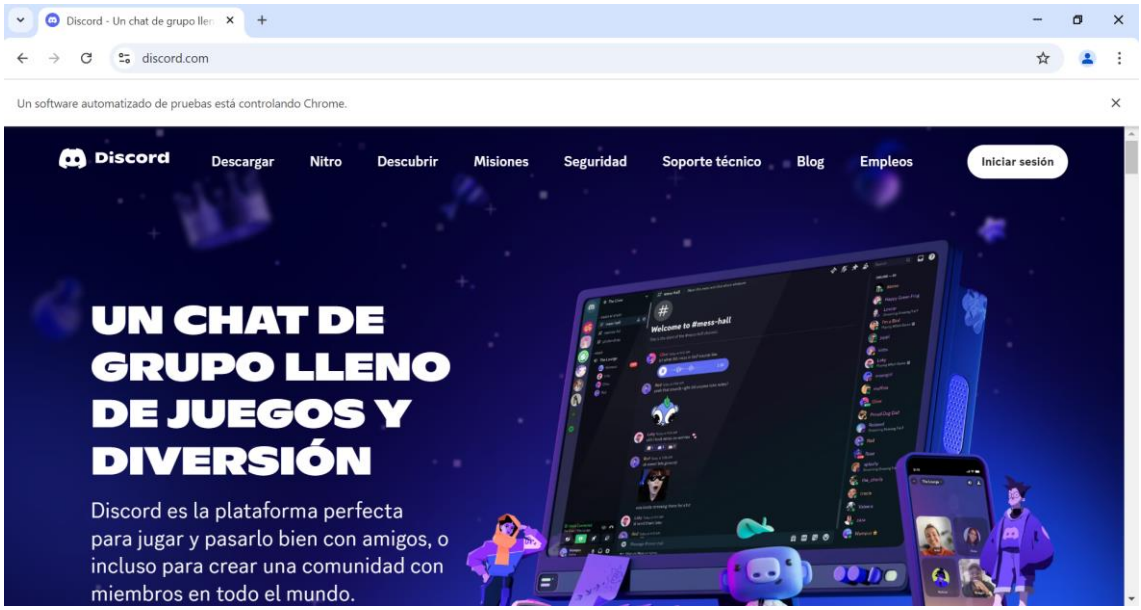


Figura 19 – 3.5. Discord sin variable de entorno Language

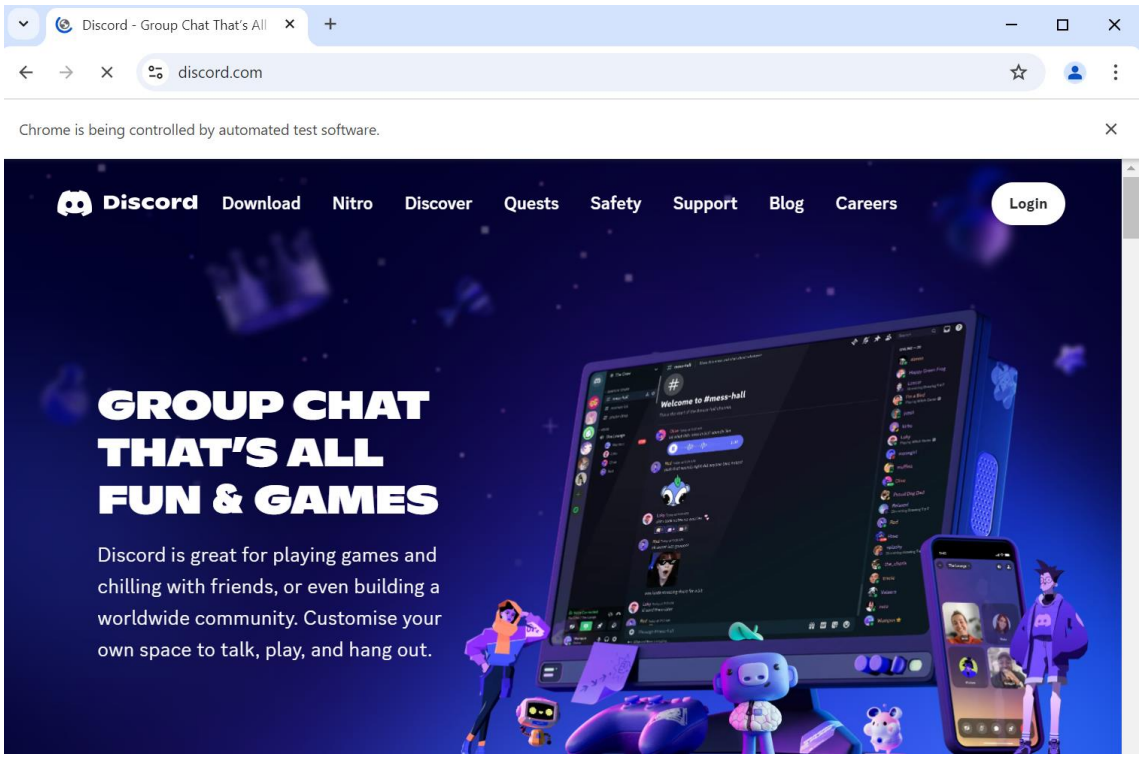
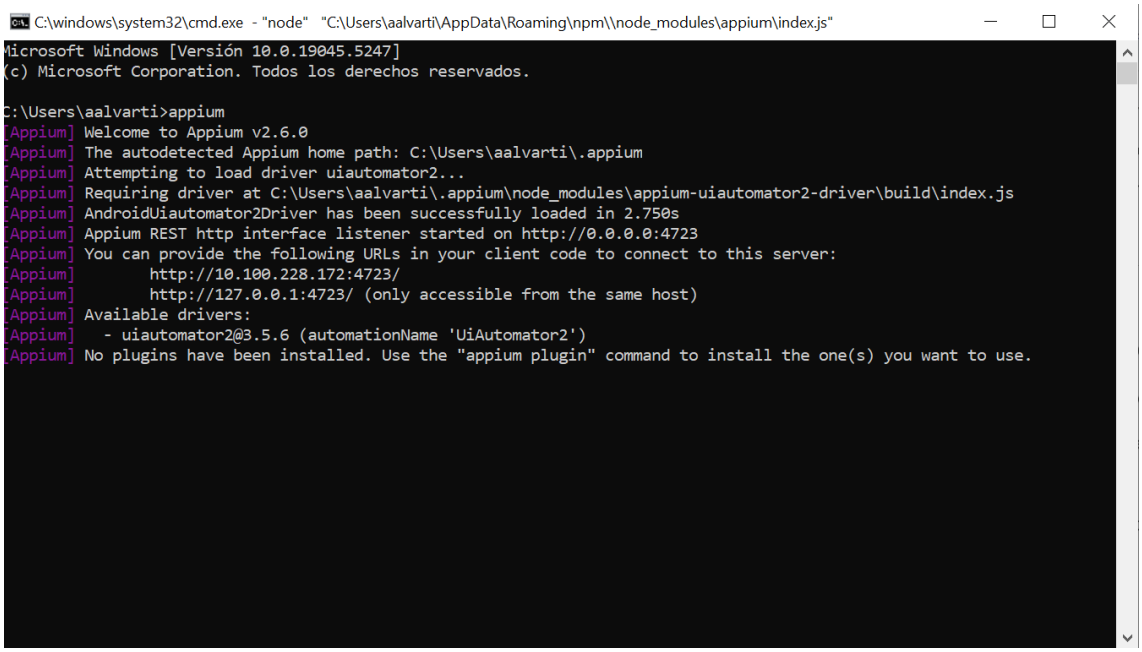


Figura 20 – 3.6. Discord con variable de entorno Language en-EN

De la misma manera se puede ejecutar el mismo test para su aplicación móvil. Para eso primero se necesita tener ejecutando Appium en el dispositivo en el que se van a hacer las pruebas, en este caso se realizarán en un emulador, por lo que Appium se ejecutara desde la terminal.



Con Appium ejecutando, se abre el emulador y se determinan las nuevas variables de entorno de la prueba. Para ejecutar las pruebas en móvil las variables serían las siguientes:

- Platform: Android
- AppActivity: com.discord.main.MainDefault
- AppIdentifier: com.discord

- Udid: emulator-5554

Como podemos observar, la plataforma ahora es Android, y en lugar de declarar la variable Application, declaramos AppActivity y AppIdentifier que nos proporcionan la ubicación de la app que vamos a probar. Por último, como estamos utilizando un emulador debemos declarar la variable Udid.

Como Appium utiliza el mismo protocolo que WebDriver, el proceso es el mismo.

Figura 21 – 3.7. Discord en móvil

Para las pruebas móviles, también se pueden utilizar las variables de entorno comunes para ambas plataformas, como Languaje y Provider. Si añadiéramos la variable de Language = fr-FR, tendríamos el mismo resultado pero con el sistema en francés (Se cambia el idioma del móvil).

Figura 22 – 3.8. Discord en móvil con variable de entorno Language fr-FR

Por último, otra variable de entorno común a ambas plataformas es la variable Provider, que permite ejecutar las pruebas de forma local o en la nube. Para ejecutar cualquiera de los ejemplos expuestos anteriormente en la nube, solo hace falta agregar la variable de entorno "Provider = SauceLabs" y agregar el usuario y el accessToken a las variables de entorno. Si se quisiera ejecutar el primer ejemplo en la nube las variables necesarias serían:

- Platform: Web
- Application: Discord
- Provider: SauceLabs
- User: xxx
- AccessToken: xxx

Con esto establecido, se puede lanzar el test. El test aparece de la siguiente forma en la página principal de SauceLabs

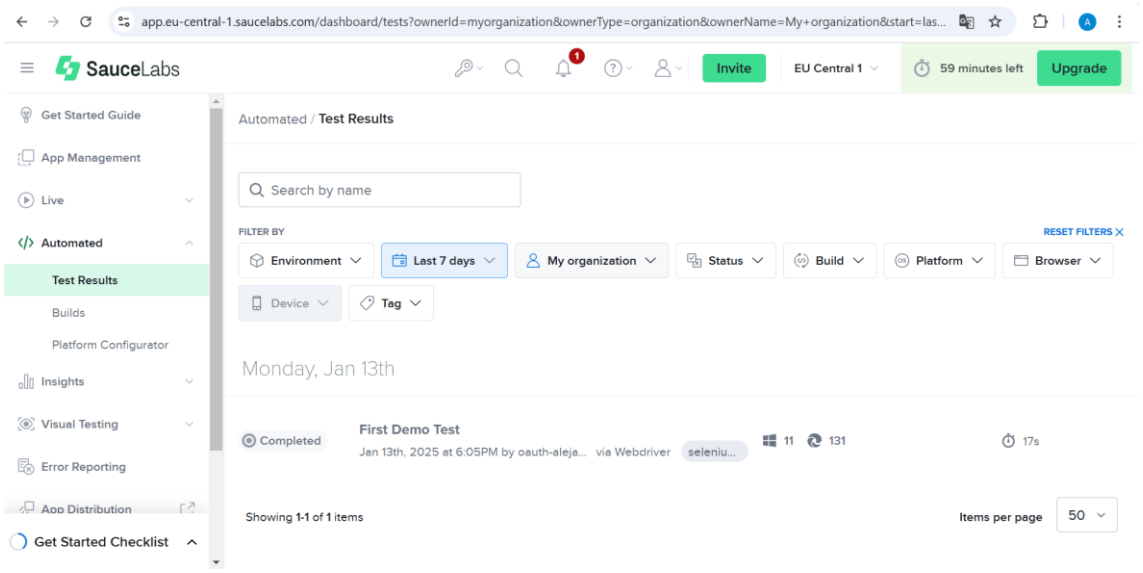


Figura 23 – 3.9. Página principal de SauceLabs

En SauceLabs podemos observar una gran cantidad de detalles sobre la ejecución de la prueba. Podemos ver la duración, las variables de entorno que se utilizaron, el dispositivo, el sistema operativo, se puede ver un video de la ejecución, screenshots, logs, entre otros.

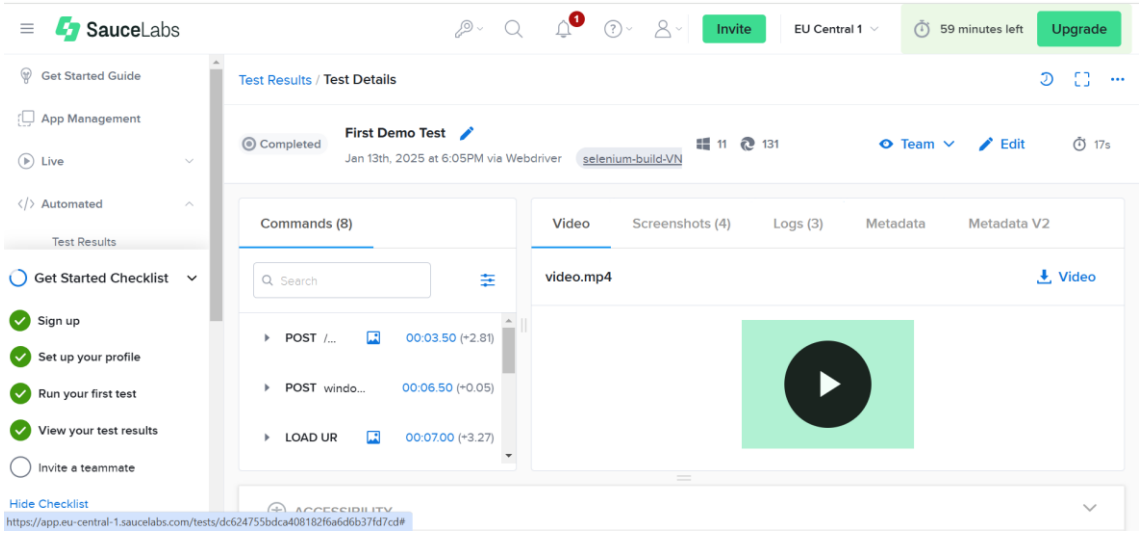


Figura 24 – 3.10. Prueba ejecutada en SauceLabs

En conclusión, el conjunto de pruebas antes descritas hacen una cobertura general de las funcionalidades del framework. Las combinaciones descritas de todas las variables de entorno desarrolladas, son solo un ejemplo. Con las variables actuales se cuentan más de 100 resultados distintos que se pueden obtener con el framework, lo que proporciona un gran abanico para la comprobación de calidad de cualquier proyecto en web y móvil.

4. Resultados y conclusiones

Todos los días sale al mercados nuevos productos software o actualizaciones de productos ya existentes. En el año 2013 se descargaron más de 800 millones de aplicaciones [87], lo que demuestra lo presente que están dichas aplicaciones software en la vida cotidiana. Esta dependencia destaca la responsabilidad que tienen las empresas y programadores de desarrollar productos que cumplan con los estándares de calidad establecidos.

En este Trabajo de Fin de Grado se ha llevado a cabo el desarrollo de un framework de automatización de pruebas web y móviles, con el fin de simplificar el proceso de mejora de calidad de los productos software. Esto se logró mediante la reutilización de código y herramientas que permitan la participación de todos los miembros del equipo, sean técnicos o no. Durante su desarrollo se presentaron diferentes desafíos, que fueron resueltos de la manera que resultó más apropiada.

En primer lugar, la utilización de herramientas como Selenium y Appium en un mismo framework era una idea compleja, y no se poseían ejemplos de desarrollos anteriores que pudieran servir como guía. Normalmente, es común ver este tipo de herramientas trabajando en entornos diferentes, ya que cada una posee configuración y dependencias específicas.

Investigando las dos herramientas se descubrió que para seguir los estándares establecidos por W3C, Appium decidió utilizar la mismas API desarrollada por Selenium. Esto fue un factor decisivo en la implementación del framework, ya que, compartir esta tecnología hacía que muchas funcionalidades de automatización fuera comunes a ambas herramientas, permitiendo la creación de test que sirvieran para pruebas web y móviles.

Una vez desarrolladas las funcionalidades básicas del framework, se pasó a implementar funcionalidades que hicieran el framework más asequible para los usuarios no técnicos del equipo. Para esto se incluyeron herramientas como Cucumber y Allure Reports. La implementación de Cucumber, implicó una reestructuración de las pruebas y el framework en general, ya que este estaba pensado para ejecutarse solo con Junit. Para la utilización de Cucucmber se crearon nuevas clases para guardar los métodos necesarios y estas se anclaron a los archivos de los escenarios.

La implementación de las pruebas del framework, en este punto, se encontraba en un buen nivel, en el contexto de usabilidad, pero el resultado de las pruebas aún se encontraba en un nivel bastante técnico, ya que estas se visualizaban en la terminal. Para esto se decidió utilizar la herramientas de reportes de Allure Reports, que permite la visualización de los resultados en una interfaz más amigable. Se agregaron los detalles de la ejecución para completar la información de las pruebas lanzadas.

El resultado ha sido un framework capaz de lanzar pruebas en entorno web y móvil, reutilizando código y combinando tecnologías para crear una herramientas que sea utilizable por miembros técnicos y no técnicos del equipo. Esta herramienta además de ahorrar costes, técnicos y económicos, agiliza el proceso de comprobación de calidad y el desarrollo general de cualquier proyecto.

5. Análisis de Impacto

El desarrollo de este framework ha significado la combinación de dos tecnologías de automatización de pruebas en una sola herramienta, lo que conlleva un avance significativo en varios aspectos.

Uno de los impactos más destacables de esta herramienta es la optimización de recursos, ya que con ella las empresas son capaces de ahorrar la infraestructura necesaria para realizar pruebas en distintos dispositivos y en distintas configuraciones. También facilita el trabajo de varios miembros del equipo, desde desarrolladores hasta testers, al permitir la ejecución de pruebas en aplicaciones web y móviles con un sola herramienta.

El 25 de septiembre del año 2015, los líderes de la Asamblea General de las Naciones Unidas establecieron un conjunto de objetivos para lograr un futuro más sostenible. En total se desarrollaron 17 objetivos, que se conocen como los Objetivos de Desarrollo Sostenible y forman parte de la nueva agenda 2030.

Figura 25 - 5.1. Objetivos de Desarrollo Sostenible

El desarrollo de este framework contribuye a 3 de estos objetivos. El primero es el obejtivo número 4: Educación de calidad. Viéndolo desde un punto de vista educativo, el framework puede ser una herramienta de formación para personas que buscan adentrarse en el área de testing, ya que al ser diseñado para ser utilizado por personas sin experiencia en programación, se convierte en la herramienta introductoria perfecta para personas sin experiencia en el área.

El segundo objetivo es el número 9: Industria, Innovación e Infraestructura. La implementación de este framework, que combina la utilización de diferentes herramientas de automatización y de visualización contribuye a la innovación tecnologíca del área del testing, fomenta la automatización eficiente con la reutilización de código y refuerza la modernización de la infraestructura de proyectos al realizar pruebas en la nube.

Por último, el framework también aporta al objetivo número 12: Producción y Consumo Responsables. Este proyecto colabora indirectamente a este objetivo al eliminar la necesidad de infraestructura física para la realización de pruebas en diferentes dispositivos. Como el framework es capaz de ejecutar pruebas en la nube, disminuye el consumo enérgico y la huella de carbono asociada a las operaciones de hardware local.

En conclusión, este proyecto no solo cumple con la función de mejorar el proceso de comprobación de calidad de proyectos, sino que también contribuye con varios de los Objetivos de Desarrollo Sostenible, demostrando el impacto que puede tener su implementación en los procesos de las compañías.

6. Bibliografía

[1] Real Academia Española, *Diccionario de la lengua española*, 23.ª ed. [En línea]. Disponible en: https://dle.rae.es. [Último acceso: 27-nov-2024].

[2] International Organization for Standardization, *UNE-EN-ISO 9000:2005. Sistemas de gestión de la calidad – Fundamentos y vocabulario*, Madrid: AENOR, 2005.

[3] International Organization for Standardization, *ISO 9000:2000 – Quality management systems – Fundamentals and vocabulary*. Geneva, Switzerland: ISO, 2000.

[4] "Concepto de calidad," *Blogadmi2*, 2010. [Online]. Available: https://blogadmi2.wordpress.com/wp-content/uploads/2010/04/sovca_-calidad.pdf. [Accessed: 27-Nov-2024].

[5] "Concepto de calidad," *Blogadmi2*, 2010. [Online]. Available: https://blogadmi2.wordpress.com/wp-content/uploads/2010/04/sovca_-calidad.pdf. [Accessed: 27-Nov-2024].

[6] R. Sasso, "Medir la calidad del 'software' antes de usarlo," *Club de Investigación Tecnológica*, 22-Jul-2021. [Online]. Available: https://www.clubdeinvestigacion.com/medir-la-calidad-del-software-antes-de-usarlo/. [Accessed: 27-Nov-2024].

[7] J. Muñoz Machado, *Gestión de calidad: Conceptos y herramientas aplicadas*. Capítulo 4. UNTREF Virtual, [Online]. Available: https://materiales.untrefvirtual.edu.ar/documentos_extras/01012_gestion_de_calidad/munoz_machado_cap4.pdf. [Accessed: 27-Nov-2024].

[8] Edelman, "2024 Edelman Trust Barometer: Supplemental Report Insights for Tech," [Online]. Available: https://www.edelman.com/sites/g/files/aatuss191/files/2024-03/2024%20Edelman%20Trust%20Barometer%20Supplemental%20Report%20Insights%20for%20Tech.pdf. [Accessed: 27-Dec-2024].

[9] M. G. Piattini, F. O. García, e I. Caballero, *Calidad de Sistemas Informáticos*, 1.ª ed., Alfaomega Grupo Editor, México, 2007. ISBN: 978-970-15-1267-8.

[10] International Organization for Standardization, *ISO/IEC 9126-1:2001 - Software Engineering - Product Quality - Part 1: Quality Model*. Geneva, Switzerland: ISO, 2001.

[11] G. J. Myers, T. Badgett, y C. Sandler, *The Art of Software Testing*, 3ª ed. Hoboken, NJ: John Wiley & Sons, Inc., 2012.

[12] ISTQB, "International Software Testing Qualifications Board," [Online]. Available: https://www.istqb.org. [Accessed: 27-Nov-2024].

[13] ISTQB, "ISTQB Glossary of Testing Terms," [Online]. Available: https://www.istqb.org/. [Accessed: 27-Nov-2024].

[14] A. Sánchez y N. Martínez, "Diferencia Entre Error, Bug, y Falla," *Diario Bug*, 16 de septiembre de 2021. [En línea]. Disponible en: https://diariobug.com/error-bug-falla/. [Accedido: 28-nov-2024].

[15] N. G. Leveson y C. S. Turner, "An Investigation of the Therac-25 Accidents," *IEEE Computer*, vol. 26, no. 7, pp. 18-41, 1993. [En línea]. Disponible en:

https://www.cs.columbia.edu/~junfeng/08fa-e6998/sched/readings/therac25.pdf. [Accedido: 28-nov-2024].

[16] J. L. Lions et al., "ARIANE 5 Flight 501 Failure Report by the Inquiry Board," European Space Agency, París, Francia, 1996. [En línea]. Disponible en: http://sunnyday.mit.edu/nasa-class/Ariane5-report.html. [Accedido: 28-nov-2024].

[17] Ethiopian Accident Investigation Bureau, "Aircraft Accident Investigation Report: B737-8 (MAX) Registered ET-AVJ," Mar. 2022. [En línea]. Disponible en: https://bea.aero/fileadmin/user_upload/ET_302__B737-8MAX_ACCIDENT_FINAL_REPORT.pdf. [Accedido: 2-dic-2024].

[18] G. Myers, *The Art of Software Testing*, 3ª ed., John Wiley & Sons, 2011.

[19] R. S. Pressman, *Ingeniería del Software: Un Enfoque Práctico*, 6ª ed., McGraw Hill, 2005.

[20] G. D. Everett y R. McLeod, *Software Testing: Testing Across the Entire Software Development Life Cycle*, Wiley, 2007.

[21] V. Pareto, *Cours d'économie politique*, vol. 1, Lausanne, Switzerland: F. Rouge, 1896.

[22] M. G. Piattini, F. O. García, y I. Caballero, *Calidad de Sistemas Informáticos*, 1ª ed., México: Alfaomega Grupo Editor, 2007.

[23] International Organization for Standardization, *ISO/IEC 25010:2011 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. Geneva, Switzerland: ISO, 2011.

[24] A. Ramírez, "El software: un producto con calidad," *Revista Facultad de Ingeniería Universidad de Antioquia*, no. 52, 2009. [En línea]. Disponible en: http://www.scielo.org.co/scielo.php?pid=S1692-33242009000300004&script=sci_arttext. [Accedido: 03-dic-2024]

[25] PMO Informática, "Tipos de pruebas no funcionales," *PMO Informática*, 20 de julio de 2016. [En línea]. Disponible en: https://www.pmoinformatica.com/2016/07/tipos-pruebas-no-funcionales.html. [Accedido: 28-nov-2024].

[26] L. G. Hayes, "Evolution of Automated Software Testing," Automated Testing Institute. [Online]. Available: http://www.automatedtestinginstitute.com. [Accessed: Oct. 2013].

[27] G. Bath and J. McKay, *The Software Test Engineer's Handbook: A Study Guide for the ISTQB Test Analyst and Technical Test Analyst Advanced Level Certificates*, 2nd ed., California, 2008.

[28] E. Dustin, J. Rashka, and J. Paul, *Automated Software Testing: Introduction, Management, and Performance*, New York, 2008.

[29] L. F. Giraldo, "Automatización de pruebas de software: una revisión," *Revista de Ciencias*, vol. 18, no. 2, pp. 45-54, 2014. [Online]. Available: https://revistas.univalle.edu/index.php/ciencias/article/view/748/1203

[30] E. Serna M., R. Martínez M., and P. Tamayo O., "Una revisión a la realidad de la automatización de las pruebas del software," *Computación y Sistemas*, vol. 23, no. 1, pp. 1-11, 2019. [Online]. Available: https://www.scielo.org.mx/scielo.php?pid=S1405-55462019000100169&script=sci_arttext

[31] "Las 12 mejores herramientas de pruebas de automatización (2024)," Guru99. [En línea]. Disponible: https://www.guru99.com/es/best-automation-testing-tools.html. [Accedido: 10-dic-2024].

[32] "20 Mejores Herramientas de Automatización de Pruebas 2024," ClickUp. [En línea]. Disponible: https://clickup.com/es-ES/blog/228197/herramientas-de-automatizacion-de-pruebas. [Accedido: 10-dic-2024].

[33] "Introducción a la herramienta de automatización Tricentis TOSCA," [En línea]. Disponible: https://spa.myservername.com/introduction-tricentis-tosca-automation-tool. [Accedido: 10-dic-2024].

[34] Tricentis, "Test automation," [En línea]. Disponible: https://www.tricentis.com/learn/test-automation. [Accedido: 10-dic-2024].

[35] Codes Code, "Pruebas de automatización a gran escala utilizando Cypress," [En línea]. Disponible: https://www.codescode.com/implementing-largescale-automation-testing-with-cypress.html. [Accedido: 10-dic-2024].

[36] Itequia, "Cypress: la herramienta que automatiza tus pruebas y garantiza la calidad de tus proyectos," [En línea]. Disponible: https://itequia.com/es/cypress-la-herramienta-que-automatiza-tus-pruebas-y-garantiza-la-calidad-de-tus-proyectos/. [Accedido: 10-dic-2024].

[37] Global Business IT, "Herramienta de Automatización Test Complete," [En línea]. Disponible: https://gbitcorp.com/blog/posts/test-complete/. [Accedido: 13-dic-2024].

[38] SmartBear Software, "TestComplete Test Object Model," [En línea]. Disponible: https://support.smartbear.com/testcomplete/docs/tutorials/getting-started/intro/object-model.html. [Accedido: 13-dic-2024].

[39] Katalon, "About Katalon Studio," [En línea]. Disponible: https://docs.katalon.com/katalon-studio/about-katalon-studio. [Accedido: 13-dic-2024].

[40] J. Pérez y M. Gómez, "Sistema de generación automática de scripts de ejecución para pruebas unitarias en aplicaciones web," *Revista Politécnica*, vol. 10, no. 2, pp. 45-56, 2014.

[41] ZAPTEST, "Una guía completa para la automatización de pruebas de software," [En línea]. Disponible: https://www.zaptest.com/es/una-guia-completa-para-la-automatizacion-de-pruebas-de-software. [Accedido: 13-dic-2024].

[42] Selenium, [En línea]. Disponible: https://www.seleniumhq.org/. [Accedido: 13-dic-2024].

[43] MachineNet, "How to Write Test Scripts," [En línea]. Disponible: https://www.machinet.net/tutorial-es/how-to-write-test-scripts. [Accedido: 13-dic-2024].

[44] Tutorials Point, "QTP - Overview," [En línea]. Disponible: https://www.tutorialspoint.com/qtp/qtp_overview.htm. [Accedido: 13-dic-2024].

[45] Cucumber, "Cucumber: Behavior Driven Development for All," [En línea]. Disponible: https://cucumber.io/. [Accedido: 13-dic-2024].

[46] IBM, "IBM Software," [En línea]. Disponible: https://www.ibm.com/software. [Accedido: 13-dic-2024].

[47] QAMinds Lab, "Reglas para crear scripts de automatización," [En línea]. Disponible: https://qamindslab.com/reglas-para-crear-scripts-de-automatizacion. [Accedido: 13-dic-2024].

[48] Abstracta, "Ambientes de Prueba e Infraestructura: Guía de Testing Continuo," [En línea]. Disponible: https://es.abstracta.us/recursos/guia-testing-continuo/ambientes-prueba-infraestructura. [Accedido: 13-dic-2024].

[49] PMO Informática, "Ambientes de Pruebas Integrales de Software: Buenas Prácticas," [En línea]. Disponible: https://www.pmoinformatica.com/2012/09/pruebas-software-ambientes.html. [Accedido: 13-dic-2024].

[50] AGESIC, "Guía para la Definición de Ambientes," [En línea]. Disponible: https://calidad-software.agesic.gub.uy/MCS-core/guidances/guidelines/guia_para_la_definicion_de_ambientes_3719FCBF.html. [Accedido: 13-dic-2024].

[51] Selenium, "Overview - Selenium Documentation," [En línea]. Disponible: https://www.selenium.dev/documentation/overview/. [Accedido: 13-dic-2024].

[52] Selenium, "Selenium components," [En línea]. Disponible: https://www.selenium.dev/documentation/overview/components/. [Accedido: 13-dic-2024].

[53] Selenium, "WebDriver - Selenium Documentation," [En línea]. Disponible: https://www.selenium.dev/documentation/webdriver/. [Accedido: 13-dic-2024].

[54] Selenium, "Getting Started - Selenium WebDriver Documentation," [En línea]. Disponible: https://www.selenium.dev/documentation/webdriver/getting_started/. [Accedido: 13-dic-2024].

[55] Selenium, "Driver Sessions," [En línea]. Disponible: https://www.selenium.dev/documentation/webdriver/drivers/. [Accedido: 13-dic-2024].

[56] Selenium, "Browser Options," [En línea]. Disponible: https://www.selenium.dev/documentation/webdriver/drivers/options/. [Accedido: 13-dic-2024].

[60] Selenium, "Interacting with web elements," [En línea]. Disponible: https://www.selenium.dev/documentation/webdriver/elements/interactions/. [Accedido: 13-dic-2024].

[61] Selenium, "Information about web elements," [En línea]. Disponible: https://www.selenium.dev/documentation/webdriver/elements/information/. [Accedido: 13-dic-2024].

[62] Selenium, "WebDriver Interactions: Navigation," [En línea]. Disponible: https://www.selenium.dev/documentation/webdriver/interactions/navigation/. [Accedido: 13-dic-2024].

[64] Selenium, "WebDriver Interactions: Cookies," [En línea]. Disponible: https://www.selenium.dev/documentation/webdriver/interactions/cookies/. [Accedido: 13-dic-2024].

[65] Selenium, "WebDriver Interactions: Print Page," [En línea]. Disponible: https://www.selenium.dev/documentation/webdriver/interactions/print_page/. [Accedido: 13-dic-2024].

[66] Selenium, "WebDriver Interactions: Windows," [En línea]. Disponible: https://www.selenium.dev/documentation/webdriver/interactions/windows/. [Accedido: 13-dic-2024].

[67] Selenium, "WebDriver Actions API," [En línea]. Disponible: https://www.selenium.dev/documentation/webdriver/actions_api/. [Accedido: 13-dic-2024].

[68] Selenium, "Keyboard actions," [En línea]. Disponible: https://www.selenium.dev/documentation/webdriver/actions_api/keyboard/. [Accedido: 13-dic-2024].

[69] Selenium, "Mouse actions," [En línea]. Disponible: https://www.selenium.dev/documentation/webdriver/actions_api/mouse/. [Accedido: 13-dic-2024].

[70] Selenium, "Scroll wheel actions," [En línea]. Disponible: https://www.selenium.dev/documentation/webdriver/actions_api/wheel/. [Accedido: 13-dic-2024].

[71] Appium, "Appium in a Nutshell," [En línea]. Disponible: https://appium.io/docs/en/latest/intro/. [Accedido: 13-dic-2024].

[72] Appium, "Introduction to Appium," *Appium Documentation*. [Online]. Available: https://appium.io/docs/en/latest/intro/appium/. [Accessed: 19-Dec-2024].

[73] Appium, "Drivers Introduction," *Appium Documentation*. [Online]. Available: https://appium.io/docs/en/latest/intro/drivers/. [Accessed: 19-Dec-2024].

[74] Allure Report, "Documentation," *Allure Report*. [Online]. Available: https://allurereport.org/docs/. [Accessed: 19-Dec-2024].

[75] Allure Report, "Features Overview," *Allure Report*. [Online]. Available: https://allurereport.org/docs/features-overview/. [Accessed: 19-Dec-2024].

[76] SauceLabs, "Products," *SauceLabs*. [Online]. Available: https://saucelabs.com/products. [Accessed: 19-Dec-2024].

[77] W. B. Kes, C. Fox, and B. A. Nejmeh, *Software Engineering in the UNIX/C Environment*. Englewood Cliffs, NJ, USA: Prentice Hall, 1991.

[78] F. J. García-Peñalvo, A. García-Holgado, and A. Vázquez-Ingelmo, "Introducción a la Ingeniería del Software," in *Recursos docentes de la asignatura Ingeniería de Software I. Grado en Ingeniería Informática. Curso 2020-2021*, F. J. García-Peñalvo, A. García-Holgado, and A. Vázquez-Ingelmo, Eds. Salamanca, España: Grupo GRIAL, Universidad de Salamanca, 2021. [Online]. Available: https://bit.ly/2WZlfWt. doi: 10.5281/zenodo.4399270.

[79] EVOTIC, "Ciclo de vida del software, etapas y modelos," [En línea]. Disponible: https://evotic.es/software-a-medida/ciclo-de-vida-del-software/. [Accedido: 26-dic-2024].

[80] J. F. Smart, *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Shelter Island, NY, USA: Manning Publications, 2015. [Online]. Available:

https://www.google.es/books/edition/BDD_in_Action/2TkzEAAAQBAJ?hl=es&gbpv=0.

[81] Cucumber.io, "Behavior-Driven Development (BDD)," [Online]. Available: https://cucumber.io/docs/bdd/. [Accessed: 26-Dec-2024].

[82] Cucumber, "Documentation," [Online]. Available: https://cucumber.io/docs/. [Accessed: 27-Dec-2024].

[83] Cucumber, "Step Definitions," *Cucumber Documentation*, [Online]. Available: https://cucumber.io/docs/cucumber/step-definitions. [Accessed: 27-Dec-2024].

[84] Cucumber, "API," *Cucumber Documentation*, [Online]. Available: https://cucumber.io/docs/cucumber/api. [Accessed: 27-Dec-2024].

[85] Cucumber, "Gherkin Reference," *Cucumber Documentation*, [Online]. Available: https://cucumber.io/docs/gherkin/reference. [Accessed: 27-Dec-2024].

[86] P. Hoyos, "QUÉ ES EL TESTING DE SOFTWARE Y POR QUÉ ES TAN IMPORTANTE EN EL DESARROLLO DE SOFTWARE," Pacifitic. [En línea]. Disponible: https://pacifitic.org/que-es-el-testing-de-software-y-por-que-es-tan-importante-en-el-desarrollo-de-software/. [Accedido: 13-ene-2025].

[87] King of App, "Estadísticas sobre descargas de aplicaciones y tendencias de uso 2022," *King of App*, 2022. [En línea]. Disponible en: https://kingofapp.com/es/estadisticas-sobre-descargas-de-aplicaciones-y-tendencias-de-uso-2022/. [Último acceso: 14-ene-2025].

[89] Naciones Unidas, "Objetivos de Desarrollo Sostenible," *Naciones Unidas - Desarrollo Sostenible*, 2023. [En línea]. Disponible en: https://www.un.org/sustainabledevelopment/es/objetivos-de-desarrollo-sostenible/. [Último acceso: 14-ene-2025].

Anexos

AllowedResolutions.yaml

```yaml
browsers:
  chrome:
    resolutions:
      - 320x568
      - 375x667
      - 768x1024
      - 800x600
      - 1024x1366
      - 1024x768
      - 1140x900
      - 1152x864
      - 1280x768
      - 1280x800
      - 1280x960
      - 1280x1024
      - 1400x1050
      - 1440x900
      - 1600x1200
      - 1680x1050
      - 1920x1200
      - 1920x1200
      - 2560x1600
  firefox:
    resolutions:
      - 800x600
      - 1024x1366
      - 1024x768
      - 1140x900
      - 1152x864
      - 1280x768
      - 1280x800
      - 1280x960
      - 1280x1024
      - 1400x1050
      - 1440x900
      - 1600x1200
      - 1680x1050
      - 1920x1200
      - 1920x1200
      - 2560x1600
```

```
safari:
  resolutions:
    - 800x600
    - 1024x1366
    - 1024x768
    - 1140x900
    - 1152x864
    - 1280x768
    - 1280x800
    - 1280x960
    - 1280x1024
    - 1400x1050
    - 1440x900
    - 1600x1200
    - 1680x1050
    - 1920x1200
    - 1920x1200
    - 2560x1600

edge:
  resolutions:
    - 800x600
    - 1024x1366
    - 1024x768
    - 1140x900
    - 1152x864
    - 1280x768
    - 1280x800
    - 1280x960
    - 1280x1024
    - 1400x1050
    - 1440x900
    - 1600x1200
    - 1680x1050
    - 1920x1200
    - 1920x1200
    - 2560x1600
```

AndroidConfiguration.yaml

```
capabilitiesAndroid:
  automationName: 'UiAutomator2'
  noReset: 'true'
  newCommandTimeout: "90"
```

WebConfiguration.yaml

```
url:
  mrc: 'https://www.marca.com/'
  disc: 'https://discord.com/'
```

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Tue Jan 14 22:47:31 CET 2025
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)