Proceso de automatización de pruebas QA para el Sistema de Información Financiero (SIF) en la División de Tecnologías de la Información y la Comunicación (DTIC)

Juan Diego Esteban Parra y Julián Andrey Rodríguez Romero

Trabajo de Grado para optar al Título de Ingeniero de Sistemas

Director

Urbano Eliécer Gómez Prada

Doctor en Tecnología Educativa

Tutor

Danny Felipe Vergel Paba

Especialista en Gerencia de Proyectos

Universidad Industrial de Santander

Facultad de Ingenierías Fisicomecánicas

Escuela de Ingeniería de Sistemas e Informática

Ingeniería de Sistemas

Bucaramanga

2026

Dedicatoria

A mi mamá, por ser mi base, mi impulso y mi mayor fan. Por su amor incondicional, su presencia constante y por estar siempre de la manera más genuina.

A Miss Moon, por saber estar en los momentos justos; por ser refugio y calma cuando más lo necesitaba. Por estar sin preguntas y quedarse sin condiciones.

A Brayan, por quedarse en lo cotidiano y volverlo especial; por la manera en que supo estar. Porque, sin decir mucho, lo cambió todo.

A mis amigos, Diego y Linares, por compartir esta etapa conmigo, por las aventuras, las risas y por hacer más livianos los días largos.

Julián Andrey Rodríguez Romero

A mi madre Eliut y a mi padre Wilson quienes se merecen todo el orgullo de haberme formado como ser humano para poder superar las difíciles etapas no solo académicas sino de la vida, quienes me enseñaron las bases para que hoy estoy en el camino que he decidido vivir.

A mi familia, hermanos, tíos, tías, nonas, primos, mis gatitos y quienes estuvieron presentes en cada momento facilitándome cada lucha interna y externa durante estos años universitarios.

A toda persona que me ha enseñado algo de valor para la vida y a cada amistad dentro o fuera de la universidad especialmente Marco, Juliana, Jhan, Lina, Daniel, Paola, a los kuni, los de la carrera y a mi compañero Julián por su esfuerzo y dedicación durante estos 2 años de TG.

A Yuliana quien ha llegado a mi vida para brindarme amor, quien me ha otorgado una nueva percepción de lo que es sentirse querido, gracias por tu oxitocina incondicional.

Juan Diego Esteban Parra

Agradecimientos

A la Universidad Industrial de Santander, por convertirse a lo largo de estos años en un lugar cercano y significativo, un segundo hogar donde se construyeron aprendizajes, sueños y metas. Gracias por la formación recibida, tanto en lo profesional como en lo humano, y por las experiencias compartidas y las personas que hicieron parte de este camino.

A la División de Tecnologías de la Información y la Comunicación, por hacer posible la realización de este trabajo de grado y por brindarnos la oportunidad de acercarnos al mundo profesional. Gracias por la confianza, el acompañamiento y el apoyo recibido durante el desarrollo de este proyecto.

Al equipo de QA, por ser un equipo humano y profesional excepcional. Gracias por el trabajo colaborativo, la disposición para ayudar, el intercambio de conocimientos y el ambiente de apoyo que hicieron de este proceso una experiencia de aprendizaje y crecimiento continuo.

A nuestro director, Urbano Gómez, por su paciencia, su guía constante y por impulsarnos siempre a ir más allá. Gracias por su mentoría, su retroalimentación oportuna y constructiva, y por su acompañamiento durante todo el desarrollo del proyecto; su apoyo fue clave para alcanzar este resultado.

A nuestro tutor, Danny Vergel, por su disponibilidad, su compromiso y por compartir su conocimiento de manera generosa. Gracias por su acompañamiento, sus aportes y por brindarnos las herramientas necesarias para fortalecer y consolidar este trabajo.

Julián Andrey Rodríguez Romero

Juan Diego Esteban Parra

Tabla de Contenido

Pág.

Introducción ... 15
1. Planteamiento del problema y Justificación ... 17
2. Objetivos ... 20
2.1 Objetivo General ... 20
2.2 Objetivos Específicos ... 20
3. Marco de Referencia ... 21
3.1 Calidad del Software ... 21
3.1.1 Pruebas funcionales ... 21
3.1.1.1 Prueba de humo ... 22
3.1.1.2 Prueba de regresión. ... 22
3.2 Metodología BDD ... 22
3.3 Automatización de Pruebas ... 23
3.4 Herramientas para la Automatización de Pruebas ... 23
3.4.1 Entornos de automatización ... 23
3.4.2 Herramientas de informes ... 24
3.4.3 Lenguajes y herramientas para escribir escenarios ... 24
3.5 Patrones de Pruebas Automatizadas ... 25
3.6 Metodología Ágil de Desarrollo Scrum ... 25
3.7 Control de Versiones para Pruebas Automatizadas ... 25
3.8 Historias de Usuario (HU) ... 26
3.8.1 Criterios de aceptación ... 26
3.8.2 Creación de casos de prueba a partir de la HU ... 26
4. Marco Metodológico ... 27
4.1 Fase 1: Evaluación de casos de prueba QA actuales y definición del proceso de automatización ... 27
4.2 Fase 2: Incursión de la metodología BDD en el ciclo de desarrollo del software ... 28
4.3 Fase 3: Diseño de la estructura y configuración del entorno para scripts automatizados ... 29
4.4 Fase 4: Evaluación de la automatización en comparación a lo manual ... 30
4.5 Fase 5: Síntesis y entregables ... 30
5. Implementación ... 31
5.1 Orientación del proceso de automatización de pruebas QA en la DTIC ... 31
5.1.1 Análisis de los casos de prueba manuales existentes en el módulo de proveedores ... 32
5.1.1.1 Revisión del alcance de los casos de prueba actuales. ... 33
5.1.1.2 Identificación de deficiencias en el diseño y ejecución.. ... 35
5.1.1.3 Priorización de casos relevantes para automatización.. ... 38

5.1.1.4 Recopilación de datos importantes en pruebas manuales.. 41
5.1.2 Evaluación de las herramientas utilizadas actualmente para pruebas QA 43
5.1.2.1 Identificación de las herramientas utilizadas actualmente y su propósito. 44
5.1.2.2 Análisis en las limitaciones de las herramientas actuales.. 46
5.1.2.3 Identificación de áreas de mejora en el uso de herramientas.. 50
5.1.3 Propuesta inicial del proceso de automatización 54
5.1.3.1 Definición del flujo de trabajo basado en hallazgos del análisis. 54
5.1.3.2 Identificación de métricas clave para medir el impacto de la automatización. 58
5.1.3.3 Propuesta preeliminar de integración con herramientas de automatización. 60
5.2 Documentación en la Wiki de la DTIC basada en la metodología BDD (Behavior-Driven Development) 63
5.2.1 Test-Driven Development (TDD) 64
5.2.1.1 Principios de TDD.. 64
5.2.1.2 Ventajas y desventajas de TDD.. 65
5.2.1.3 Ciclo iterativo de TDD 65
5.2.2 Behavior-Driven Development (BDD) 66
5.2.2.1 Lenguaje Gherkin 67
5.2.3 Principios de BDD 69
5.2.4 Ventajas y desventajas de BDD 71
5.2.5 Beneficios de la metodología BDD en la automatización 73
5.2.6 Comparación entre BDD y TDD 75
5.2.7 Estrategias para la integración de BDD en el ciclo de desarrollo ágil 77
5.2.8 Documentación final de BDD en la wiki de la DTIC 80
5.3 Estructura y diseño de las pruebas automatizadas 83
5.3.1 Selección de herramientas de automatización 83
5.3.1.1 Entornos de automatización. 84
5.3.1.2 Herramientas de generación de informes. 87
5.3.1.3 Frameworks BDD para la creación de casos de prueba. 90
5.3.2 Análisis de patrones de diseño para la automatización de pruebas 93
5.3.2.1 Descripción de los patrones evaluados.. 93
5.3.2.2 Evaluación de aplicabilidad para cada patrón. 96
5.3.2.3 Ventajas y desventajas de cada patrón. 98
5.3.2.4 Patrón seleccionado y justificación en el contexto de la DTIC. 101
5.3.3 Definición del proceso de automatización 102
5.3.3.1 Identificación de pruebas a automatizar. 103
5.3.3.2 Planificación de la automatización. 106
5.3.3.3 Diseño de la solución de automatización. 109
5.3.3.4 Implementación y desarrollo de script. 113

5.3.3.5 Ejecución de las pruebas automatizadas. 115
5.3.3.6 Análisis y validación de los resultados.. 116
5.3.3.7 Mantenimiento de los scripts y mejora continua.. 117
5.3.3.8 Diagrama general del proceso de automatización de pruebas 118
5.4 Pruebas del plan de automatización propuesto 119
5.4.1 Análisis y selección de casos de prueba seleccionados para automatizar 120
5.4.2 Diseño de los casos de prueba automatizados 121
5.4.3 Configuración técnica del entorno de automatización..122
5.4.3.1 Instalación y configuración de herramientas.. 122
5.4.3.2 Configuración del entorno de pruebas.. 125
5.4.4 Desarrollo y ejecución de los scripts de pruebas automatizadas 126
5.4.4.1 Estructura del código de automatización.. 126
5.4.4.2 Ejecución y validación de los scripts.. 129
5.4.4.3 Optimización, reutilización y mantenimiento del código. 132
5.4.5 Entregables finales del proceso de automatización 133
5.4.5.1 Publicación de los scripts automatizados en el repositorio.. 133
5.4.5.2 Estructura del framework de automatización.. 134
6. Análisis de resultados.. 136
6.1 Resultados de la ejecución de pruebas automatizadas..136
6.2 Comparación entre pruebas manuales y pruebas automatizadas 138
6.2.1 Análisis comparativo del tiempo de ejecución 138
6.2.2 Análisis comparativo de la cobertura de pruebas 139
6.2.3 Análisis comparativo de hallazgos y defectos 140
6.3 Análisis de reportes automatizados por tipo de proveedor 140
6.4 Gestión y validación de incidencias derivadas de la automatización 142
7. Conclusiones 144
8. Trabajo futuro 145
Referencias Bibliográficas 147

Lista de Tablas

Pág.

Tabla 1. Datos de ejecución de pruebas manuales por funcionalidad 41
Tabla 2. Criterios definidos para comparación y elección de framework de automatización 85
Tabla 3. Criterios definidos para comparación y elección de una herramienta de generación de informes 89
Tabla 4. Criterios definidos para la comparación y elección de un framework BDD para la creación de casos de prueba..91

Lista de Figuras

Pág.

Figura 1. Flujo inicial del proceso de automatización 64
Figura 2. Ciclo iterativo de TDD 67
Figura 3. Actividades realizadas en BDD 68
Figura 4. Pasos Gherkin 69
Figura 5. BDD y TDD combinadas 78
Figura 6. Integración de BDD en el ciclo de desarrollo ágil 81
Figura 7. Estructura en la Wiki de la guía teórica BDD 82
Figura 8. Estructura en la Wiki de la guía de adopción de BDD en el equipo de QA 83
Figura 9. Esquema del patrón Page Object Model (POM) 94
Figura 10. Esquema del patrón Screenplay 95
Figura 11. Esquema del patrón Command 96
Figura 12. Flujo de automatización de pruebas final 120
Figura 13. Vista general de la épica de proveedores 122
Figura 14. Estructura de los archivos .feature 123
Figura 15. Verificación de versión de Java por consola 124
Figura 16. Verificación de versión de Maven en consola 125
Figura 17. Sincronización del proyecto Maven en el IDE 126
Figura 18. Configuración del WebDriver para el navegador 126
Figura 19. Clase definida para navegar sitio web 127
Figura 20. Ejemplo de clase en Definitions con anotaciones Cucumber 129
Figura 21. Ejemplo de Page Object Model con selectores 129
Figura 22. Ejemplo de métodos reutilizables en Steps 130
Figura 23. Estructura de la clase Runner 130
Figura 24. Estructura de carpetas del proyecto en el IDE 131
Figura 25. Comandos de Maven para generar el reporte 132

Figura 26. Resumen general de un reporte Serenity 133
Figura 27. Ejemplo del reporte detallado para cada escenario 134
Figura 28. Implementación inicial del método de carga de archivos 135
Figura 29. Ejemplo de refactorización del método anterior 135
Figura 30**.** Repositorio de QA Automation en donde se consolidaron los scripts 137
Figura 31. Estructura final del framework de automatización de pruebas 138
Figura 32. Tiempo de ejecución de pruebas automatizadas para proveedor persona natural 140
Figura 33. Tiempo de ejecución de pruebas automatizadas para proveedor persona jurídica 140
Figura 34. Tiempo de ejecución de pruebas automatizadas para proveedor persona jurídica extranjera 140
Figura 35. Reporte automatizado de ejecución para proveedor persona natural 144
Figura 36. Reporte automatizado de ejecución para proveedor persona jurídica 144
Figura 37. Reporte automatizado de ejecución para proveedor persona jurídica extranjera 145
Figura 38. Historias de usuario relacionadas al rediseño de proveedores 146
Figura 39. Ejemplo de algunos issues reportados en la herramienta de gestión 146

Lista de Apéndices externos

Los apéndices están adjuntos y pueden visualizarse en la base de datos de la biblioteca UIS.

Apéndice A. Casos de prueba en formato Gherkin

Glosario

Automatización de pruebas: proceso de utilizar herramientas y scripts para ejecutar pruebas de software de manera automática, reduciendo la intervención manual y mejorando la eficiencia.

Behavior-Driven Development (BDD): metodología de desarrollo de software que se centra en la colaboración entre desarrolladores, testers y stakeholders no técnicos mediante el uso de un lenguaje común para describir el comportamiento esperado del sistema.

Casos de prueba: conjunto de condiciones o variables bajo las cuales se verifica si un sistema o software funciona correctamente según los requisitos.

Criterios de aceptación: condiciones que deben cumplirse para que una historia de usuario o funcionalidad se considere completada y aceptada.

Eficiencia: capacidad del proceso de pruebas automatizadas para optimizar tiempo, recursos y esfuerzo en comparación con las pruebas manuales.

Framework: conjunto de herramientas y bibliotecas que proporcionan una estructura estandarizada para el desarrollo y la automatización de pruebas.

Historia de usuario: descripción de funcionalidades o requerimientos desde la perspectiva del usuario final.

Integración Continua (CI): práctica de desarrollo que implica integrar y validar automáticamente los cambios en el código fuente para detectar errores de forma temprana.

Lenguaje Gherkin: sintaxis estructurada utilizada en BDD para describir escenarios de prueba mediante frases clave como "Dado", "Cuando" y "Entonces".

Patrón de diseño: solución general y reutilizable para problemas comunes en el diseño de sistemas, como el modelo Page Object Model (POM) y el patrón Screenplay.

Pruebas funcionales: validación de que las funcionalidades del software cumplen con los requisitos especificados, enfocándose en la entrada y salida de datos.

Pruebas manuales: proceso de verificar manualmente el funcionamiento del software sin el uso de herramientas automatizadas.

Pruebas de regresión: proceso de validar que los cambios recientes en el código no han afectado negativamente las funcionalidades existentes.

Reportes de ejecución: documentos o archivos generados después de ejecutar pruebas automatizadas que muestran los resultados obtenidos, incluyendo fallos y errores.

Scrum: metodología ágil de gestión de proyectos que divide el trabajo en ciclos llamados sprints, fomentando la colaboración y la entrega continua.

Selenium: herramienta de automatización de pruebas de software que permite interactuar con aplicaciones web de forma programada.

Serenity BDD: framework de pruebas que integra BDD con herramientas como Selenium, proporcionando reportes detallados y facilidad para escribir escenarios de prueba.

Resumen

Título: Proceso de automatización de pruebas QA para el Sistema de Información Financiero (SIF) en la División de Tecnologías de la Información y la Comunicación (DTIC)[1*]

Autores: Juan Diego Esteban Parra y Julián Andrey Rodríguez Romero[2**3**]

Palabras Clave: Automatización de pruebas, Pruebas funcionales, BBD, POM, Scrum

Descripción:

Este proyecto tiene como objetivo definir un proceso de automatización de pruebas de aseguramiento de calidad (QA) para el Sistema de Información Financiero (SIF) de la División de Tecnologías de la Información y la Comunicación (DTIC) de la Universidad Industrial de Santander. La automatización se plantea como una estrategia para optimizar la detección de defectos, mejorar la eficiencia del proceso de validación y garantizar la repetibilidad de las pruebas, en concordancia con los estándares institucionales de calidad. Para su desarrollo, se adoptó la metodología Behavior-Driven Development (BDD), integrada con el marco ágil Scrum, permitiendo construir escenarios de prueba a partir de historias de usuario claramente definidas y fortalecer la alineación entre los requerimientos funcionales y las validaciones automatizadas.

El proceso propuesto contempla la planificación de pruebas, la selección del stack tecnológico, el desarrollo de scripts bajo el patrón Page Object Model (POM) y la ejecución controlada de escenarios automatizados. Asimismo, se establecieron métricas como el tiempo de ejecución, la cobertura funcional y la detección de defectos, con el fin de comparar el enfoque manual frente al automatizado. Los resultados evidencian mejoras en eficiencia y calidad, consolidando una base metodológica sostenible para la implementación progresiva de pruebas automatizadas en el SIF y su posible extensión a otros sistemas institucionales.

[1*] Trabajo de Grado
[2**] Facultad de Fisicomecánicas. Escuela de Ingeniería de sistemas e informática. Director: Urbano Eliécer Gómez Prada, Ph.D. en Tecnología Educativa.
[3**] DTIC. Codirector: Danny Felipe Vergel Paba, Especialista en Gerencia de Proyectos.

Abstract

Title: QA Test Automation Process for the Financial Information System (SIF) in the Division of Information and Communication Technologies (DTIC) [4*]

Author(s): Juan Diego Esteban Parra and Julián Andrey Rodríguez Romero [5**6**]

Key Words: Test Automation, Functional Testing, BDD, POM, Scrum

Description:

This project aims to define a quality assurance (QA) test automation process for the Financial Information System (SIF) of the Information and Communication Technologies Division (DTIC) at the Universidad Industrial de Santander. Automation is proposed as a strategy to optimize defect detection, improve validation efficiency, and ensure test repeatability, in alignment with institutional quality standards. For its development, the Behavior-Driven Development (BDD) methodology was adopted and integrated with the Scrum agile framework, enabling the construction of test scenarios based on clearly defined user stories and strengthening the alignment between functional requirements and automated validations.

The proposed process includes test planning, selection of the technological stack, script development under the Page Object Model (POM) design pattern, and controlled execution of automated scenarios. Additionally, metrics such as execution time, functional coverage, and defect detection rate were established in order to compare the manual and automated approaches. The results demonstrate improvements in efficiency and quality, consolidating a sustainable methodological foundation for the progressive implementation of automated testing within the SIF and its potential extension to other institutional systems.

[4*] Thesis/Final Degree Project
[5**] Faculty of Physical-Mechanical Engineering. School of Systems and Computer Engineering. Advisor: Urbano Eliécer Gómez Prada, Ph.D. in Educational Technology.
[6**] DTIC. Co-Advisor: Danny Felipe Vergel Paba, Specialist in Project Management.

Introducción

La División de Tecnologías de la Información y la Comunicación (DTIC) de la Universidad Industrial de Santander (UIS) es responsable del desarrollo y mantenimiento de sistemas informáticos que optimizan los procesos de gestión y administración en la universidad, en el marco de una plataforma integral conocida como el Sistema de Información Administrativo (UISARD). Actualmente, uno de los proyectos prioritarios de la DTIC es la renovación del Sistema de Información Financiero (SIF), un subsistema clave de UISARD que permite una administración eficiente de los recursos financieros de la universidad, el cual tiene como propósito central, mejorar la calidad en los procesos financieros y el manejo de sus recursos, en beneficio de la comunidad universitaria (D. Vergel, comunicación personal, 8 de agosto de 2024).

El crecimiento continuo del SIF, dentro de UISARD, ha hecho evidente la necesidad de promover que cada módulo desarrollado funcione de manera confiable y cumpla con los estándares de calidad definidos por la DTIC. En este contexto, el equipo de Aseguramiento de la Calidad (QA) de la DTIC desempeña un rol esencial, ya que verifica que el software cumpla con los requisitos funcionales y responda a las necesidades de la comunidad universitaria. Hasta ahora, la estrategia de QA ha dependido en gran medida de pruebas manuales, una práctica que, aunque efectiva en ciertos casos, se ha vuelto insuficiente debido al crecimiento y a la complejidad estructural que presenta el SIF (D. Vergel, comunicación personal, 8 de agosto de 2024).

Por este motivo, fue necesario implementar un proceso de automatización de pruebas para disminuir el tiempo y los recursos destinados a brindar calidad en el proceso del software, en donde un sistema, componente o proceso debía asegurar que el producto cumpliera con los requisitos especificados y respondiera a las expectativas y necesidades del usuario (IEEE, 1991).

La automatización permitió al equipo de QA cubrir más escenarios de prueba en menor tiempo, reducir la dependencia de las pruebas manuales y facilitar la ejecución repetitiva de casos de prueba, asegurando que cada componente del SIF mantuviera su funcionalidad ante actualizaciones y cambios en el sistema. Con esta estrategia, el equipo de QA pudo responder a los desafíos de escalabilidad del SIF, permitiendo que el sistema creciera de manera sólida y confiable.

La metodología que sustentó esta automatización fue el desarrollo orientado por comportamiento, conocido como Behaviour-Driven Development (BDD), una evolución del desarrollo basado en pruebas o Test-Driven Development (TDD), y se considera una metodología ágil altamente eficiente para la creación de software.

BDD se basa en la automatización de pruebas y emplea herramientas y métodos de análisis avanzados que contribuyen a prevenir defectos y mejorar la calidad del software (Dookhun & Nagowah, 2019). Además, facilita la redacción de los casos de prueba en un lenguaje natural, lo cual permite que los requisitos del sistema sean claros y comprensibles para los involucrados, desde desarrolladores hasta personal no técnico.

Con esta metodología, el equipo de QA pudo ejecutar pruebas automatizadas sobre casos de prueba funcionales en la interfaz del SIF, asegurando su calidad y alineación con los objetivos del sistema.

En función a las ideas expresadas en los párrafos anteriores, se da claridad en que este proyecto buscó implementar un enfoque de pruebas automatizadas que fortaleciera el proceso de aseguramiento de calidad y mejorase la eficiencia en el desarrollo de software en la DTIC. El cumplimiento de los objetivos del proyecto se desarrolla principalmente en la Sección 5 – Implementación; sin embargo, una parte del último objetivo específico también se aborda y

complementa en la Sección 6 – Análisis de resultados. Aunque el propósito fue definir el proceso de automatización para el SIF, el diseño y ejecución de casos de prueba automatizados se enfocaron únicamente en el módulo de proveedores, optimizando así los recursos y asegurando que este módulo cumpliera con los estándares de calidad establecidos.

1. Planteamiento del problema y Justificación

El equipo de control de calidad (QA) de la División de Tecnologías de la Información y la Comunicación (DTIC) de la Universidad Industrial de Santander (UIS) es el encargado de velar por la calidad en el desarrollo de software. Sin embargo, actualmente sus actividades de prueba y de documentación se realizan de forma manual, lo que ha generado importantes desafíos en el cumplimiento eficiente de los procesos de aseguramiento de calidad.

Actualmente, el equipo de QA emplea herramientas básicas como Excel, ShareX y Taiga para apoyar la creación y ejecución de casos de prueba (Roa, s. f.). Esta dependencia de herramientas manuales no solo prolonga los tiempos de respuesta en cada actividad, sino que también pone de manifiesto un déficit en el diseño de los casos de prueba y su ejecución.

La falta de un enfoque estructurado y automatizado dificulta la creación de casos de prueba coherentes y la ejecución eficiente de los mismos, lo que se traduce en una limitada capacidad para cubrir todas las funcionalidades del software de manera integral. Lo anterior, se vuelve aún más desafiante debido a la complejidad del crecimiento de los sistemas de la universidad, particularmente en el Sistema de Información Financiero (SIF).

La ejecución manual de las pruebas no solo prolonga el ciclo de desarrollo de software, sino que también dificulta el cumplimiento de los estándares de calidad definidos por la DTIC,

tales como usabilidad, mantenibilidad y eficiencia (Roa, s. f.), que exigen un software confiable a las necesidades de la comunidad universitaria.

Además, la continua evolución y actualización del SIF, caracterizada por el desarrollo constante de nuevos módulos y su integración con el resto del sistema, demanda pruebas de regresión periódicas para ayudar a que las nuevas funcionalidades no afecten el correcto funcionamiento de las ya existentes.

Sin un enfoque de pruebas automatizadas, la tarea manual de probar un software se convierte en un proceso lento y poco sostenible, afectando la calidad del sistema y limitando la capacidad de respuesta del equipo de QA.

En este contexto ágil de la DTIC, donde la adaptabilidad y la rapidez son esenciales, surge la necesidad de implementar estrategias que permitan responder de forma continua a los cambios y nuevas demandas de los sistemas administrativos.

Este enfoque promueve el desarrollo iterativo, con entregas en ciclos cortos y constantes, facilitando ajustes y mejoras en cada fase (Agile Alliance, 2014). Por este motivo, la implementación de pruebas automatizadas aumentará la eficiencia del equipo de QA, permitiendo ejecutar un mayor volumen de pruebas en menos tiempo y detectar errores de manera oportuna.

Al mismo tiempo, la automatización facilitará el seguimiento detallado de cada prueba a través de informes y estadísticas, mejorando la trazabilidad de su ejecución y asegurando una entrega de software más rápida y confiable.

En definitiva, la automatización de pruebas se presenta como una solución estratégica para mejorar los procesos de aseguramiento de calidad en el SIF, permitiendo que el sistema crezca y se adapte a nuevas necesidades sin comprometer su estabilidad y rendimiento.

A su vez, este enfoque sienta las bases para que, en el futuro, la automatización pueda escalar a otros proyectos de la DTIC, creando una estructura de pruebas más robusta y adaptable a las demandas crecientes de la universidad.

2. Objetivos

2.1 Objetivo General

Definir el proceso para la automatización de pruebas QA centrado en casos de pruebas funcionales del módulo proveedores del Sistema de Información Financiero (SIF) soportado en la metodología BDD, para el mejoramiento de la trazabilidad, la reducción de los tiempos de evaluación del ciclo del software en el contexto ágil de la DTIC y la generación de informes estadísticos.

2.2 Objetivos Específicos

Realizar un análisis de los casos de prueba QA actuales, caracterizados por su ejecución manual y déficit en el diseño, orientado a la implementación de un proceso de automatización de pruebas, mediante herramientas que serán evaluadas en términos de compatibilidad, tiempo y entrega de productos de calidad en los módulos asignados para el Sistema de Información Financiero (SIF) en la DTIC.

Documentar en la Wiki de la DTIC los aspectos y prácticas relevantes que se incorporarán en el proceso de pruebas automatizadas a partir de la indagación de la metodología BDD.

Proponer la estructura y elementos del diseño de las pruebas automatizadas que abarquen los casos identificados para la ejecución eficiente, la mejora de la trazabilidad y la generación de informes estadísticos.

Realizar pruebas del plan propuesto, en el módulo de proveedores del SIF, para la comparación de los resultados con respecto a las manuales, en cuanto a tiempos de ejecución, cobertura y cantidad de defectos.

3. Marco de Referencia

Para entender y contextualizar adecuadamente la implementación de pruebas automatizadas, es fundamental explorar los conceptos y metodologías clave que sustentan su desarrollo. Este apartado aborda los pilares técnicos y teóricos que facilitan la calidad en el software y la eficiencia en su aseguramiento.

3.1 Calidad del Software

La calidad del software se refiere al grado en que el producto desarrollado satisface los requisitos especificados por el cliente y se alinea con los estándares de desarrollo previamente establecidos (Pressman, 1993). Estos estándares definen y determinan la utilidad y la existencia del producto dentro de su contexto de aplicación (García León & Beltrán Benavides, 1995).

Asimismo, la calidad de un producto de software se distingue por características como funcionalidad, eficiencia, compatibilidad, usabilidad, confiabilidad, seguridad, mantenibilidad y portabilidad (ISO/IEC, 2011).. Dichos atributos del software cuentan con diversas maneras para brindar calidad, especialmente, el aspecto funcional es el más relevante, ya que es el núcleo del producto.

Si un producto no cumple con la funcionalidad esperada, resulta ineficiente, sin importar el cumplimiento de otras características. Generalmente se mide con pruebas funcionales, en donde se priorizan las reglas de negocio del cliente o usuario.

3.1.1 Pruebas funcionales

Las pruebas funcionales son aquellas que se ejecutan para evaluar si un componente o sistema satisface los requisitos funcionales, en donde la lógica del software se pone a prueba mediante ejecuciones supervisadas, ya que los datos de entrada son definidos previamente y los

datos de entrada son definidos previamente y los resultados esperados se conocen de antemano (ISTQB, s. f.). Es decir, el objetivo principal de dichas pruebas es la validación del cumplimiento de los requerimientos del producto, los cuales se evalúan en base a casos de prueba diseñados. Existen algunos tipos de pruebas que se especifican a continuación.

3.1.1.1 Prueba de humo. La prueba de humo es un tipo de prueba cuyo objetivo es verificar que las funcionalidades más críticas de un software operen correctamente antes de continuar con fases más profundas de validación (Rasmusson, 2016). En otras palabras, es una prueba que ejecuta las funciones más importantes de un programa para obtener los mínimos necesarios de su funcionamiento, depurando fallos tempranamente, y a su vez, determinando si es lo suficientemente estable como para realizar pruebas más rigurosas.

3.1.1.2 Prueba de regresión. La prueba de regresión se ejecuta para confirmar que los cambios recientes en el código no han afectado negativamente ninguna de las funcionalidades ya existentes. La prueba de regresión tiene como finalidad garantizar que no se introduzcan nuevos defectos en áreas del software que no han sido modificadas tras la incorporación de una nueva funcionalidad, mejora o corrección en otra parte del sistema (Naik & Tripathy, 2011).

3.2 Metodología BDD

El desarrollo orientado por comportamiento, conocido como BDD (Behaviour-Driven Development), es una metodología que fomenta la colaboración entre los diferentes roles de un equipo de desarrollo de software, centrando sus esfuerzos en comprender y satisfacer las necesidades del negocio. Este enfoque se basa en la creación de especificaciones claras y compartidas, las cuales sirven tanto como documentación como base para la validación del software (Smart & Molak, 2023).

3.3 Automatización de Pruebas

La automatización de pruebas tiene como objetivo realizar pruebas funcionales de manera más rápida y eficiente (Garg, 2014). Como resultado, muchos mercados han incorporado diversos procesos automatizados en sus líneas de producción, incluido el desarrollo de software. Sin embargo, aunque la automatización de pruebas ofrece numerosos beneficios como repetibilidad, reutilización y velocidad, es importante entender qué se puede y qué no se puede probar de forma automática para evitar el mal uso de recursos. Asimismo, es importante brindar a los desarrolladores herramientas que faciliten y aceleren la implementación de las pruebas automatizadas.

3.4 Herramientas para la Automatización de Pruebas

Existen diversas herramientas de automatización que apoyan las pruebas en el desarrollo de software, facilitando su desarrollo y gestión (Calidad y Software, s. f.). Estas herramientas se pueden clasificar según su funcionalidad: algunas están dedicadas a la gestión de pruebas, otras al desarrollo de pruebas. Actualmente existen herramientas que brindan soporte integral en ambas áreas, adaptadas a las necesidades específicas de las pruebas que se deben realizar.

3.4.1 Entornos de automatización

Los entornos de automatización son herramientas o plataformas diseñadas para automatizar las pruebas de software. Principalmente, permiten simular la interacción del usuario con la aplicación, verificar su correcto funcionamiento y detectar eficazmente posibles fallos. A su vez, se caracteriza por la reducción del tiempo de prueba, la posibilidad de realizar ejecuciones repetitivas y consistentes, y la capacidad de realizar pruebas en múltiples entornos (como distintos navegadores o sistemas operativos) simultáneamente.

Existen notables entornos de automatización como, por ejemplo, Selenium, Cypress y Appium, que se adaptan a necesidades específicas que van desde pruebas web hasta pruebas de aplicaciones móviles.

3.4.2 Herramientas de informes

Las herramientas de generación de informes de pruebas automatizadas ayudan a obtener una representación visual y detallada de los resultados de las pruebas. Estas herramientas no solo muestran qué pruebas se aprobaron o fallaron, sino que también muestran datos valiosos sobre la cobertura de las pruebas, el tiempo de ejecución y los patrones de error.

Algunas de las herramientas de generación de informes fundamentales son Serenity BDD, Allure Report, y ExtentReports, las cuales permiten crear informes claros y visualmente eficaces que ayudan a identificar rápidamente los errores y facilitan la trazabilidad de las pruebas.

3.4.3 Lenguajes y herramientas para escribir escenarios

Los lenguajes y herramientas para escribir escenarios permiten la descripción de casos de prueba de una manera sencilla, estructurada y similar al lenguaje natural. Esto permite una colaboración fácil entre equipos técnicos y no técnicos, ya que todos pueden entender cómo funciona, y en algunos casos, contribuir al diseño de pruebas. Estas herramientas se rigen por principios como Behavior-Driven Development (BDD) y, a menudo, utilizan estructuras como "Given-When-Then" (Marchese, Zen, & Villafiorita, s. f.). para expresar el comportamiento esperado del sistema. Un ejemplo conocido es Cucumber, que utiliza el lenguaje Gherkin para crear escenarios fáciles de entender.

3.5 Patrones de Pruebas Automatizadas

Los patrones de pruebas automatizadas son una forma de resolver ciertos problemas en dichas pruebas, mediante el uso de un conjunto de prácticas establecidas que funcionan bien para muchas personas (Kovalenko, 2014). Estos patrones no sólo proporcionan soluciones válidas, sino que también ayudan a estandarizar y simplificar el proceso de pruebas, haciéndolo más eficiente y confiable. Entre los patrones más relevantes en la automatización de pruebas, se encuentran algunos que destacan como POM, Screenplay y Command.

3.6 Metodología Ágil de Desarrollo Scrum

La metodología ágil de desarrollo SCRUM representa un marco de trabajo cuyo objetivo es la supervisión constante del estado del software existente, en la cual la participación del cliente se basa en establecer las prioridades y metas a cumplir por parte del equipo SCRUM, quien se autogestiona con el fin de mejorar la forma en la que se entregan los resultados (Fuentes, 2015). Su integración con la automatización de pruebas QA promueve un ciclo de desarrollo más rápido, eficiente y adaptable, optimizando la calidad y la eficiencia del producto.

3.7 Control de Versiones para Pruebas Automatizadas

El control de versiones es una práctica importante en el desarrollo de software, la cual implica el seguimiento y la gestión de los cambios realizados en el código fuente a lo largo del tiempo. En el contexto de las pruebas automatizadas, su importancia se incrementa, ya que permite a los equipos de desarrollo y aseguramiento de calidad (QA) colaborar de manera eficiente y mantener un historial detallado de las modificaciones en los scripts de prueba.

Por otra parte, al utilizar sistemas de control de versiones, como por ejemplo, Git, es posible sincronizar las versiones y evidenciar que los cambios no entren en conflicto con los de otros

usuarios, facilitando tanto la integración continua como la entrega constante en entornos de desarrollo ágiles (Microsoft, s. f.).

3.8 Historias de Usuario (HU)

Las historias de usuario son una herramienta clave en metodologías ágiles, utilizadas para expresar los requerimientos del sistema desde la perspectiva del usuario final. Destacan por describir de manera breve y clara las funcionalidades que un sistema debe tener para resolver los problemas de los usuarios o satisfacer sus necesidades. A diferencia de los requerimientos tradicionales, las historias de usuario promueven un enfoque colaborativo, en el que el equipo de desarrollo y el dueño del producto mantienen conversaciones continuas para afinar los detalles y asegurar que se cumplan las expectativas del cliente (Cohn, 2009).

3.8.1 Criterios de aceptación

Los criterios de aceptación son condiciones específicas que una historia de usuario debe cumplir para ser considerada completa y aceptada por el cliente. Estos criterios definen los resultados esperados de una funcionalidad, asegurando que cumpla con las expectativas del usuario y los requisitos del negocio (Cohn, 2004). En otras palabras, son las "reglas" que determinan cuándo una historia se puede dar por terminada, y su propósito es ayudar a que lo entregado sea lo que realmente se necesita.

3.8.2 Creación de casos de prueba a partir de la HU

La creación de casos de prueba a partir de historias de usuario (HU) es un proceso fundamental en el desarrollo ágil de software. Consiste en transformar los requerimientos expresados en las HU en escenarios específicos que validen si el sistema cumple con las

expectativas del usuario. Cada caso de prueba debe reflejar un criterio de aceptación definido en la HU, asegurando que la funcionalidad implementada sea la esperada.

Para elaborar estos casos de prueba, es fundamental analizar detalladamente la historia de usuario, identificar los escenarios posibles y definir los datos de entrada y resultados esperados. Este enfoque ayuda a que se cubran tanto los caminos principales como los alternativos, permitiendo detectar posibles defectos antes de la implementación (Guía para la generación de casos de prueba a partir de historias de usuario, s. f.).

Además, facilita la comunicación entre el equipo de desarrollo y el cliente, ya que los casos de prueba sirven como base para verificar que el producto final cumple con los requerimientos establecidos.

4. Marco Metodológico

Para alcanzar los objetivos planteados, se define un marco metodológico basado en un enfoque estructurado y dividido en fases. Cada fase aborda un conjunto específico de actividades orientadas a definir, implementar y evaluar el proceso de automatización de pruebas QA del módulo de proveedores del SIF, en el contexto ágil de la DTIC. A continuación, se presentan las fases que guían este proyecto.

4.1 Fase 1: Evaluación de los casos de prueba QA actuales y definición del proceso de automatización

En esta fase se analiza el estado actual de los casos de prueba del módulo de proveedores, las herramientas utilizadas para su ejecución y registro, y se identifican oportunidades de mejora. Con base en los hallazgos, se diseña un proceso de automatización que permita optimizar las pruebas funcionales y reducir los tiempos del ciclo de desarrollo.

Actividad 1. Recopilación de información sobre casos de prueba:

- Identificar y documentar los casos de prueba manuales actuales.
- Analizar su alineación con los requisitos funcionales y la cobertura de pruebas.

Actividad 2. Revisión de herramientas actuales:

- Evaluar las herramientas utilizadas para ejecutar pruebas (por ejemplo, hojas de cálculo, herramientas de gestión de pruebas).
- Revisar los métodos empleados para registrar resultados y realizar seguimientos.

Actividad 3. Identificación de deficiencias:

- Detectar casos de prueba redundantes, deficitarios o que requieran una mejor definición.

Actividad 4. Definición del proceso de automatización:

- Diseñar un flujo de trabajo para la automatización de pruebas.

4.2 Fase 2: Incursión de la metodología BDD en el ciclo de desarrollo del software

En esta fase se estudia y adapta la metodología BDD al contexto del ciclo de desarrollo de la DTIC. El objetivo es identificar en qué etapas de desarrollo QA debe intervenir con pruebas automatizadas basadas en BDD, asegurando una integración temprana de las pruebas funcionales en el proceso.

Actividad 1. Indagación sobre BDD y sus beneficios:

- Revisar conceptos clave, beneficios y buenas prácticas relacionadas con BDD.
- Documentar los hallazgos en la Wiki de la DTIC para futuras referencias.

Actividad 2. Definición del punto de intervención de QA en el ciclo de desarrollo:

- Identificar el momento adecuado para que QA comience a automatizar los casos de prueba (por ejemplo, al recibir historias de usuario detalladas).

- Diseñar un flujo que permita a QA trabajar en paralelo con el desarrollo del software.

Actividad 3. Documentación y alineación:

- Guía Teórica de BDD en la Wiki de la DTIC.
- Guía en la Wiki de la DTIC con la adopción de prácticas de BDD en el equipo de QA.

4.3 Fase 3: Diseño de la estructura y configuración del entorno para los scripts automatizados

En esta fase se configura el entorno de trabajo necesario para la automatización de pruebas y se diseña la estructura de los scripts. También se valida y optimiza el diseño para ayudar a la eficacia y mantenibilidad del framework de pruebas.

Actividad 1. Configuración del entorno de automatización:

- Instalar y configurar todas las herramientas de automatización seleccionadas
- Establecer el entorno de trabajo

Actividad 2. Justificación del enfoque y estructura de trabajo:

- Elegir un patrón de automatización y justificar su selección con base en las necesidades del proyecto.
- Diseñar la estructura de directorios y la organización de los scripts.

Actividad 3. Diseño de scripts iniciales:

- Crear scripts automatizados para los casos de prueba prioritarios definidos en la fase 1.
- Promover que los scripts sean modulares y reutilizables.

Actividad 4. Validación y optimización de scripts:

- Validar los scripts ejecutándolos en diferentes escenarios y entornos.
- Identificar oportunidades de optimización para mejorar la eficiencia y reducir tiempos de ejecución.

4.4 Fase 4: Evaluación de la automatización en comparación a lo manual

En esta fase se mide el impacto del proceso de automatización mediante la comparación entre los tiempos de prueba manuales y automatizados, evaluando también la cobertura y cantidad de defectos encontrados.

Actividad 1. Medición de tiempos en pruebas manuales:

- Revisar las pruebas manuales realizadas y obtener sus tiempos de ejecución.

Actividad 2. Ejecución de pruebas automatizadas:

- Implementar los scripts en entornos controlados y registrar los tiempos obtenidos.

Actividad 3. Análisis comparativo:

- Contrastar los resultados entre pruebas manuales y automatizadas, evaluando la reducción en tiempos, cobertura y cantidad de defectos detectados.

Actividad 4. Documentación de resultados:

- Consolidar los informes estadísticos de las pruebas.

4.5 Fase 5: Síntesis y entregables

En esta fase se consolidan los resultados obtenidos a lo largo de las fases anteriores y se formalizan los entregables finales del proyecto, los cuales reflejan la implementación del proceso de automatización de pruebas y su aplicación en un entorno real de desarrollo.

Actividad 1. Consolidación y versionamiento de los scripts automatizados:

- Organizar y publicar los scripts automatizados desarrollados en el repositorio Git, permitiendo su control de versiones, consulta y reutilización por parte del equipo de QA.

Actividad 2. Documentación técnica de la solución de automatización:

- Elaboración de la documentación técnica que describe el proceso de automatización implementado, la arquitectura del framework, la estructura del código y los lineamientos básicos para su ejecución y mantenimiento, integrados dentro del documento del trabajo de grado y su apéndice.

Actividad 3. Documentación final del proyecto:

- Redacción y consolidación del documento final de la tesis de trabajo de grado en modalidad de práctica empresarial, en el cual se presentan la metodología aplicada, los resultados obtenidos, las conclusiones y las oportunidades de trabajo futuro.

5. Implementación

En esta sección se presenta la implementación de los objetivos planteados, y se explica cómo se podría mejorar de pruebas de calidad en la DTIC, pasando de pruebas manuales a un enfoque automatizado en el módulo de proveedores. Se analiza el estado actual de las pruebas, se identifican deficiencias y se priorizan los casos más pertinentes para automatización.

Además, se define e implementa un proceso estructurado de pruebas automatizadas a través de herramientas, buscando disminuir los tiempos de respuesta y promover una mayor confiabilidad en el software.

5.1 Orientación del proceso de automatización de pruebas QA en la DTIC

El propósito de esta sección es establecer la orientación del proceso de automatización de pruebas de calidad de software (QA) en la División de Tecnologías de la Información y la Comunicación (DTIC), tomando como caso de estudio el módulo de proveedores del Sistema de Información Financiero. Para ello, se analiza el estado actual de las pruebas existentes, las cuales se caracterizan principalmente por su ejecución manual, con el fin de identificar fortalezas, debilidades y oportunidades de mejora en el diseño y ejecución de los casos de prueba.

A partir de este análisis, se determinan los casos de prueba que presentan mayor pertinencia para su automatización, considerando criterios como criticidad funcional, frecuencia de ejecución y estabilidad de la funcionalidad. Adicionalmente, se definen y recopilan métricas clave que permitirán comparar el desempeño de las pruebas manuales frente a las pruebas automatizadas, facilitando la evaluación de las diferencias en términos de tiempo de ejecución, cobertura y calidad de los resultados obtenidos.

5.1.1 Análisis de los casos de prueba manuales existentes en el módulo de proveedores

El módulo de proveedores es una parte fundamental del SIF, ya que gestiona información y procesos relacionados con un gran volumen de usuarios. Su correcto funcionamiento es esencial para orientar la incorporación, validación y pagos de proveedores de manera eficiente y segura.

Dentro de este módulo, se manejan distintos tipos de proveedores (persona natural, empresa jurídica, empresa jurídica extranjera), por lo que es fundamental validar que los datos registrados sean correctos y cumplan con las reglas establecidas. Por este motivo, se evalúa la cobertura y efectividad de las pruebas manuales actuales, con un enfoque en los procesos CRUD y la validación de información importante para cada tipo de proveedor.

5.1.1.1 Revisión del alcance de los casos de prueba actuales. Para ayudar con la calidad y estabilidad del módulo de proveedores en el SIF, es fundamental evaluar la cobertura de los casos de prueba manuales existentes. Un análisis detallado permite identificar brechas en las pruebas actuales, asegurando que todas las funcionalidades críticas estén validadas y que el sistema maneje adecuadamente los distintos escenarios de usuario. La revisión se llevó a cabo con base en los siguientes criterios:

Criterios de evaluación

- **Cobertura funcional:** Se analizó si los casos de prueba existentes contemplan todas las funcionalidades esenciales del módulo de proveedores, incluyendo la gestión de creación, modificación, eliminación y consulta de proveedores (CRUD). Además, se verificó si los casos de prueba validan procesos relevantes como la validación de algunos documentos y la activación de proveedores.
- **Cobertura de reglas de negocio:** Se revisó si las pruebas consideran todas las reglas de negocio establecidas para los distintos tipos de proveedores (personas naturales, nacionales y extranjeras). Esto incluyó restricciones sobre los tipos de documentos requeridos, la complejidad de cada formulario, y validaciones adicionales en distintos campos.
- **Cobertura de validaciones y restricciones:** Se examinó si los casos de prueba incluyen validaciones de errores, restricciones y mensajes de advertencia. Esto abarcó la verificación de campos obligatorios, la correcta aplicación de restricciones de formato en la entrada de datos y la validación de errores en la interacción con otros módulos del sistema.

Metodología para la revisión de los casos de prueba

- **Revisión de la documentación existente:** Se analizaron los casos de prueba documentados en la herramienta de gestión de pruebas utilizada por la DTIC, verificando su estructura, claridad y alineación con los requerimientos del sistema.
- **Ejecución de pruebas en ambiente controlado:** Se ejecutaron algunos de los casos de prueba manuales para validar su efectividad y detectar posibles inconsistencias o brechas en la cobertura.
- **Comparación con especificaciones funcionales:** Se contrastaron los casos de prueba actuales con la documentación de requerimientos y/o historias de usuario para detectar funcionalidades que no están siendo validadas adecuadamente.

Hallazgos principales

Tras el análisis, se identificaron las siguientes observaciones sobre el estado actual de los casos de prueba manuales del módulo de proveedores:

- **Cobertura limitada en escenarios alternativos y de error:** La mayoría de los casos de prueba están enfocados en flujos principales de éxito, pero no validan adecuadamente escenarios alternativos, como intentos de registro con información incompleta o proveedores ya existentes.
- **Generalidad y baja especificidad en los casos de prueba:** Se evidenció que varios casos de prueba fueron definidos de manera excesivamente general, sin detallar claramente las reglas de negocio ni los escenarios específicos que se pretendían validar. Esta situación dificulta que actores externos al proceso comprendan con precisión qué se está probando, especialmente en otros módulos

del SIF que consumen información del proveedor, como el módulo de contratación, donde los datos del proveedor son transversales a gran parte del flujo funcional.

- **Falta de tiempo para ejecutar las pruebas:** Se pudo observar que los tiempos de ejecución de las pruebas manuales son bastante extensos, y a medida que se incorporan nuevas funcionalidades, estos aumentan aún más, ya que las pruebas se vuelven más exhaustivas. Esto ha ocasionado un déficit en su ejecución, dejando muchas áreas sin contemplar.
- **Inconsistencia en la documentación y gestión de pruebas:** No hay una clasificación clara de los casos de prueba en ejecución periódica y aquellos que han quedado obsoletos. Esto genera incertidumbre sobre qué pruebas siguen siendo relevantes para la operación actual del sistema.

Estos hallazgos permiten identificar oportunidades de mejora en la estrategia de pruebas manuales y sirven como base para la priorización de casos pertinentes a ser automatizados en fases posteriores del proyecto.

5.1.1.2 Identificación de deficiencias en el diseño y ejecución. A partir del análisis detallado de los casos de prueba manuales, se identificaron distintas deficiencias que afectan la calidad, eficiencia y confiabilidad de las pruebas. Estas deficiencias se clasificaron en dos grandes áreas: problemas en el diseño de los casos de prueba y dificultades en su ejecución.

Para esta evaluación, se tomaron en cuenta factores de cobertura de pruebas, consistencia en la documentación y optimización de procesos. Se analizaron los formatos utilizados en cada fase de pruebas, la estructura de cada caso de prueba y su alineación con los requisitos del sistema. Además, se compararon los tiempos de ejecución y la efectividad de las pruebas en la detección de defectos. A continuación, se detallan los hallazgos más relevantes:

Deficiencias en el diseño de los casos de prueba

- **Redundancia de pruebas sin valor agregado:** Se identificó la presencia de múltiples casos de prueba que validan el mismo flujo funcional sin aportar nuevas variaciones o escenarios. Esto no solo incrementa innecesariamente el tiempo de ejecución, sino que también dificulta la identificación de pruebas significativas y el rendimiento de los esfuerzos al momento de la validación.
- **Falta de detalles en la especificación de pruebas:** Algunos casos de prueba se encontraron bastante generales y no desglosan con precisión los pasos a seguir. Esta ambigüedad genera interpretaciones distintas entre los *testers* y puede provocar inconsistencias en los resultados. Por ejemplo, en pruebas relacionadas con la validación de proveedores, algunos casos no especifican claramente qué pasos se deben seguir para cada tipo de proveedor, pues no se le apunta a la especificidad, dejando espacio para ejecuciones erróneas.
- **Inconsistencia en la estructura y documentación:** Se evidenció que los casos de prueba a pesar de que siguen un formato estandarizado, no se cumple con la información requerida en todas las columnas. En varios registros, falta información en secciones relevantes como precondiciones, datos de prueba específicos o resultados esperados. Esto dificulta la trazabilidad de las pruebas y su reutilización en futuras ejecuciones.
- **Cobertura parcial en validaciones de integración y datos:** Si bien la mayoría de los casos de prueba cubren funcionalidades individuales dentro del módulo de proveedores, se encontró que las pruebas de integración con otros módulos del SIF (como contratación) son insuficientes.

Deficiencias en la ejecución de los casos de prueba

- **Tiempo excesivo en la ejecución manual:** Se observó que algunas pruebas requieren validaciones repetitivas que podrían optimizarse mediante la automatización, sobre todo al realizar las pruebas de regresión. Por ejemplo, cada vez que se modifican ciertos parámetros de los proveedores, se deben realizar múltiples validaciones en distintos puntos del sistema, lo que prolonga innecesariamente los tiempos de prueba.
- **Errores en la documentación de resultados:** Se encontraron inconsistencias entre la documentación de las pruebas y los resultados reales obtenidos en ejecución. En algunos casos, los reportes no reflejan con precisión los errores detectados, generando confusión cuando se realiza su revisión, y retrasando su resolución.
- **Dependencia de datos externos y configuraciones manuales:** Muchas pruebas dependen de información que debe cargarse manualmente antes de su ejecución, como la creación de proveedores con características específicas. Esto introduce retrasos en la ejecución de pruebas y aumenta la posibilidad de errores humanos al preparar los datos de prueba.

En general, estos hallazgos reflejan la necesidad de mejorar tanto el diseño como la ejecución de los casos de prueba, cumplir adecuadamente con la estandarización de la documentación y avanzar en la automatización de pruebas para reducir tiempos y priorizar la cobertura.

5.1.1.3 Priorización de casos relevantes para automatización. Dado el impacto del módulo de proveedores en el flujo del SIF y la necesidad de optimizar el proceso de validación, se realizó un análisis completo para identificar los casos de prueba más relevantes para la automatización. La selección de estos casos se basa en el alcance de los procesos involucrados, la frecuencia con la que se ejecutan y el nivel de riesgo que representan en caso de fallos. Además, se consideró el beneficio que aportaría la automatización en términos de reducción de tiempos de prueba y mejora en la detección de errores.

El módulo de proveedores involucra múltiples interacciones con otros módulos del sistema, tales como contratación. Esto significa que cualquier error en la información registrada puede generar inconsistencias en el manejo general del SIF, afectando desde la habilitación de proveedores hasta la correcta liquidación de pagos. Por esta razón, se establecieron criterios específicos para priorizar los casos de prueba a automatizar, asegurando que aquellos de mayor impacto sean cubiertos de manera eficiente.

Criterios de selección de casos de prueba para automatización

Para definir qué pruebas deben ser automatizadas, se establecieron los siguientes criterios de priorización:

- **Frecuencia de ejecución:** Se priorizaron los casos de prueba que deben ejecutarse con cada actualización del sistema o que forman parte del proceso diario de validación de proveedores. Esto permite reducir el esfuerzo manual y ayudar con la estabilidad del sistema a lo largo del tiempo.
- **Complejidad del proceso:** Se identificaron casos que involucran múltiples pasos, validaciones y condiciones específicas. En particular, aquellos que requieren el ingreso y validación de datos detallados o campos específicos según el tipo de

proveedor, asegurando que la información registrada sea consistente en todo el sistema.

- **Impacto en otros módulos:** Se priorizaron las pruebas que afectan la integración con otros módulos importantes, como contratación. Cualquier inconsistencia en la información de un proveedor podría generar errores en estos procesos, lo que hace fundamental su validación automatizada.
- **Nivel de riesgo asociado a fallos:** Se identificaron los casos donde un error podría generar consecuencias significativas, como la habilitación incorrecta de un proveedor o problemas en la validación de documentos y cuentas bancarias.
- **Tiempo de ejecución manual:** Se consideraron los casos de prueba que actualmente requieren un esfuerzo significativo en su ejecución manual debido a la cantidad de datos a validar, la necesidad de configurar información o la repetición constante de los mismos pasos.

Casos críticos y relevantes seleccionados para automatización

Con base en los criterios anteriores, se determinaron las siguientes funcionalidades como prioritarias para la automatización de pruebas:

1. **Creación, Actualización, Visualización y Eliminación de proveedores**
    - **Proveedor Persona Natural:** Este proceso consta de 5 pasos (interfaces) fundamentales para completar todo el formulario de creación el proveedor que deben ser validados de manera detallada:
        - Información básica
        - Responsabilidades tributarias
        - Actividades económicas

    - Información bancaria
    - Seguridad social
- **Proveedor Empresa Nacional:** Similar al proceso de persona natural, también cuenta con 5 pasos, aunque con ligeras variaciones en algunos campos:
    - Información básica
    - Responsabilidades tributarias
    - Actividades económicas
    - Información bancaria
    - Seguridad social
- **Proveedor Empresa Extranjera:** A diferencia de los anteriores, este tipo de proveedor solo requiere un paso para completar el formulario y registrarse:
    - Información básica

2. Consulta de Proveedores

La funcionalidad de consulta permite buscar proveedores utilizando 6 criterios diferentes, por lo que es fundamental validar su correcto funcionamiento con diversas combinaciones de datos:

- Nombre
- Documento
- Tipo de proveedor
- Estado del proveedor
- Ciudad de registro

Estos apartados representan los procesos más relevantes dentro del módulo de proveedores, ya que cualquier error en su ejecución podría comprometer la integridad de la información y afectar el correcto funcionamiento del sistema. La automatización de estas pruebas permitirá detectar errores de manera más rápida, reducir la carga operativa del equipo de QA y mejorar la estabilidad general del SIF.

5.1.1.4 Recopilación de datos importantes en la ejecución de pruebas manuales. Para evaluar la eficiencia y el impacto de las pruebas manuales en el módulo de proveedores, se recopilaron datos clave sobre el proceso de ejecución. Estos datos incluyen el tiempo promedio de ejecución, la cantidad de casos de prueba ejecutados y los issues reportados por cada Historia de Usuario (HU) relacionada con los apartados relevantes previamente identificados: CRUD de proveedores y su respectiva consulta.

Este análisis permite identificar cuellos de botella en la ejecución manual de pruebas, medir la efectividad de la detección de errores y justificar la necesidad de automatización en los procesos con mayor carga de trabajo. A continuación, se presenta la recopilación de los datos organizados por funcionalidad:

Tabla 1

Datos de ejecución de pruebas manuales por funcionalidad

Funcionalidad	Historia de Usuario (HU)	Tiempo de ejecución	Casos de prueba ejecutados	Issues reportados
CRUD Proveedores	#515 Crear información básica de un proveedor tipo persona	5 horas	12	5
	#516 Crear información tributaria de un proveedor	2 horas	7	1

#617 Estados y beneficios proveedor	3 horas	10	1
#521 Crear actividades económicas de un proveedor	3 horas	12	2
#688 Ajuste guardar actividades económicas	3 horas	12	0
#2018 Ajuste HU 521 V2 actividades económicas	2 horas	9	0
#448 Crear y editar cuenta bancaria de un proveedor	2 horas	8	2
#612 Crear múltiples cuentas bancarias proveedor	4 horas	12	2
#757 Visualizar información básica proveedor	2 horas	8	1
#817 Visualizar responsabilidad, estados y actividades	2 horas	10	1
#761 Automatizar llenar formulario de proveedores	4 horas	13	5
#1026 Automatizar edición de la información de un proveedor	4 horas	12	3
#1055 Ajuste flujo proveedores	2 horas	12	1
#1575 Ajuste HU 437 V2 campos tipo	3 horas	8	0

	suggestion interfaz proveedores			
	#1576 Afiliaciones salud y pensión proveedores	4 horas	10	3
	#436 Crear información básica de proveedores tipo empresa	4 horas	12	3
	#747 Crud Sucursales para proveedores empresa	5 horas	12	3
	#763 Ajuste bancos de un proveedor empresa para las sucursales	6 horas	18	1
	#1257 Ajuste proveedor empresa extranjera	3 horas	13	3
Consulta de Proveedores	#437 Crear Interfaz de proveedores (Consultar proveedores)	3 horas	8	1
Total	-	66 horas	218	35

Esta recopilación de datos servirá como referencia para comparar, en etapas posteriores, el impacto de la automatización frente a las pruebas manuales. Al analizar estas métricas, será posible evaluar la reducción en tiempos de ejecución, la mejora en la detección de errores y la optimización del proceso de ejecución y validación en el módulo de proveedores.

5.1.2 Evaluación de las herramientas utilizadas actualmente para pruebas de QA

El aseguramiento de calidad dentro de la DTIC depende en gran medida de las herramientas utilizadas para documentar, ejecutar y gestionar las pruebas. Actualmente, el proceso de pruebas se basa en herramientas manuales y semimanuales, lo que implica ciertos desafíos en términos de eficiencia y automatización. Por este motivo, en esta sección se analizan las herramientas utilizadas en el área de QA, identificando su propósito, las limitaciones que presentan y las oportunidades de mejora para optimizar el proceso.

5.1.2.1 Identificación de las herramientas utilizadas actualmente y su propósito. El equipo de QA, en su día a día, cuenta con diversas herramientas a la hora de realizar todo el proceso de pruebas, desde el diseño de los casos de prueba hasta su ejecución y documentación. A continuación, se detallan las herramientas empleadas actualmente, su propósito dentro del flujo de pruebas y su impacto en la gestión de calidad:

- **Excel - Gestión y documentación de casos de prueba**

  Excel es la herramienta principal utilizada por el equipo de QA para documentar los casos de prueba manuales. A través de hojas de cálculo organizadas, se registran cada uno de los escenarios de prueba, los pasos a seguir, los datos de entrada y los resultados esperados. Durante la ejecución de las pruebas, los testers completan manualmente cada caso en una lista de chequeo, marcando el estado de la prueba (exitosa, fallida, bloqueada, etc.) y agregando comentarios adicionales si es necesario.

  Además, en Excel se adjunta la evidencia de cada prueba, lo que implica la inserción de capturas de pantalla o referencias a archivos externos que respalden los resultados obtenidos. Este método permite mantener un registro estructurado, pero tiene desventajas significativas, como la falta de integración con otras herramientas de gestión y

la alta dependencia del ingreso manual de datos, lo que puede generar errores o inconsistencias.

- **Excel - Generación de informes de pruebas**

Al finalizar cada sprint, el equipo de QA debe consolidar toda la información de pruebas ejecutadas en un informe de calidad. Para ello, se recopilan manualmente los datos de los casos de prueba de cada funcionalidad evaluada, el estado final de cada caso, los defectos reportados y otros indicadores, como la tasa de éxito de las pruebas y todo aquello que sigue pendiente por probar.

Este proceso también se realiza en Excel, lo que significa que cada tester debe extraer la información de su propia hoja de pruebas y trasladarla al informe final. Esto no solo implica una tarea repetitiva y propensa a errores, sino que además dificulta la generación de reportes en tiempo real, ya que la consolidación de datos solo se realiza al término de cada sprint.

- **Taiga - Gestión de defectos y tareas**

Para el registro y seguimiento de defectos, el equipo de QA emplea Taiga, una herramienta de gestión ágil que permite organizar las tareas del equipo de desarrollo, UX y QA. Cada issue identificado durante las pruebas es reportado en Taiga con la siguiente información:

1. **Tipo de defecto:** Se clasifican en funcionales, GUI (Interfaz gráfica) o GAP (Funcionalidad faltante).
2. **Descripción del defecto:** Se documenta el problema encontrado, incluyendo los pasos para reproducirlo y el resultado esperado.

3. **Estado:** Se clasifica el defecto según su avance en el flujo de resolución (abierto, en progreso, resuelto, cerrado, etc.).
4. **Prioridad:** Se asigna un nivel de criticidad al defecto para determinar su impacto en el sistema.
5. **Responsable:** Se indica el miembro del equipo encargado de revisar y corregir el defecto.

Taiga proporciona una visión más estructurada del estado de los defectos y facilita la comunicación con los desarrolladores. Sin embargo, al no estar integrado directamente con la documentación de pruebas en Excel, los testers deben trasladar manualmente los defectos encontrados en las pruebas hacia Taiga, lo que incrementa el esfuerzo de trabajo y la posibilidad de que algunos errores no sean correctamente documentados o rastreados.

En este sentido, el uso de Excel y Taiga ha permitido estructurar el proceso de pruebas en la DTIC, proporcionando un método organizado para la documentación y gestión de calidad. Sin embargo, la dependencia de herramientas manuales genera varios desafíos, como la fragmentación de la información, la falta de automatización y el esfuerzo operativo necesario para consolidar los datos en informes de calidad.

5.1.2.2 Análisis en las limitaciones de las herramientas actuales. Si bien las herramientas utilizadas actualmente en el proceso de QA han permitido estructurar la documentación y el seguimiento de pruebas, presentan diversas limitaciones que afectan la eficiencia, escalabilidad y capacidad de análisis del equipo. Estas deficiencias generan una mayor carga operativa manual y reducen la capacidad de respuesta ante defectos.

El análisis de estas limitaciones es valioso para determinar áreas de mejora y justificar la necesidad de la automatización mediante herramientas más avanzadas e integradas. Enseguida, se presentan las principales problemáticas identificadas:

1. Falta de automatización en la ejecución y documentación de pruebas

Actualmente, la ejecución de pruebas manuales se registra en hojas de cálculo de Excel, donde cada tester debe documentar los pasos seguidos, ingresar los resultados obtenidos y anexar evidencia en forma de capturas de pantalla o enlaces a archivos externos. Este proceso es completamente manual, lo que introduce varias limitaciones:

- **Alto esfuerzo operativo:** La necesidad de registrar cada caso de prueba y actualizar manualmente su estado incrementa el tiempo requerido para completar un ciclo de pruebas, especialmente en proyectos con un alto volumen de funcionalidades por evaluar.
- **Mayor riesgo de errores humanos:** Al depender de la entrada manual de datos, existe la posibilidad de inconsistencias, omisiones o errores en el registro de pruebas, lo que puede afectar la precisión de los reportes de calidad.
- **Falta de estandarización en la documentación:** Aunque existen formatos predefinidos en Excel, la redacción y estructura de los casos de prueba

pueden variar entre testers, lo que dificulta la interpretación homogénea de los resultados.

2. Fragmentación del proceso de pruebas

El flujo de trabajo de QA se encuentra disperso en múltiples herramientas sin una integración directa entre ellas:

- Documentación y ejecución de pruebas en Excel
- Gestión de defectos en Taiga
- Consolidación de reportes manualmente en Excel

Dado que estas herramientas no están interconectadas, la información de pruebas y defectos debe ser transferida manualmente entre ellas. Esto genera los siguientes problemas:

- **Duplicidad de trabajo:** Los testers deben registrar los defectos detectados en Excel y luego trasladarlos manualmente a Taiga, lo que implica una tarea redundante.
- **Pérdida de trazabilidad:** Dado que los casos de prueba y los defectos no están vinculados directamente, es difícil rastrear qué pruebas generaron qué errores, lo que impacta la capacidad de análisis de causa raíz.
- **Retrasos en la consolidación de datos:** Al no contar con una fuente única de información, el proceso de recopilación de datos para informes es tardado y propenso a errores, ya que depende de la extracción manual de información desde múltiples archivos.

3. Dificultad para analizar métricas en tiempo real

El proceso de pruebas genera una gran cantidad de información que podría ser utilizada para evaluar la calidad del software y optimizar el proceso de desarrollo. Sin embargo, debido a la naturaleza manual de la documentación en Excel, el análisis de métricas es un proceso tardado y limitado:

- **No hay actualización en tiempo real:** Como los datos se consolidan al final de cada sprint, no es posible obtener métricas actualizadas sobre la cantidad de pruebas ejecutadas, defectos detectados o tasa de éxito en la ejecución.
- **Falta de visualización gráfica:** La información se encuentra en hojas de cálculo sin una estructura visual clara, lo que dificulta la identificación de tendencias o patrones en la calidad del software.
- **Dificultad para la toma de decisiones rápida:** Sin métricas accesibles en tiempo real, los miembros de los equipos de QA y desarrollo no pueden anticipar problemas ni realizar ajustes tempranos en el proceso de pruebas.

4. Proceso de reporte de defectos poco optimizado

El registro de defectos en Taiga es un proceso que debe realizarse de manera manual, copiando la información desde Excel y adaptándola al formato de la herramienta. Esto genera varias ineficiencias:

- **Inconsistencias en la documentación:** Al depender de la transcripción manual, existe la posibilidad de que los detalles del defecto no se registren con la misma precisión o estructura en ambas plataformas.

- **Pérdida de información:** Si un defecto no es transferido correctamente desde Excel a Taiga, puede quedar fuera del radar del equipo de desarrollo, afectando la calidad del software.
- **Mayor tiempo de respuesta en la corrección de errores:** Como los defectos no se reportan de manera inmediata en una plataforma integrada, los desarrolladores pueden tardar más en atender problemas críticos.

5. Escalabilidad limitada

A medida que el sistema crece y se desarrollan nuevas funcionalidades, el volumen de casos de prueba y defectos a gestionar aumenta significativamente. Sin una solución automatizada, el proceso de pruebas manuales enfrenta serias dificultades para escalar:

- **Carga operativa insostenible:** La gestión de cientos de casos de prueba de forma manual se vuelve cada vez más demandante y propensa a errores.
- **Falta de reutilización de pruebas:** No existen mecanismos eficientes para reutilizar pruebas ya ejecutadas, lo que obliga a repetir procesos manuales en cada sprint.
- **Dificultad para mantener la documentación actualizada:** A medida que el sistema evoluciona, los casos de prueba deben ajustarse para reflejar los cambios. Sin herramientas automatizadas, esta actualización depende del esfuerzo manual del equipo.

Las herramientas actuales han sido útiles para estructurar el proceso de QA, pero presentan múltiples limitaciones que afectan la eficiencia y escalabilidad del equipo. La falta de automatización, la fragmentación del proceso y la dificultad para obtener métricas en tiempo real generan retrasos y esfuerzos innecesarios que impactan la calidad del producto.

Estas limitaciones evidencian la necesidad de una evolución en la estrategia de pruebas, explorando soluciones que permitan integrar la documentación, ejecución y gestión de defectos en una plataforma unificada, reduciendo la carga manual y mejorando la trazabilidad del proceso.

5.1.2.3 Identificación de áreas de mejora en el uso de herramientas. A partir del análisis de las limitaciones en las herramientas actualmente utilizadas en el proceso de QA, se han identificado diversas oportunidades de mejora para optimizar la gestión de pruebas, reducir la carga operativa manual y mejorar la trazabilidad de los defectos. La implementación de metodologías de automatización y la integración de herramientas de pruebas facilitarán un flujo de trabajo más eficiente, reduciendo tiempos de ejecución y mejorando la confiabilidad de los resultados. A continuación, se presentan las principales áreas de mejora identificadas:

1. Implementación de una metodología de automatización de pruebas

Actualmente, el equipo de QA documenta y ejecuta los casos de prueba de forma manual en Excel, lo que implica una alta carga de trabajo y un riesgo elevado de errores humanos en la ejecución y consolidación de evidencia. La automatización permitirá optimizar este proceso, permitiendo una ejecución más precisa y repetible.

- **Propuesta de mejora:** Implementar un framework de automatización de pruebas que permita la ejecución estructurada de los casos de prueba de manera repetitiva y confiable, reduciendo la intervención manual.
- **Beneficios esperados:**
  - Disminución del tiempo de ejecución de pruebas repetitivas.
  - Generación automática de reportes de pruebas con los resultados.
  - Reducción del margen de error humano en la documentación de pruebas.
  - Mayor confiabilidad en pruebas de regresión y de validación continua.

2. Uso de pruebas automatizadas para la documentación y evidencia

El registro manual de evidencias de prueba en Excel requiere capturas de pantalla y documentación detallada, lo que ralentiza el proceso y dificulta la trazabilidad de los resultados. Implementar una estrategia automatizada de generación de reportes y evidencias facilitará la gestión de pruebas.

- **Propuesta de mejora:** Incorporar una solución que permita capturar automáticamente la evidencia de cada ejecución de prueba y generar reportes estructurados con logs detallados, reduciendo la necesidad de documentación manual.
- **Beneficios esperados:**
  - Reportes estructurados con indicadores de éxito/fallo en cada ejecución.
  - Reducción del tiempo dedicado a la recopilación de evidencia.

3. Mejora en la trazabilidad de defectos en Taiga

Actualmente, la documentación de defectos en Taiga se realiza de forma manual con base en los resultados obtenidos en Excel, lo que puede generar inconsistencias y pérdida de información. Con la implementación de un framework de automatización de pruebas, los reportes generados podrán servir como fuente de referencia directa para los desarrolladores.

- **Propuesta de mejora:** Incorporar en Taiga los enlaces a los reportes generados automáticamente en cada ejecución de pruebas, permitiendo que los desarrolladores tengan acceso directo a la evidencia de fallos y resultados de las pruebas.
- **Beneficios esperados:**
  - Mayor claridad en la comunicación entre QA y desarrollo.

    - Reducción del esfuerzo manual en la documentación de defectos.
    - Mejora en la trazabilidad de defectos al contar con información estructurada.

4. Generación de métricas más precisas para el análisis de calidad

El informe de pruebas al final de cada sprint requiere la consolidación manual de datos desde Excel, lo que impide obtener métricas en tiempo real y aumenta el esfuerzo en la generación de reportes. Con la incorporación de reportes estructurados en las pruebas automatizadas, será posible obtener estadísticas más claras que faciliten la elaboración del informe final de calidad.

- **Propuesta de mejora:** Aprovechar las métricas generadas en los reportes automáticos de ejecución de pruebas para analizar tendencias y detectar patrones que permitan mejorar la calidad del software.
- **Beneficios esperados:**
    - Eliminación del esfuerzo manual en la consolidación de datos para métricas.
    - Obtención de estadísticas más precisas sobre la ejecución de pruebas.
    - Mayor confiabilidad en los informes de calidad del sprint.
    - Facilita la toma de decisiones basadas en datos concretos.

5. Adaptación del equipo QA en automatización de pruebas

Uno de los principales retos para la implementación de nuevas herramientas es la curva de aprendizaje del equipo. La transición de un proceso completamente manual a un entorno de pruebas automatizadas requiere formación en metodologías y herramientas especializadas.

- **Propuesta de mejora:** Incursionar al equipo de QA en metodologías de automatización de pruebas, abordando tanto la creación de pruebas como el análisis de resultados.
- **Beneficios esperados:**
    - Mayor eficiencia en la ejecución y documentación de pruebas.
    - Reducción de la curva de aprendizaje en la adopción de nuevas herramientas.
    - Mejor adaptación del equipo a metodologías ágiles y pruebas automatizadas.
    - Incremento en la calidad y confiabilidad del proceso de QA.

La evaluación de las herramientas utilizadas en el proceso de QA ha permitido identificar múltiples oportunidades de mejora. La implementación de un framework de automatización de pruebas, los reportes de pruebas y defectos generados, y su implementación en el equipo permitirán optimizar el proceso de pruebas, reduciendo la carga de trabajo y mejorando la calidad del software.

Estas mejoras permitirán evolucionar hacia un modelo de pruebas más eficiente, basado en la automatización y en la trazabilidad de defectos, facilitando el cumplimiento de los objetivos de calidad del software de manera sostenible y escalable.

5.1.3 Propuesta inicial del proceso de automatización

Con base en el análisis previo, se propone una estrategia inicial de automatización del proceso de pruebas en QA que mejore la ejecución de pruebas, facilite la trazabilidad de defectos y permita medir el impacto de la automatización a lo largo del tiempo.

5.1.3.1 Definición del flujo de trabajo basado en hallazgos del análisis. La propuesta de automatización de pruebas busca estructurar un flujo de trabajo que permita mejorar la calidad del software, optimizar tiempos y apoyar la trazabilidad de las validaciones realizadas. En este sentido, teniendo en cuenta el análisis de la gestión actual de pruebas y los desafíos identificados, se define el siguiente flujo de trabajo, asegurando que la automatización se implemente de manera gradual y controlada.

1. Definición de Casos de Prueba

Antes de iniciar la automatización, es fundamental establecer qué pruebas serán automatizadas y cómo se organizarán para facilitar su efectividad.

- **Selección de escenarios a automatizar**
    - Identificación de funcionalidades críticas que requieren validaciones constantes.
    - Priorización de pruebas repetitivas que representen una alta carga de trabajo en la ejecución manual.
- **Estructuración de los casos de prueba**
    - Definición clara de los pasos y condiciones de cada prueba.
    - Uso de un formato estandarizado para facilitar su implementación en la fase de automatización.
    - Registro de las dependencias necesarias para la ejecución, como datos de prueba y configuraciones específicas.

2. Desarrollo de Pruebas Automatizadas

Esta etapa se enfoca en la implementación gradual de los casos de prueba, asegurando que sean comprensibles, reutilizables y fáciles de mantener.

- **Implementación de los casos de prueba**
  - Desarrollo de las pruebas de manera estructurada para promover su correcto funcionamiento.
  - Organización clara en la escritura de los scripts de prueba.
  - Administración de datos de prueba y configuraciones necesarias.
- **Mantenimiento y actualización de pruebas**
  - Revisión y ajuste de los casos de prueba ante cambios en el sistema.
  - Estrategias para gestionar la evolución de los escenarios automatizados sin afectar la estabilidad de las pruebas existentes.

3. Ejecución de Pruebas

Para mantener un monitoreo efectivo, es necesario establecer estrategias de ejecución que permitan validar el comportamiento del sistema en distintos momentos del desarrollo.

- **Planificación y ejecución de pruebas**
  - Definición de la frecuencia de ejecución según las necesidades del equipo de desarrollo.
  - Ejecución de pruebas en diferentes entornos para evaluar el comportamiento del sistema en distintas condiciones.
- **Captura de resultados y evidencias**
  - Registro de los resultados obtenidos en cada prueba ejecutada.
  - Generación de evidencias que faciliten la identificación de problemas y su posterior análisis.

4. Generación de Reportes de Prueba

Para mejorar la trazabilidad y la toma de decisiones, se establecerá un mecanismo de generación de reportes con información estructurada sobre la ejecución de pruebas.

- **Registro y almacenamiento de resultados**
  - Generación de reportes con el detalle de cada ejecución de prueba.
  - Registro de métricas clave como porcentaje de pruebas exitosas, tiempos de ejecución y fallos detectados.
- **Integración con herramientas de gestión**
  - Inclusión de enlaces a los reportes en la plataforma de gestión de proyectos para facilitar el acceso a los desarrolladores.
  - Mecanismos de comunicación con el equipo para el análisis conjunto de resultados.

5. Registro de Defectos y Seguimiento

El flujo de trabajo debe ayudar a que los defectos detectados sean correctamente documentados y gestionados.

- **Análisis de defectos detectados**
  - Clasificación de los errores según su impacto y criticidad.
  - Identificación de patrones en los defectos para mejorar la calidad del sistema.
- **Priorización y reporte de defectos**
  - Registro de los errores detectados con el detalle necesario para su análisis y corrección.
  - Definición de prioridades en conjunto con el equipo de desarrollo para optimizar la solución de problemas.

Este flujo de trabajo proporciona una estructura clara para la transición hacia la automatización de pruebas, asegurando que se realice de manera controlada y con un impacto positivo en la gestión de calidad del software.

5.1.3.2 Identificación de métricas clave para medir el impacto de la automatización. Para evaluar de manera precisa el impacto de la automatización en el proceso de pruebas, es fundamental seleccionar métricas que permitan comparar de forma objetiva los resultados obtenidos con los métodos manuales y los alcanzados con la implementación de pruebas automatizadas. La elección de estas métricas se basa en la necesidad de medir mejoras en eficiencia, alcance y calidad del proceso de validación de software, asegurando que la automatización aporte un valor real a la estrategia de aseguramiento de calidad.

Las métricas seleccionadas han sido definidas considerando tanto su aplicabilidad dentro del contexto actual del equipo de QA como su capacidad para ofrecer datos cuantificables que faciliten el análisis comparativo. En este sentido, se han identificado tres métricas clave que permitirán evaluar la efectividad de la automatización en relación con las pruebas manuales: tiempo de ejecución, cobertura de casos de prueba y calidad de defectos. Estas métricas proporcionarán una visión integral sobre la optimización del tiempo de validación, la cantidad de escenarios que pueden ser evaluados con la automatización y la capacidad de detectar fallos en el software.

- **Tiempo de Ejecución de Pruebas**

  Esta métrica mide el tiempo que toma ejecutar un conjunto de pruebas automatizadas en comparación con su ejecución manual. Su propósito es determinar en qué medida la automatización reduce el tiempo necesario para validar

el software y cómo impacta en la velocidad de retroalimentación para el equipo de desarrollo.

Por tanto, se registrará el tiempo total empleado en la ejecución de pruebas manuales previamente recopilado y se comparará con los tiempos obtenidos en la ejecución de pruebas automatizadas. La idea es analizar la reducción en el tiempo de ejecución y su impacto en la eficiencia del ciclo de desarrollo, considerando tanto la ejecución individual de pruebas como la ejecución en conjunto dentro del flujo de trabajo.

- **Cobertura de Casos de Prueba**

Esta métrica evalúa la cantidad de escenarios de prueba que pueden ser ejecutados de manera automatizada en comparación con las pruebas manuales. Su objetivo es determinar si la automatización permite validar una mayor cantidad de funcionalidades o las más importantes en menor tiempo, asegurando un mayor alcance en las evaluaciones del software.

Para esto, se comparará el número total de casos de prueba ejecutados manualmente con el número de casos que pueden ser cubiertos con pruebas automatizadas en el mismo período de tiempo. De igual forma, se piensa analizar la capacidad de ejecución simultánea de pruebas, identificando en qué medida la automatización contribuye a ampliar la cobertura de pruebas del sistema sin aumentar la carga de trabajo del equipo.

- **Cobertura de Defectos**

Esta métrica permitirá evaluar la cantidad de defectos detectados mediante pruebas automatizadas en relación con los defectos identificados durante la

ejecución manual, con el fin de analizar si la automatización mejora la detección temprana de fallos y contribuye a la estabilidad del software.

Dado que varios de los defectos identificados en pruebas manuales ya han sido corregidos, la aplicación de esta métrica se enfocará principalmente en los nuevos desarrollos asociados al rediseño del módulo de proveedores del SIF, donde la automatización permitirá validar de forma continua los cambios implementados.

Por lo tanto, se analizará si los casos de prueba automatizados logran identificar defectos que podrían no ser evidentes en la ejecución manual y si su detección temprana permite reducir el esfuerzo de corrección en etapas posteriores del desarrollo.

El seguimiento de estas métricas proporciona datos concretos sobre el impacto de la automatización, lo cual permite realizar ajustes estratégicos en su implementación. Posteriormente, en la sección de resultados, se presentará un análisis detallado de los valores obtenidos junto con representaciones gráficas que faciliten la comparación entre el proceso manual y el automatizado.

5.1.3.3 Propuesta preliminar de integración con herramientas de automatización. La implementación de la automatización de pruebas requiere una integración estructurada dentro del proceso de aseguramiento de calidad, asegurando que las pruebas sean ejecutadas de manera eficiente, los resultados sean documentados adecuadamente y la trazabilidad de los defectos sea óptima. Para lograrlo, es fundamental establecer una estrategia que permita la correcta adopción de la automatización en el flujo de trabajo del equipo, asegurando su alineación con las prácticas de desarrollo y control de calidad existentes.

Esta propuesta preliminar plantea un esquema de integración basado en distintos elementos significativos, que van desde la definición del entorno de automatización hasta el mantenimiento

continuo del proceso. La selección de herramientas específicas se llevará a cabo en fases posteriores del proyecto, pero en esta etapa se establecen los lineamientos generales que guiarán la implementación.

1. **Entorno de Automatización de Pruebas**

   Para la ejecución de pruebas automatizadas, se implementará un entorno de automatización que permita validar funcionalidades de la aplicación de manera estructurada y replicable. Este entorno deberá soportar la ejecución de pruebas funcionales y de regresión, permitiendo que los cambios en el sistema no generen impactos negativos en las funcionalidades ya validadas.

   Además, se definirá un lenguaje para la escritura de escenarios de prueba que permita describir los casos de manera comprensible tanto para testers como para desarrolladores y otros miembros del equipo. Esto facilitará la colaboración y el mantenimiento de los scripts de prueba.

2. **Ejecución y Control de Versiones**

   Las pruebas automatizadas podrán ejecutarse en distintos entornos, desde estaciones de trabajo locales hasta servidores dedicados que permitan la ejecución controlada dentro del flujo de desarrollo. Se establecerá una estrategia de control de versiones que garantice la correcta gestión de los scripts de prueba y facilite su integración en el ciclo de desarrollo.

3. **Generación y Almacenamiento de Reportes**

   Se implementará un sistema de generación de reportes que documente los resultados de las pruebas de manera estructurada, proporcionando métricas relevantes sobre la ejecución y evidencias visuales. Estos reportes permitirán

realizar un análisis detallado del estado del software y facilitarán la toma de decisiones en el proceso de calidad.

Además, se definirá un mecanismo de almacenamiento de reportes que permita la consulta y el análisis histórico de los resultados, asegurando la trazabilidad y la mejora continua del proceso de pruebas automatizadas.

4. **Trazabilidad y Gestión de Defectos**

Para promover la correcta documentación de los defectos detectados durante la ejecución de pruebas automatizadas, se establecerá un proceso de trazabilidad que permita vincular los reportes de pruebas con las incidencias registradas en la herramienta de gestión de proyectos. Esto facilitará el acceso a la información para los desarrolladores y agilizará la identificación y resolución de errores.

El uso de esta herramienta permitirá reducir la necesidad de documentación manual y optimizar el flujo de trabajo en la gestión de defectos, asegurando que cada hallazgo pueda ser analizado con la evidencia correspondiente.

5. **Mantenimiento y Mejora Continua**

Para ayudar con la estabilidad y evolución del entorno de automatización, se establecerá una estrategia de mantenimiento que contemple la revisión periódica de los scripts de prueba, la actualización del entorno ante cambios en la aplicación y la optimización de los procesos de ejecución.

Esta propuesta preliminar servirá como base para la integración de la automatización dentro del proceso de calidad, permitiendo optimizar el tiempo de validación, mejorar la cobertura

de pruebas y ayudar la detección de defectos en el software. En la *figura 1* se detalla el paso a paso:

Figura 1

Flujo inicial del proceso de automatización

5.2 Documentación en la Wiki de la DTIC basada en la metodología BDD (Behavior-Driven Development)

Como parte del proceso de automatización de pruebas, se realizó la documentación en la Wiki de la DTIC basada en la metodología Behavior-Driven Development (BDD), con el fin de consolidar un referente común para el equipo de QA. Esta documentación se orienta tanto a la comprensión conceptual de la metodología como a la adopción progresiva de sus prácticas dentro del proceso de pruebas, sirviendo como punto de partida para estandarizar el diseño de casos y apoyar la implementación del enfoque propuesto en el módulo de proveedores. Para comenzar es necesario abordar conceptos los siguientes conceptos:

5.2.1 Test-Driven Development (TDD)

El Test-Driven Development (TDD), conocido en español como "Desarrollo Guiado por Pruebas", es una práctica iterativa de diseño de software que surgió como parte del enfoque de programación extrema (Extreme Programming, XP). Esta metodología se caracteriza por seguir dos principios fundamentales: primero, escribir pruebas automatizadas antes de desarrollar el código que se probará, y segundo, eliminar cualquier duplicación de código mediante un proceso de refactorización.

Kent Beck (2003), uno de los principales exponentes de esta práctica, define TDD como una herramienta para "manejar el miedo durante la programación", ya que permite a los desarrolladores trabajar con mayor confianza al mostrar que cada pieza de código cumple con su propósito antes de ser implementada.

5.2.1.1 Principios de TDD. Al hacer uso de TDD, se requiere que el desarrollador enfoque su atención en el comportamiento deseado del software (qué hace) antes de preocuparse por su implementación (cómo lo hace). Esto implica comprender a fondo los requisitos antes de escribir cualquier código. Además, la automatización de las pruebas es un pilar central, ya que permite ejecutar constantemente el conjunto completo de pruebas para verificar que cualquier cambio en el código no afecta negativamente a las funcionalidades existentes. Este enfoque iterativo no solo mejora la calidad del código, sino que también facilita su mantenimiento y escalabilidad.

Un aspecto fundamental de TDD es la eliminación de duplicaciones de código, lo que se logra mediante la refactorización continua. Este proceso asegura que el código sea limpio, legible y eficiente. El lema asociado a TDD, "If it’s green, the code is clean" (Si está en verde, el código está limpio), refuerza la idea de que un código funcional y bien diseñado debe pasar todas las pruebas antes de ser considerado completo.

5.2.1.2 Ventajas y desventajas de TDD. TDD ofrece numerosos beneficios en el ciclo de desarrollo de software. En primer lugar, fomenta un diseño orientado al comportamiento y ayuda a los desarrolladores a centrarse en los objetivos del sistema antes de abordar detalles técnicos. En segundo lugar, la automatización de pruebas asegura que los errores se detecten temprano en el proceso de desarrollo, reduciendo los costos asociados con correcciones tardías. Finalmente, al promover la refactorización constante, TDD ayuda a que el código sea de alta calidad y esté alineado con los principios de diseño limpio.

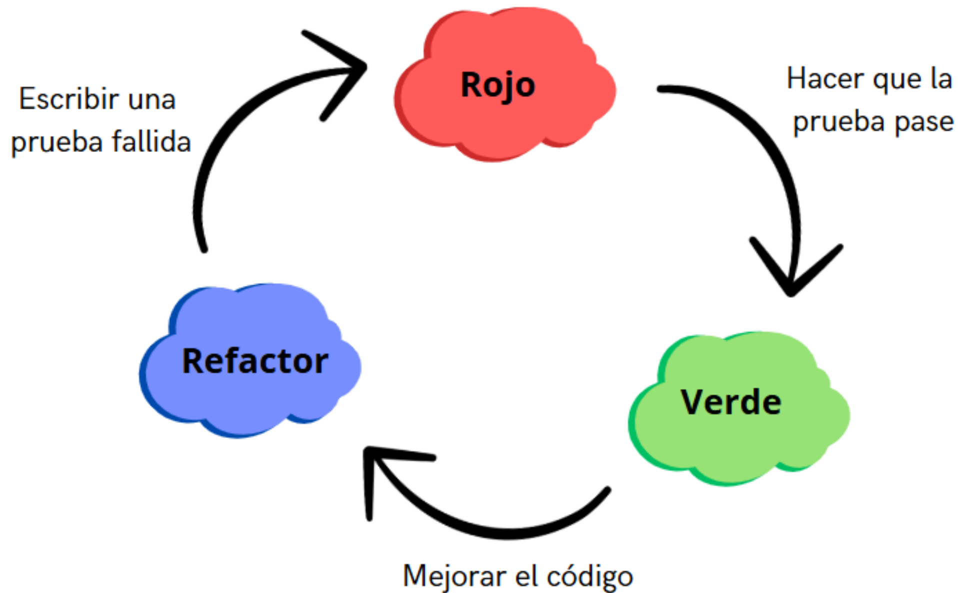
Aunque TDD ofrece múltiples beneficios, también presenta ciertas desventajas. Por un lado, implementar TDD puede ser inicialmente costoso en términos de tiempo y esfuerzo, especialmente para desarrolladores sin experiencia previa en la metodología. Además, escribir pruebas antes de codificar requiere un entendimiento profundo de los requisitos y puede ser un desafío en proyectos con especificaciones poco claras o cambiantes. Por último, si las pruebas no se actualizan correctamente durante el mantenimiento, pueden volverse obsoletas y perder su efectividad, dificultando la evolución del software.

5.2.1.3 Ciclo iterativo de TDD. El ciclo de trabajo en TDD, conocido como "Rojo-Verde-Refactorización", puede explicarse de la siguiente manera, y visualizarse en la *Figura 2*:

- **Fase roja:** El desarrollador escribe una prueba que inicialmente falla, ya que no existe el código que debe pasarla.
- **Fase verde:** Se desarrolla el código mínimo necesario para que la prueba pase. En esta etapa, el enfoque está en cumplir los requisitos funcionales.
- **Refactorización:** Una vez que la prueba es superada, el código se optimiza y se limpia para mejorar su calidad, asegurando que sea más legible y sostenible, pero sin alterar su funcionalidad (Beck, 2003).

Figura 2

Ciclo iterativo de TDD



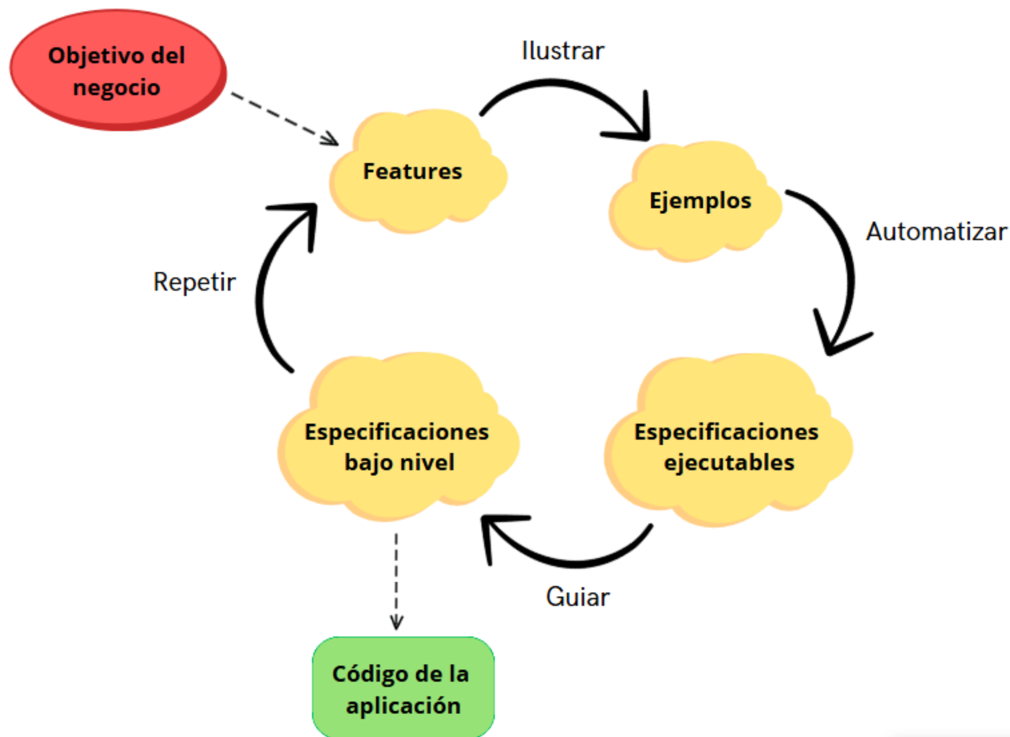
Nota. Adaptado de *TDD: contexto*, por Gonzalez, A. G. (2022, junio 1), Blog de hiberus. https://www.hiberus.com/crecemos-contigo/todo-lo-que-necesitas-saber-de-tdd-en-3-minutos/

5.2.2 Behavior-Driven Development (BDD)

Behavior-Driven Development (BDD) es una metodología ágil que surgió como una evolución del Test-Driven Development (TDD), con el propósito de mejorar la comunicación entre desarrolladores, testers y clientes en el proceso de desarrollo de software (North, 2009). A través de la utilización de un lenguaje natural estructurado, BDD permite describir el comportamiento esperado del sistema mediante ejemplos concretos, lo que facilita la alineación entre los requisitos del negocio y la implementación técnica (Smart, 2017). Un ejemplo ilustrativo sobre las actividades realizadas en BDD, se pueden ver en la *Figura 3*:

Figura 3

Actividades realizadas en BDD



Nota. Adaptado de *Prototype*, por MBA Skool (s. f.). https://www.mbaskool.com/business-concepts/it-and-systems/13712-prototype.html

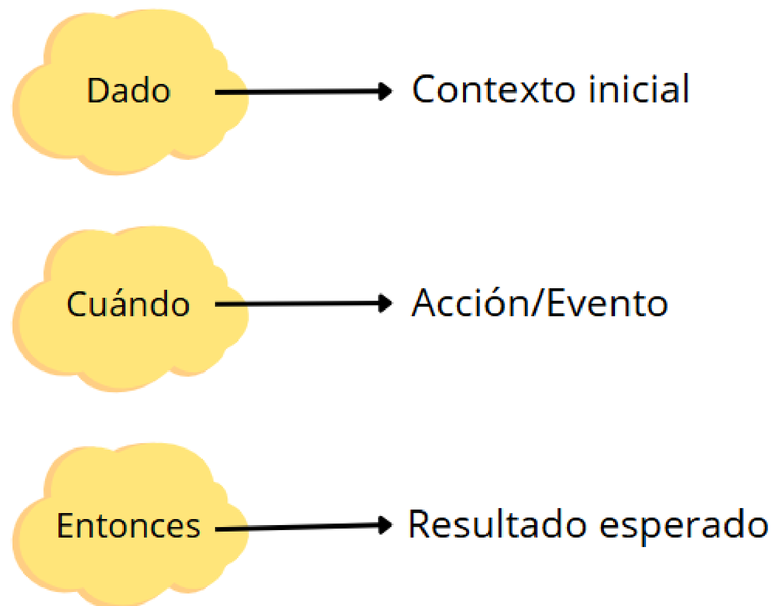
5.2.2.1 Lenguaje Gherkin. A diferencia de otras metodologías, BDD no se centra únicamente en la ejecución de pruebas, sino en la definición colaborativa del comportamiento del software. Esto se logra mediante la especificación de escenarios de uso en un formato Given-When-Then, llamado Gherkin, que posteriormente pueden ser automatizados con distintas herramientas (Keogh, 2014). Los escenarios en Gherkin siguen tres fases, mostrado enseguida y evidencias en la Figura 4:

1. **Preparación del contexto**: Se usa *Given* para definir el estado inicial.
2. **Ejecución de la acción**: Se usa *When* para representar la acción a evaluar.

3. **Verificación del resultado**: Se usa *Then* para comprobar la salida esperada.

Figura 4

Pasos Gherkin



Nota. Adaptado de *Given-When-Then pattern in unit tests*, por Starzyk, M. (2017, abril 3), J-Labs. https://www.j-labs.pl/en/tech-blog/given-when-then-pattern-in-unit-tests/

Además, al ser Gherkin un lenguaje utilizado para escribir escenarios en pruebas automatizadas, presenta una sintaxis estructurada con palabras clave específicas adicionales a las presentadas anteriormente (Matsinopoulos, 2020):

- **Feature:** Define la funcionalidad a probar con un título y una descripción opcional.
- **Background:** Contiene pasos comunes que deben ejecutarse antes de cada escenario.
- **Scenario:** Representa una prueba específica con una estructura clara.

- **Scenario Outline:** Permite definir escenarios reutilizables con diferentes datos.
- **Examples:** Lista de valores utilizados en un Scenario Outline.
- **And / But / *:** Permiten mejorar la legibilidad sin cambiar la ejecución.

El siguiente ejemplo muestra cómo estructurar un escenario en Gherkin para probar la actualización de una foto de perfil. Se utiliza la palabra clave Background para definir pasos comunes que deben ejecutarse antes de cada escenario, evitando la repetición innecesaria. Luego, se describe un Scenario específico donde el usuario sube una nueva foto de perfil, siguiendo la estructura de Given (contexto inicial), When (acción) y Then (resultado esperado).

```
Feature: Actualización de foto de perfil
  Background:
       Given El usuario ha iniciado sesión
       And Accede a la sección de perfil
        Scenario: Subir una nueva foto
             When Hace clic en la foto de perfil actual
             Then Aparece un área para cargar la nueva foto
```

5.2.3 Principios de BDD

BDD se fundamenta en una serie de principios que guían su aplicación en el desarrollo de software. Estos principios no solo establecen las bases para la metodología, sino que también buscan fomentar la colaboración y la alineación entre los distintos actores involucrados en el proceso. Al seguir estos lineamientos, los equipos pueden orientar a que los requisitos del negocio se traduzcan correctamente en software funcional y probado.

- **Colaboración entre roles:** Uno de los pilares fundamentales de BDD es la comunicación efectiva entre desarrolladores, testers y stakeholders (Beck, 2003). A diferencia de enfoques tradicionales donde cada rol trabaja de forma aislada, en

BDD se fomenta la participación activa de todos los involucrados para definir el comportamiento del sistema desde múltiples perspectivas.

- **Definición clara del comportamiento esperado:** BDD se basa en la creación de escenarios que describen cómo debe reaccionar el sistema ante diferentes condiciones. Estos escenarios utilizan ejemplos específicos que ayudan a eliminar ambigüedades en los requerimientos y a orientar que el software cumpla con las expectativas del usuario final (Wynne & Hellesoy, 2017).
- **Uso de lenguaje natural:** A diferencia de otras metodologías donde las pruebas pueden ser difíciles de comprender para personas ajenas al desarrollo, BDD utiliza formatos estructurados como Gherkin, que permiten escribir los escenarios en lenguaje natural (Astels, 2003). Esto facilita la comprensión de los casos de prueba y promueve la colaboración interdisciplinaria.
- **Pruebas como documentación viviente:** Los escenarios de BDD no solo sirven como pruebas automatizadas, sino que también actúan como documentación actualizada del sistema. Esto permite que cualquier miembro del equipo, incluso sin conocimientos técnicos, pueda comprender el comportamiento del software a lo largo del tiempo (Marick, 2003).
- **Automatización y ejecución continua:** Una vez definidos los escenarios de prueba, estos pueden ser convertidos en pruebas automatizadas que se ejecutan de manera recurrente dentro del ciclo de desarrollo. Esto permite detectar errores de manera temprana y promover que el software mantenga su calidad a lo largo del tiempo (Meszaros, 2007).

5.2.4 Ventajas y desventajas de BDD

La adopción de BDD en el desarrollo de software ofrece múltiples beneficios, tanto en términos de calidad del producto como en la eficiencia del equipo de trabajo. Al centrarse en la definición del comportamiento del sistema desde el inicio, BDD permite reducir la cantidad de defectos en producción y acompañar a que las funcionalidades entregadas sean las esperadas por los usuarios.

- **Mejor comunicación entre equipos:** Al utilizar un lenguaje común y accesible para todos los actores involucrados, BDD facilita la colaboración entre desarrolladores, testers y stakeholders. Esto reduce malentendidos en los requerimientos y mejora la alineación entre la visión del negocio y la implementación técnica (Gherkin, 2015).
- **Reducción de defectos en producción:** La especificación de escenarios detallados y su automatización permiten detectar errores desde etapas tempranas del desarrollo, lo que disminuye la cantidad de fallos en producción y reduce los costos de corrección de errores tardíos (Crispin & Gregory, 2009).
- **Desarrollo enfocado en el usuario:** Al definir el comportamiento del sistema desde la perspectiva del usuario final, BDD ayuda a que las funcionalidades entregadas realmente satisfagan las necesidades del negocio y brinden una mejor experiencia al usuario (Adzic, 2011).
- **Facilita la automatización de pruebas:** Al estructurar los escenarios de prueba en formato Given-When-Then, BDD permite convertir estos casos en pruebas automatizadas de manera más sencilla, lo que mejora la eficiencia en el proceso de aseguramiento de calidad (Freeman & Pryce, 2010).

- **Código más mantenible y comprensible:** La integración de BDD con procesos de refactorización y documentación viviente permite que el código y los requisitos del sistema evolucionen de manera alineada, evitando la degradación de la calidad del software a lo largo del tiempo (Fowler, 2018).

A pesar de sus múltiples beneficios, BDD también presenta ciertos desafíos y limitaciones que deben ser considerados antes de su implementación. La correcta adopción de esta metodología requiere un cambio en la cultura del equipo de trabajo, así como una inversión inicial en capacitación y herramientas adecuadas.

- **Curva de aprendizaje elevada:** La implementación efectiva de BDD requiere que los equipos comprendan no sólo la metodología, sino también las herramientas necesarias para automatizar pruebas y estructurar escenarios correctamente. Esto puede generar una barrera de entrada para equipos sin experiencia en enfoques ágiles (Rahman, 2019).
- **Mayor esfuerzo en la fase inicial:** A diferencia de metodologías tradicionales, BDD requiere una inversión de tiempo considerable en la definición de escenarios y en la colaboración entre diferentes roles. Aunque esto puede traducirse en una mejor calidad del software a largo plazo, inicialmente puede ralentizar el desarrollo (Humble & Farley, 2010).
- **Dependencia de una comunicación efectiva:** Para que BDD funcione correctamente, es esencial que los stakeholders, testers y desarrolladores participen activamente en la definición de los escenarios de prueba. Si la comunicación entre estos actores es deficiente, la metodología puede volverse ineficiente y generar ambigüedades en los requerimientos (Pichler, 2012).

- **Posible sobrecarga de documentación:** Aunque los escenarios de BDD actúan como documentación viviente, si no se gestionan adecuadamente pueden volverse extensos y difíciles de mantener. Es necesario establecer buenas prácticas para evitar la acumulación de información redundante o desactualizada (Coplien & Bjørnvig, 2010).
- **No es adecuado para todos los proyectos:** En sistemas altamente técnicos o con requerimientos que cambian constantemente, BDD puede no ser la metodología más eficiente. Su enfoque basado en escenarios estructurados puede resultar demasiado rígido para ciertos tipos de desarrollo (Bosch, 2014).

5.2.5 Beneficios de la metodología BDD en la automatización

La automatización de pruebas es un componente importante en el desarrollo de software moderno, ya que permite validar funcionalidades de manera eficiente y continua. La metodología Behavior-Driven Development (BDD) se ha consolidado como una estrategia efectiva para optimizar este proceso, integrando la colaboración entre equipos, la definición clara de requisitos y la generación de pruebas automatizadas basadas en el comportamiento esperado del software.

BDD facilita la automatización al permitir que los escenarios escritos en lenguaje natural sean interpretados directamente por herramientas de pruebas automatizadas. Esto no solo mejora la precisión de las pruebas, sino que también asegura que los requisitos del negocio se reflejan fielmente en el desarrollo y validación del software.

El uso de BDD en la automatización de pruebas ofrece múltiples ventajas que impactan tanto en la calidad del software como en la eficiencia del desarrollo. A continuación, se presentan los beneficios más relevantes.

- **Mejora la comunicación y la comprensión de requisitos:** BDD promueve un lenguaje común entre desarrolladores, testers y stakeholders a través del uso de escenarios en lenguaje natural. Esto reduce la ambigüedad en la definición de los requisitos y facilita una mejor comprensión del comportamiento esperado del software (Pichler, 2012).
- **Facilita la automatización de pruebas desde el inicio:** A diferencia de otros enfoques, BDD integra la automatización de pruebas desde las primeras etapas del desarrollo. Los escenarios definidos con su lenguaje natural pueden ser convertidos directamente en pruebas automatizadas con otras herramientas, lo que agiliza la validación y reduce el esfuerzo manual en las pruebas (Rahman, 2019).
- **Asegura la trazabilidad de las pruebas:** La metodología BDD permite un seguimiento claro desde los requisitos hasta la ejecución de las pruebas automatizadas. Esto facilita la detección temprana de errores y permite que los cambios en el código no rompan funcionalidades previamente implementadas.
- **Reduce el costo de mantenimiento de las pruebas:** Gracias a la estructura modular de los escenarios en BDD, las pruebas pueden ser fácilmente reutilizadas y mantenidas sin necesidad de reescribir grandes volúmenes de código. Esto es fundamental en proyectos a largo plazo, donde la evolución del software requiere constantes ajustes en las pruebas automatizadas (Meszaros, 2007).
- **Genera una gran documentación:** Los escenarios BDD sirven como documentación funcional del sistema, ya que describen el comportamiento esperado del software en un formato estructurado y fácil de leer. A diferencia de la

documentación tradicional, que tiende a quedar obsoleta, los escenarios BDD se actualizan en paralelo con cada cambio en el código (Freeman & Pryce, 2010).

- **Mejora la cobertura de pruebas y la detección de errores:** Al enfocarse en el comportamiento del usuario final, BDD permite cubrir una amplia gama de escenarios en las pruebas automatizadas, incluyendo pruebas funcionales, de regresión y de aceptación. Lo anterior, mejora la calidad del software y minimiza el riesgo de defectos en ambientes como preproducción y producción.
- **Facilita la integración con metodologías ágiles y DevOps:** BDD se alinea perfectamente con prácticas ágiles y entornos DevOps, ya que permite realizar pruebas de forma rápida y eficiente en cada iteración del desarrollo. BDD se alinea perfectamente con prácticas ágiles y entornos DevOps, ya que permite realizar pruebas de forma rápida y eficiente en cada iteración del desarrollo. Esto acelera el time-to-market del software y asegura entregas continuas de alta calidad (Wynne & Hellesoy, 2017).

5.2.6 Comparación entre BDD y TDD

BDD no es una metodología completamente nueva, sino una evolución que incorpora las mejores prácticas de TDD y amplía su alcance al involucrar a todos los miembros del equipo de desarrollo. De hecho, en sus inicios, BDD surgió como una mejora conceptual de TDD, reemplazando la palabra "test" por "should" (debería), enfatizando así la validación del comportamiento esperado de la aplicación en lugar de centrarse en pruebas unitarias aisladas (Wynne & Hellesoy, 2017).

Si bien tanto BDD (Behavior-Driven Development) como TDD (Test-Driven Development) se basan en pruebas para guiar el desarrollo de software, sus enfoques y objetivos

son distintos. TDD se centra en la implementación técnica, donde los desarrolladores escriben pruebas unitarias antes de codificar la funcionalidad, asegurando que cada componente individual funcione correctamente. En cambio, BDD prioriza el comportamiento del sistema desde la perspectiva del usuario final, definiendo escenarios en lenguaje natural para que el software cumpla con los requisitos del negocio.

Otra diferencia relevante se presenta en la colaboración dentro del equipo. Mientras que, TDD es una práctica orientada principalmente a desarrolladores, quienes escriben y ejecutan pruebas unitarias de manera aislada. Por su parte, BDD fomenta la participación de diferentes roles, como analistas de negocio, testers, desarrolladores y stakeholders, promoviendo una mejor comunicación y alineación con los objetivos del sistema.

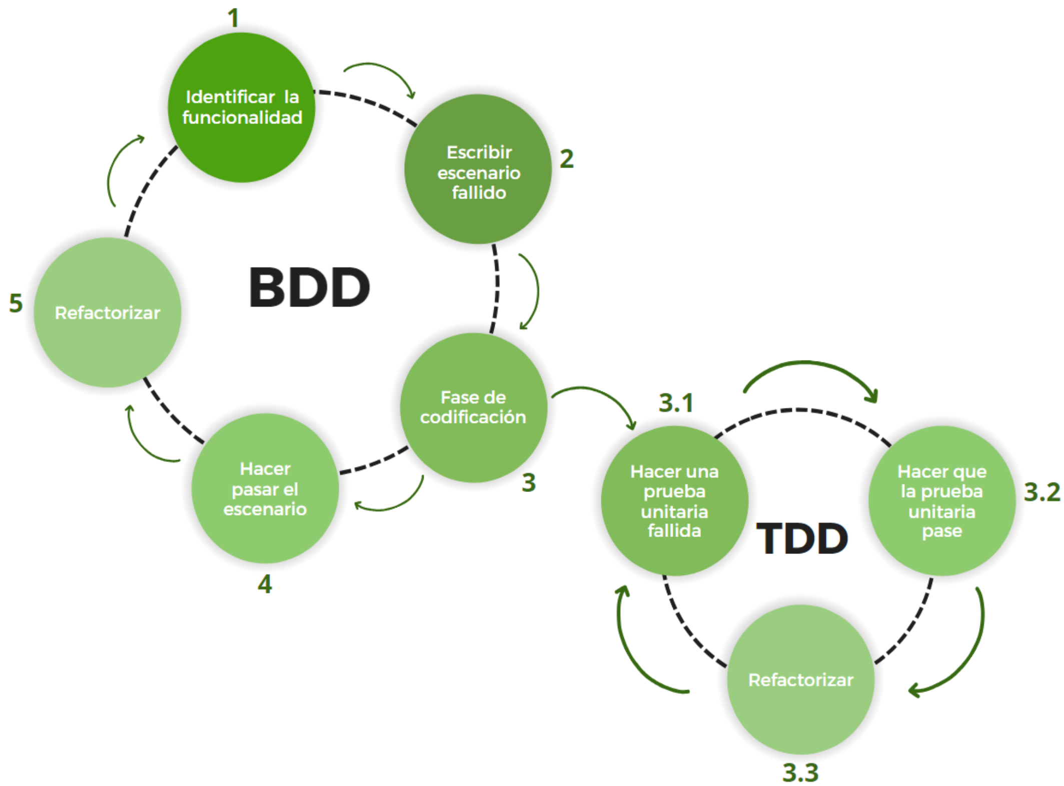
El lenguaje y las herramientas utilizadas en cada metodología también marcan una diferencia importante. TDD emplea lenguajes de programación convencionales junto con frameworks como JUnit, NUnit o PyTest (Meszaros, 2007), para la creación de pruebas unitarias. BDD utiliza un lenguaje estructurado basado en el formato Given-When-Then (como Gherkin) y herramientas como Cucumber o SpecFlow (Rahman, 2019), lo que facilita la comprensión y validación de los escenarios por parte de todos los involucrados.

En cuanto a su ámbito de aplicación, TDD se enfoca en pruebas unitarias, verificando el correcto funcionamiento de módulos individuales antes de su integración (Humble & Farley, 2010). Por otro lado, BDD abarca pruebas funcionales, de aceptación e integración, asegurando que las funcionalidades se alineen con los requisitos del negocio y la experiencia del usuario.

En la práctica, los desarrolladores pueden seguir aplicando TDD dentro de un marco más amplio de BDD, combinando lo mejor de ambos enfoques para mejorar la calidad del software. Lo anterior se puede observar en la *Figura 5* presentada a continuación:

Figura 5

BDD y TDD complementarias



Nota. Adaptado de *BDD vs TDD. Diferencias y aplicaciones*, por Molina, A. (2023, enero 3), Blog de hiberus. https://www.hiberus.com/crecemos-contigo/bdd-vs-tdd-diferencias-y-aplicaciones/

5.2.7 Estrategias para la integración de BDD en el ciclo de desarrollo ágil

Dentro de los marcos ágiles más utilizados, Scrum se ha consolidado como un estándar en la industria. La integración de BDD en Scrum permite mejorar la definición de los requisitos, automatizar pruebas funcionales y apoyar a que el producto entregado realmente refleje las necesidades del negocio.

En la metodología Scrum, BDD se puede contemplar al incorporar pruebas automatizadas en cada sprint, ayudando a la constante evaluación y realización de mejoras del producto, y a su

vez, reduciendo la extensa documentación manual, ya que estas pruebas generan reportes precisos de forma inmediata. Adicionalmente, gracias a la velocidad de ejecución resultado de la automatización, permite la identificación y corrección de defectos de manera temprana al incorporar nuevas funcionalidades sin comprometer la estabilidad del sistema por un periodo de tiempo crítico, asegurando el cumplimiento de las entregas con los estándares de calidad propuestos y los cambios sugeridos por el cliente.

En este sentido, la integración de BDD en Scrum mejora la comunicación dentro del equipo y permite que el desarrollo esté alineado con las necesidades del negocio. Para lograrlo, es importante aplicar estrategias en cada fase del ciclo ágil que permitan definir, validar y refinar los requisitos de manera clara y colaborativa.

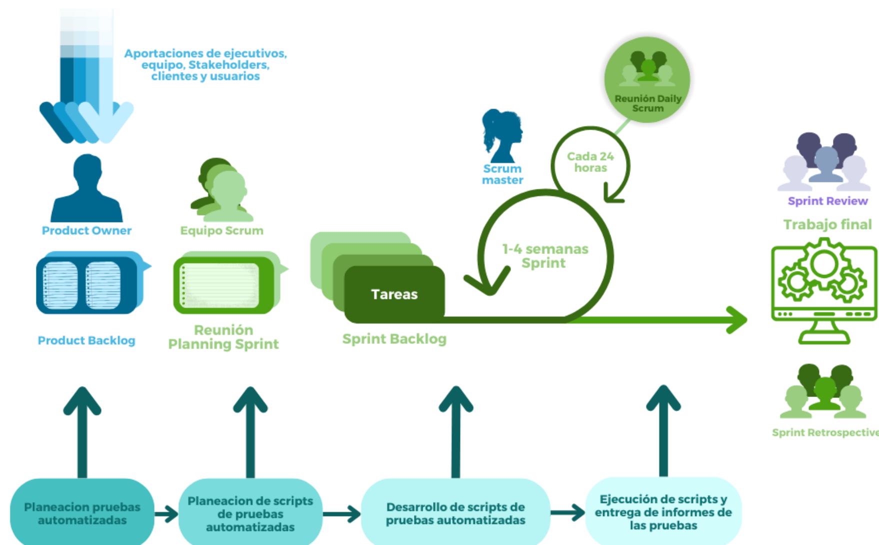
- **Claridad en los requisitos:** Desde la definición de las Historias de Usuario, es útil describir el comportamiento esperado del sistema con escenarios bien estructurados. Esto ayuda a evitar ambigüedades y a que todos los involucrados tengan una visión clara de lo que se debe desarrollar.
- **Colaboración en la planificación:** Durante las reuniones de planeación, el equipo trabaja en conjunto para detallar las historias con ejemplos específicos. Esto permite alinear expectativas, reducir malentendidos y mejorar la comprensión de los requisitos antes de iniciar el desarrollo.
- **Validación durante el desarrollo:** A medida que avanza el Sprint, el equipo se asegura de que cada funcionalidad cumpla con los criterios definidos previamente. Verificar el comportamiento del sistema en cada entrega ayuda a detectar errores de forma temprana y a realizar ajustes cuando sea necesario.

- **Revisión del trabajo realizado:** En la Sprint Review, los escenarios definidos previamente sirven como referencia para validar que el resultado final cumple con lo esperado. Esto facilita la demostración del producto y permite recibir retroalimentación clara y objetiva.
- **Mejora continua:** Durante la Retrospectiva, el equipo analiza cómo ha sido la aplicación de BDD y qué aspectos pueden optimizarse en los próximos sprints. Identificar oportunidades de mejora permite fortalecer la colaboración y hacer más eficiente el proceso de desarrollo.

En la *Figura 6*, se representa la metodología Scrum con todo el enfoque descrito anteriormente.

Figura 6

Integración de BDD en el ciclo de desarrollo ágil



Nota. Adaptado de *Why companies are adopting Agile vs Waterfall*, por Patil, G. (2017, abril 14), Mainpageinc. https://www.mainpageinc.com/single-post/2017/04/13/why-companies-are-adopting-agile-vs-waterfall

5.2.8 Documentación final de BDD en la wiki de la DTIC

Con el fin de promover la transferencia de conocimiento y la adopción sostenible de la metodología BDD (Behavior-Driven Development) dentro del equipo de QA de la DTIC, se documentaron en la wiki institucional los conceptos, fundamentos y prácticas necesarias para su comprensión e implementación.

Esta documentación se construyó a partir de la investigación realizada en las secciones previas (definiciones, principios, beneficios, comparación con TDD e integración con enfoques ágiles), y se aterrizó a un contexto operativo orientado al proceso de automatización de pruebas del SIF, especialmente el módulo de proveedores.

La documentación en la wiki se organizó en dos componentes complementarios (Como el acceso al contenido es privado, podrá consultarse con la debida autorización de la DTIC):

1. **Guía teórica de BDD:** orientada a presentar BDD de forma estructurada y entendible, incluyendo definiciones, componentes (lenguaje ubicuo, criterios de aceptación, escenarios), sintaxis Gherkin, buenas prácticas, ejemplos y referencias. Su objetivo es que cualquier integrante del equipo (incluyendo nuevos miembros o actores externos) pueda comprender BDD, su propósito y su aplicación en el ciclo de vida del software. Una parte de la estructura general de la guía se presenta en la *Figura 7*.

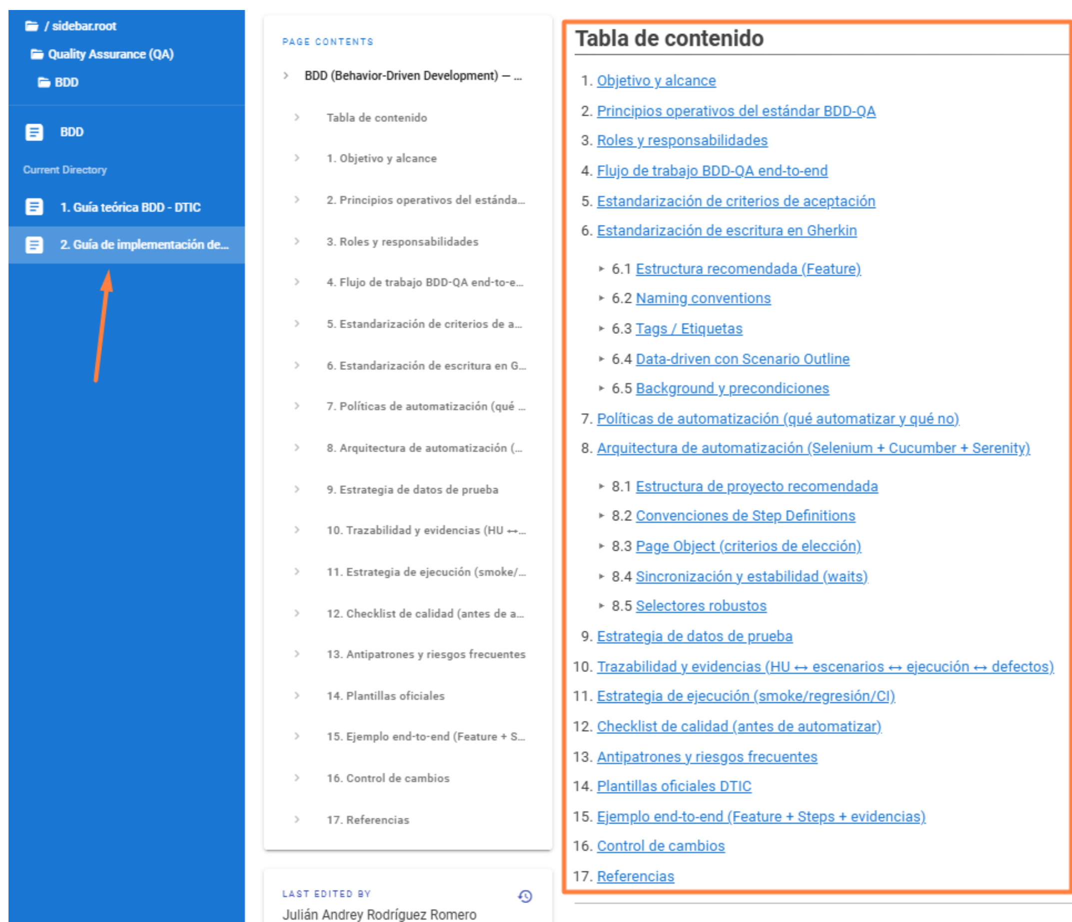
Figura 7

Estructura en la Wiki de la guía teórica BDD

2. **Guía de implementación de BDD en el proceso de QA y automatización:** enfocada en estandarizar cómo se aplicará BDD dentro del flujo de trabajo del equipo. Este componente define convenciones para la escritura de escenarios, criterios mínimos de calidad, estructura de artefactos, uso de etiquetas, lineamientos para la automatización, y prácticas recomendadas para apoyar la mantenibilidad y consistencia. La estructura de la guía se evidencia en la *Figura 8*.

Figura 8

Estructura en la Wiki de la guía de adopción de BDD en el equipo de QA



De esta forma, la Wiki de la DTIC queda establecida como un punto de referencia para la consulta y mantenimiento de la información relacionada con la metodología BDD y su aplicación en el proceso de pruebas automatizadas. La documentación allí consignada permite unificar criterios dentro del equipo de QA, facilitar la comprensión del proceso por parte de nuevos integrantes y dar continuidad a las prácticas definidas, sirviendo como soporte para la evolución y sostenibilidad del proceso de automatización en los módulos del SIF.

5.3 Estructura y diseño de las pruebas automatizadas

En esta sección se aborda la definición de la estructura y el diseño de las pruebas automatizadas, a partir de la selección técnica de las herramientas y tecnologías empleadas, así como del diseño del framework de automatización. Adicionalmente, se establece el proceso que guía la construcción, ejecución y mantenimiento de las pruebas automatizadas, ayudando a su alineación con los objetivos de calidad del proyecto y con el enfoque de pruebas definido para el módulo de proveedores.

5.3.1 Selección de herramientas de automatización

El primer paso para la implementación de pruebas QA automatizadas consiste en la selección adecuada de las herramientas de automatización, las cuales permiten simular las interacciones del usuario y verificar que las funcionalidades del sistema operen correctamente bajo distintos escenarios (Digital.ai, s. f.). En este sentido, es necesario analizar y seleccionar herramientas de pruebas funcionales que faciliten la programación y ejecución de pruebas automatizadas, que sean compatibles con el lenguaje Java utilizado para el desarrollo del Sistema de Información Administrativo (UISARD), y que además permitan la generación de reportes y la adopción de la metodología BDD.

De manera complementaria, también se deben considerar otras herramientas de apoyo al proceso que, si bien no están directamente orientadas a la automatización de pruebas, resultan fundamentales para el desarrollo del proyecto. Entre ellas se encuentra GitLab, utilizado como sistema de control de versiones del código fuente, y los entornos de desarrollo integrados donde se configuran los frameworks y librerías de automatización.

Adicionalmente, se recomienda hacer uso del mismo Entorno de Desarrollo Integrado (IDE) que se utiliza en la DTIC, IntelliJ IDEA Community, debido a su soporte nativo para Maven,

herramienta de gestión de dependencias que permite estructurar, construir y documentar proyectos de software a partir del archivo POM en cualquier proyecto (Apache Maven, s. f.).

5.3.1.1 Entornos de automatización. Un entorno de automatización es una integración de herramientas, frameworks y dependencias compatibles entre sí que contribuyen a la ejecución de estas pruebas automatizadas proporcionando métodos, funciones, patrones de diseño y hasta la generación de informes. Además, es importante mantener un desarrollo ágil y confiable implementando metodologías como lo es BDD.

Identificación de frameworks candidatos para pruebas automatizadas

Este es un paso relevante en la estrategia de las pruebas automatizadas. Para ello, se deben evaluar factores como la repetitividad de las pruebas, la compatibilidad con herramientas de automatización, lenguajes de programación y navegadores requeridos para las pruebas en DTIC. Un framework de automatización de pruebas es una colección de herramientas y directrices las cuales sirven para el diseño, la creación y ejecución de casos de prueba (QAlified, s. f.). Los más destacados son:

- **Playwright:** es una herramienta de automatización hecha para realizar pruebas end-to-end en páginas web, dispositivos móviles y APIs (Sngular, s. f.).
- **Selenium:** el framework selenium es un conjunto de herramientas de código abierto para la automatización de pruebas, la cual permite escribir scripts de prueba en diferentes lenguajes de programación como Java, C#,Python, entre otros.Además, estas pruebas se pueden ejecutar en diferentes sistemas operativos y variedad de navegadores web como Firefox, Chrome, Edge y Safari a través de la suite de herramientas de Selenium, donde se incluye Selenium WebDriver, el motor

principal para la creación de pruebas automatizadas en aplicaciones web (QAlified, s. f.).

- **Cypress:** es una herramienta que ocupa un lugar importante entre las pruebas front-end para aplicaciones web. Es especialmente útil para proyectos que utilizan frameworks de frontend como React y Angular. Cypress permite realizar pruebas end-to-end, unitarias y de integración, además de contar con una interfaz intuitiva (TIVIT, s. f.).
- **TestCafe:** es un marco de pruebas de extremo a extremo fácil de usar. Se destaca por ser un ejecutor de pruebas gratuito y de código abierto. También, es una potente aplicación de escritorio, utilizada frecuentemente en servicios web de calidad empresarial (DevExpress, s. f.).

Comparación de frameworks mediante criterios definidos

Al entrar a comparar los frameworks de automatización lo principal a valorar, además de su uso gratuito, es la compatibilidad con el lenguaje java presente en el código del SIF, con base en esto se descartaron herramientas como Testcafe y Cypress las cuales están pensadas para trabajar con lenguajes como javascript o typescript, también se descartaron de inicio herramientas como appium ya que tiene un uso ideal para aplicativos móviles. Esto se evidencia en la *Tabla 2*.

Tabla 2

Criterios definidos para la comparación y elección de un framework de automatización

	Playwright	Selenium	Cypress	Testcafe	Appium
Lenguajes	Javascript, python, java, C# y .NET	Java, Python, C#, Ruby, Javascript, PHP, etc	Javascript, Typescript	Javascript, Typescript	Java, Python, C#, Javascript, Ruby
Navegadores	Integrado para Chrome y Firefox	Chrome, Firefox, Edge y Safari mediante web drivers	Todos los navegadores a excepción de IE y safari	Cualquier navegador	Chrome, Opera, Edge y Safari mediante web drivers
Herramientas de automatización	Compatible con herramientas basadas en BDD y automatización	Compatible con herramientas basadas en BDD y automatización	Compatible con herramientas basadas en BDD y otras herramientas de automatización	Compatible con frameworks de automatización pero no con herramientas BDD	Compatible con herramientas basadas en BDD y automatizació n
Código abierto	Código abierto desarrollado por microsoft	Código abierto por la comunidad SeleniumHQ	Código abierto de MIT	Código abierto desarrollada por DevExpress	Código abierto apache
Webdrivers	No requiere webdrivers ya que tiene integrado un par de navegadores	Si requiere instalación de webdrivers	No debido a al uso de su propio motor	No ya que usa un proxy para la comunicación con navegadores	Si requiere instalación de webdrivers
Rendimiento	Gran eficiencia debido a su arquitectura	Puede ser lento	Tiene un gran rendimiento ya que se ejecuta en el mismo ciclo de eventos que el navegador	Rápido pero más lento que cypress en las pruebas masivas	Es lento debido a uso de emuladores
Depuración	Avanzada con capturas y videos	Requiere complementos para depuración avanzada	Uso de herramientas con inspección en tiempo real	Buena depuración pero menor a cypress	Complejo debido al uso de emuladores
Comunidad	Comunidad en crecimiento	Gran comunidad, bastante establecida lo que permite mayor documentación e interacción con la	Amplia comunidad además de muy activa	Comunidad en crecimiento	Amplia y popular pero enfocada a pruebas móviles

		herramienta			
Ventajas	Mayor rendimiento a selenium además de compatibilidad con lenguajes y herramientas bdd	Compatibilidad con lenguajes, navegadores, herramientas bdd y gran comunidad	Facilidad en su configuración además de la extensa comunidad	No requiere web drivers y un soporte para múltiples navegadores	Gran integración con muchas herramientas
Desventajas	No compatible con los navegadores necesarios, comunidad pequeña	Necesidad de webdrivers, ejecución lenta a comparación de cypress y playwright, ejecución detallada para tareas sencillas	Baja compatibilidad con navegadores	Menos extenso en cuanto a plugins y personalización	Requieren web drivers y emuladores además de una difícil configuración y uso enfocado a aplicaciones móviles

Selección del framework de automatización y justificación técnica

La elección más consistente en este caso para la realización de pruebas automatizadas es el framework Selenium que incluye la herramienta selenium webdriver con la que se puede tener compatibilidad con todos los navegadores existentes. Adicionalmente, es el framework más usado en java y cuenta con un código abierto lo cual permite tener bastante documentación. No obstante, lo que más destaca es su fácil integración con otras herramientas de automatización, permitiendo mayor flexibilidad al escribir pruebas en un lenguaje natural y documentar detalladamente cada caso de prueba.

5.3.1.2 Herramientas de generación de informes. En la actualidad existen variedades de herramientas para la generación de reportes las cuales son compatibles con librerías y frameworks para la automatización de pruebas funcionales con el objetivo de que todo se documente inmediatamente se realizan las pruebas.

Identificación de herramientas para generación de reportes

Entre las herramientas existentes se debe identificar la más adecuada para el contexto DTIC. Por ello, se presentan las más destacadas a continuación:

- **Serenity BDD:** Serenity es una librería de código abierto para la automatización de pruebas cuyo principal objetivo al generar reportes es documentar pruebas automatizadas mediante "Living documentation", así que no es solo una herramienta de reportes, sino también un framework para la automatización de prueba. Serenity utiliza las pruebas para generar informes que documentan resultados como aprobados o fallidos (Serenity BDD, s. f.).
- **Extent Report:** es una librería de código abierto que genera informes de pruebas, se integra con Selenium WebDriver, Cucumber, Java, entre otros, y permite la entrega de reportes con información sobre los casos de prueba automatizados (Vitelli, s. f.).
- **Allure Report:** es una herramienta popular para visualizar los resultados de una prueba automatizada. Produce informes que pueden ser leídos por todos, sin necesidad de conocimientos técnicos (Allure Report, s. f.).

Comparación de herramientas de informes mediante criterios definidos

Para el equipo de QA una parte importante de la ejecución de pruebas es la documentación de estas mismas, por lo que, al hablar de automatización de pruebas, la elección de una herramienta de reportes que permita la documentación continua y sin requerir un gran esfuerzo debe ser una prioridad, ya que el potencial de reducción de tiempo puede ser alto. Por lo mencionado anteriormente, la herramienta Serenity destaca por su funcionalidad "living documentation", sin embargo, herramientas como Extent reports y allure report que su único rol es la generación de

reportes, se caracterizan más en aspectos como la personalización de cada reporte y su interfaz gráfica.

Tabla 3

Criterios definidos para la comparación y elección de una herramienta de generación de informes

	Serenity BDD	ExtentReports	Allure Report
Lenguajes	Java, Gherkin	Java, .NET, JavaScript	Java, Python, JavaScript
Soporte nativo con BDD	Permite escribir pruebas en Gherkin y ejecutarlas fácilmente.	No implementa BDD debido a que es una herramienta de reportes	Es una herramienta que se puede integrar con cucumber pero no tiene soporte nativo
Herramientas de automatización	Compatibilidad con herramientas de automatización como Selenium, Appium, JBehave, Cucumber	Compatibilidad con herramientas de automatización como Selenium, Cucumber, Appium	Compatibilidad con herramientas de automatización como Selenium, Cucumber, Cypress
Reportes	Reportes detallados, "Living documentation" lo cual genera capturas de pantalla paso a paso de cada caso	Reportes personalizables con HTML	Reportes interactivos y visuales
Ventajas	Integración con múltiples herramientas, generación de documentación viva y uso de su propia librería para automatizar	Informes dinámicos y buena compatibilidad con herramientas	Buenos visuales, compatibilidad con las herramientas y lenguajes

Desventajas	Uso complejo generando una difícil curva de aprendizaje	Herramienta que únicamente genera reportes y no interactúa con navegadores	Difícil configuración, simple función de reportes y no interactúa con navegadores

Selección de la herramienta de informes y justificación técnica. Al darle prioridad a herramientas que permitan la documentación continua y actualizada mediante la automatización, la herramienta más beneficiosa para el contexto del proyecto es Serenity BDD, debido a que este framework tiene soporte nativo con BDD, permitiendo la integración directa con las otras herramientas seleccionadas, lo que simplifica la ejecución de las pruebas, la generación de informes y permite escribir pruebas en lenguaje Gherkin (Digital.ai, s. f.). Adicionalmente si se desea, Serenity tiene la capacidad de realizar documentación viva, la cual consiste en la constante actualización de los reportes donde se adjuntan capturas del paso a paso de cada caso y su descripción (Serenity BDD, s. f.).

5.3.1.3 Frameworks BDD para la creación de casos de prueba. En este apartado se investigan las diferentes herramientas existentes para la escritura de casos de prueba en un lenguaje natural basándose en el concepto de Behavior Driven Development (BDD).

Identificación de frameworks BDD para escribir escenarios de prueba

- **JBehave:** JBehave es un marco de pruebas para el desarrollo impulsado por la metodología BDD, fomentando la colaboración entre desarrolladores, QA y empresarios no técnicos en un proyecto de software (JBehave, s. f.).
- **Behave:** Behave es un framework que se basa en BDD, pero enfocado en el lenguaje Python; permite escribir en Gherkin, facilitando la colaboración entre equipos (Behave, s. f.).

- **Cucumber:** El framework de cucumber BDD es una herramienta de automatización de pruebas que se utiliza para escribir pruebas de software en un formato legible fácilmente. Las pruebas se escriben en un lenguaje natural denominado Gherkin que describe el comportamiento esperado del software (QAlified, s. f.).
- **SpecFlow:** La herramienta specflow es un marco de pruebas de código abierto para aplicaciones .NET. Puede generar pruebas BDD utilizando SpecFlow y automatizarlas utilizando Selenium al mismo tiempo (BrowserStack, s. f.).

Comparación de frameworks BDD mediante criterios definidos

En la elección de lo que es la herramienta basada en BDD se descartan principalmente frameworks como Specflow, el cual se encuentra obsoleto, y Behave, siendo compatible solo con el lenguaje Python. De esta forma, las únicas opciones posibles son JBehave, el cual usa un lenguaje propio para escribir escenarios, y Cucumber con el que se escriben casos de prueba con el lenguaje Gherkin.

Tabla 4

Criterios definidos para la comparación y elección de un framework BDD para la creación de casos de prueba

	Jbehave	behave	Cucumber	Specflow
Lenguajes BDD	Lenguaje propio	Gherkin	Gherkin	Gherkin
Lenguaje de programación	Java	Python	Java, JavaScript, Ruby, Python, .NET, etc.	C#
Herramientas de automatización	Compatible con herramientas de automatización como	Selenium, Appium	Compatible con herramientas de automatización como	Selenium, Appium

	selenium y serenity		selenium, cypress, serenity y appium	
Usabilidad	Compleja debido a su configuración y formato del lenguaje	Menor soporte de herramientas	Sencilla debido a su gran documentación	Fácil uso para usuarios C#
Comunidad	Menos popular	Menor comunidad	Muy popular, gran comunidad y amplia documentación	Activa en usuarios del lenguaje .Net
Ventajas	Exclusividad con java y configuración personalizable	La mejor opción para usuarios de python	Mayor integración con herramientas y lenguajes además una amplia comunidad	Buena opción para usuarios de C#
Desventajas	Uso de un solo lenguaje de programación y un lenguaje BDD distinto a gherkin	Menor soporte de herramientas y comunidad, solo compatible con python	Requiere plugins adicionales para integrarse con herramientas de reportes	Herramienta obsoleta

Selección del framework BDD y justificación técnica

Debido a la complejidad que requiere el adaptarse a un nuevo lenguaje natural para la escritura de escenarios, y al formato sencillo que trabaja el equipo QA actualmente para definir los casos de prueba, la herramienta más útil es Cucumber, debido a que al usar el lenguaje Gherkin, ayuda a reducir la curva de aprendizaje.

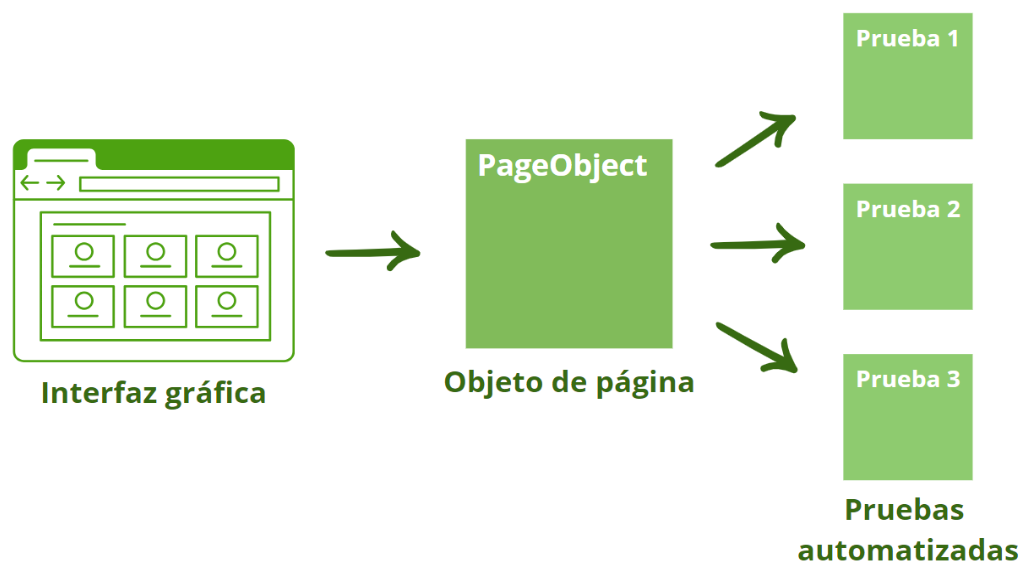
5.3.2 Análisis de patrones de diseño para la automatización de pruebas

En este apartado se analizan tres enfoques comunes en la automatización: Page Object Model (POM), Screenplay y Command, evaluando su aplicabilidad y determinando la mejor opción para el contexto de la DTIC.

5.3.2.1 Descripción de los patrones evaluados. Para la automatización de pruebas es importante evaluar distintos patrones de diseño que permiten estructurar y organizar los scripts de prueba. A continuación, se describen tres patrones comúnmente utilizados: Page Object Model (POM): El patrón de objetos de página se utiliza para representar las páginas web involucradas en una prueba como si fueran objetos en el mismo lenguaje de programación utilizado para escribir los casos de prueba (Leotta, Clerissi, Ricca, & Spadaro, 2013). En la *Figura 9* es presentado un esquema del patrón POM y su estructura.

Figura 9

Esquema del patrón Page Object Model (POM)



Nota. Adaptado de *Patrones de diseño en la automatización: PageObject o ScreenPlay*, por Lopez Seguir, A. Q. (s. f.), SlideShare. https://es.slideshare.net/slideshow/patrones-de-diseo-en-la-automatizacin-pageobject-o-screenplay/120023640

- **Screenplay:** El patrón de diseño guión es una metodología de automatización de pruebas que se centra en describir la interacción del usuario con la aplicación en términos de actores

y tareas. Este patrón es parte de la biblioteca Serenity BDD, pero su concepto se puede aplicar en cualquier framework de automatización (Kovalenko, 2014). En otras palabras, su propósito es que las pruebas describan la forma en la que un usuario puede interactuar con la interfaz para conseguir un objetivo. En la *Figura 10* es presentado un esquema del patrón Screenplay.

Figura 10

Esquema del patrón Screenplay

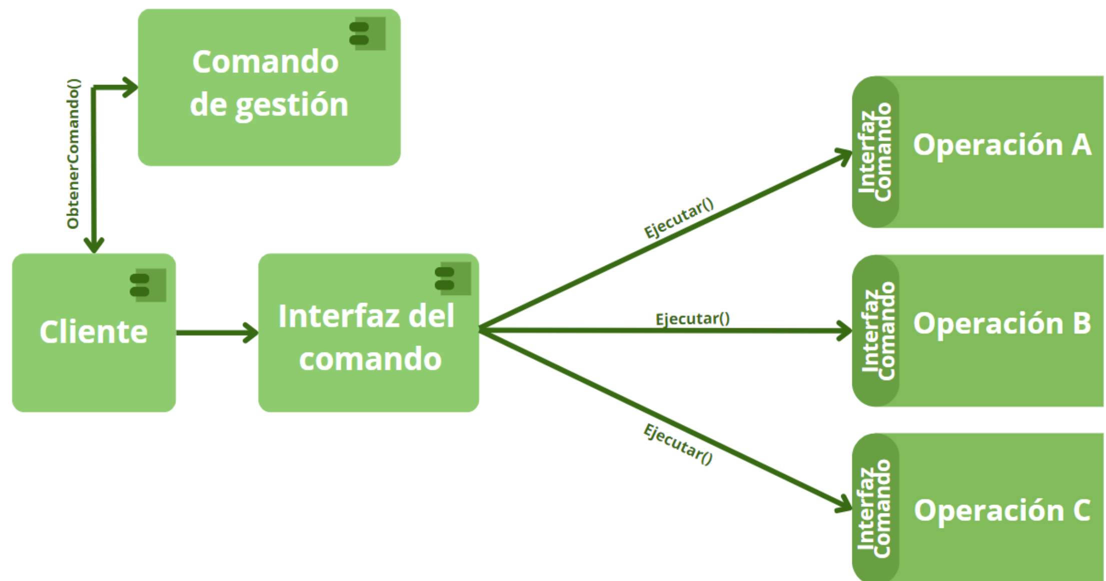
Nota. Adaptado de *Screenplay pattern*, por Sharma, N. (2020, junio 11), TestVagrant. https://medium.com/testvagrant/screenplay-pattern-3490c7f0c23c

- **Command:** El patrón de Comandos, se caracteriza por organizar las acciones de prueba como comandos independientes. Cada comando representa una acción que puede ejecutarse en la aplicación bajo prueba. También puede describirse como un patrón de

comportamiento que transforma una solicitud en un objeto autónomo que incluye la información relevante sobre la misma (Refactoring Guru, s. f.). En la *Figura 11* es presentado un esquema del patrón Command y su estructura.

Figura 11

Esquema del patrón Command



Nota. Adaptado de *Command*, por Reactiveprogramming.io (s. f.). https://reactiveprogramming.io/blog/es/patrones-de-diseno/command

5.3.2.2 Evaluación de aplicabilidad para cada patrón. Para este proyecto, es fundamental seleccionar un patrón de diseño que facilite la automatización de pruebas sin generar una curva de aprendizaje elevada. Dado que el equipo está en una fase inicial de automatización, se prioriza una estrategia estructurada y fácil de mantener, que permita escalar las pruebas conforme el proyecto **avance. A continuación, se evalúa la aplicabilidad de cada patrón en este contexto:**

- **Page Object Model (POM):** Es un patrón ampliamente adoptado en la automatización de pruebas debido a su estructura clara y modular. Entre sus características más relevantes se encuentran (Q2E Banking, s. f.):
  - **Simplicidad y claridad:** Su estructura basada en clases permite una organización intuitiva del código, donde cada página de la aplicación se representa como un objeto con sus respectivos elementos y acciones.
  - **Facilidad de mantenimiento:** Al centralizar la localización de elementos en una sola clase, cualquier cambio en la interfaz de usuario solo requiere modificación en un punto del código, reduciendo el impacto de los cambios en la aplicación.
  - **Curva de aprendizaje accesible:** Su implementación es sencilla y se adapta bien a equipos que están iniciando en la automatización de pruebas.
  - **Escalabilidad progresiva:** Aunque no es el patrón más avanzado, permite evolucionar con el tiempo hacia estructuras más complejas si es necesario.
- **Screenplay:** es un enfoque basado en la reutilización y modularidad mediante la interacción de actores y tareas. Presenta particularidades como (BrowserStack, s. f.):
  - **Mayor modularidad y reutilización:** Su enfoque basado en actores y tareas permite estructurar las pruebas de manera más flexible y reutilizable.

    - **Expresividad en la automatización:** Al describir acciones en un lenguaje más cercano al negocio, facilita la colaboración entre testers y otros roles del equipo.
    - **Mayor complejidad inicial:** Su implementación requiere un cambio de paradigma en la escritura de pruebas, lo que podría representar una curva de aprendizaje más pronunciada para el equipo.
    - **Aplicabilidad a largo plazo:** Es ideal para proyectos con alta demanda de escalabilidad y mantenibilidad, pero puede ser innecesariamente complejo en las primeras etapas de automatización.

- **Command:** encapsula acciones en objetos, permitiendo su ejecución y reutilización de manera flexible. Sus particularidades radican en (GeeksforGeeks, s. f.):
    - **Encapsulación de acciones:** Cada acción se maneja como un objeto independiente, lo que proporciona flexibilidad en la ejecución y parametrización de pruebas.
    - **Separación de lógica y ejecución:** Ayuda a desacoplar las pruebas de la estructura de la aplicación, facilitando cambios sin afectar la automatización.
    - **Menor intuición en etapas iniciales:** La gestión de múltiples comandos puede hacer que la automatización se vuelva más abstracta y difícil de entender para equipos que están iniciando.
    - **Útil en entornos avanzados:** Es una opción viable para pruebas más dinámicas y configurables, pero no es la mejor alternativa cuando se busca simplicidad y rapidez en la implementación.

5.3.2.3 Ventajas y desventajas de cada patrón. A continuación, se presentan las ventajas y desventajas de los patrones evaluados:

Page Object Model (POM): El patrón POM organiza las pruebas automatizadas separando la lógica de la interfaz de usuario en clases independientes. A continuación, se presentan sus ventajas y desventajas (QA Touch, s. f.).

- **Ventajas:**
  - **Mantenibilidad:** Separa la lógica de prueba de la lógica de la interfaz de usuario, facilitando las modificaciones sin afectar las pruebas existentes.
  - **Reutilización de código:** Permite reutilizar los objetos de página en múltiples casos de prueba, reduciendo la duplicación de código.
  - **Legibilidad:** Mejora la organización del código al representar cada página o componente como una clase independiente.
  - **Facilidad de implementación:** Es un patrón sencillo de adoptar, incluso para equipos sin mucha experiencia en automatización de pruebas.
- **Desventajas:**
  - **Acoplamiento con la interfaz de usuario:** Si la interfaz cambia con frecuencia, es necesario actualizar los objetos de página constantemente.
  - **No tan fácil para pruebas altamente dinámicas:** Puede volverse difícil de manejar en aplicaciones con muchas interacciones dinámicas.

Screenplay: Screenplay define pruebas en términos de actores y tareas, promoviendo modularidad y desacoplamiento. A continuación, se presentan sus ventajas y desventajas (Pragma, s. f.).

- **Ventajas:**

- **Mayor desacoplamiento:** Los actores representan a los usuarios y sus interacciones, reduciendo la dependencia de la estructura de la interfaz.
- **Reutilización modular:** Facilita la creación de pruebas más flexibles y escalables mediante la combinación de tareas reutilizables.
- **Mejor mantenimiento en proyectos grandes:** Se adapta bien a sistemas complejos con múltiples funcionalidades, minimizando la redundancia en las pruebas.

- **Desventajas:**
  - **Mayor curva de aprendizaje:** Su implementación requiere mayor conocimiento en programación orientada a objetos y principios SOLID.
  - **Complejidad inicial:** Puede ser excesivo para proyectos pequeños o para equipos que inician en la automatización de pruebas.

Command: El patrón Command encapsula acciones como objetos independientes, mejorando la reutilización pero aumentando la complejidad. A continuación, se presentan sus ventajas y desventajas (Refactoring Guru, s. f.).

- **Ventajas:**
  - **Encapsulación de comandos:** Permite definir acciones como objetos independientes, mejorando la flexibilidad y reutilización.
  - **Desacoplamiento entre emisores y receptores:** Facilita la modificación de acciones sin afectar directamente a las pruebas.
  - **Registro y reversión de acciones:** Puede ser útil para pruebas en las que se necesite implementar operaciones de "deshacer".
- **Desventajas:**

    - **Mayor cantidad de clases:** Puede aumentar la complejidad del código debido a la necesidad de definir múltiples comandos individuales.
    - **No es el más intuitivo para pruebas UI:** Su enfoque en la ejecución de comandos lo hace más adecuado para lógica de negocio que para pruebas de interfaz de usuario.

Estos aspectos han sido considerados para seleccionar el patrón más adecuado para el contexto del proyecto.

5.3.2.4 Patrón seleccionado y justificación en el contexto de la DTIC. Tras analizar los diferentes patrones de diseño presentados anteriormente, se ha determinado que Page Object Model (POM) es el patrón más adecuado para la automatización de pruebas dentro del contexto del proyecto. Esta elección responde tanto a las necesidades técnicas del sistema como a la transición de pruebas manuales a pruebas automatizadas dentro de la División de Tecnologías de la Información y las Comunicaciones (DTIC). Por consiguiente, enseguida se exponen las principales razones para seleccionarlo.

- **Facilita la transición de pruebas manuales a automatizadas:** Desde la instauración del equipo de QA en la DTIC, todos sus procesos de validación han sido a través de pruebas manuales, lo que implica una curva de aprendizaje al adoptar automatización. En este sentido, POM proporciona una estructura sencilla e intuitiva, lo cual permite que los testers manuales se adapten progresivamente sin necesidad de conocimientos avanzados en programación.
- **Estructura modular y mantenible:** En la DTIC, los sistemas a probar presentan interfaces dinámicas y en constante evolución. Y aunque, POM en ese aspecto no es tan fácil de manejar, sí permite separar la lógica de prueba de la interfaz de usuario, ayudando a

encapsular los elementos en clases reutilizables. Por esto, se reduce el impacto de los cambios en la UI, evitando la modificación constante de los scripts de prueba.

- **Escalabilidad para los sistemas de la DTIC:** A medida que el número de pruebas automatizadas crezca, es fundamental contar con un patrón que permita ampliar el alcance sin comprometer la mantenibilidad. De esta manera, POM favorece la escalabilidad mediante la reutilización de componentes, facilitando la incorporación de nuevas pruebas sin afectar las ya existentes.
- **Integración con las herramientas utilizadas en la DTIC:** La automatización en la DTIC debe alinearse con las herramientas seleccionadas anteriormente, como Selenium y Serenity BDD. Por lo cual, POM es un estándar ampliamente compatible con estos frameworks, lo que asegura una integración eficiente en el contexto ágil actual.
- **Optimización del tiempo y reducción del esfuerzo manual:** La implementación de pruebas automatizadas busca optimizar el tiempo dedicado a validaciones repetitivas. POM permite escribir scripts eficientes y reutilizables, lo que disminuye la carga de trabajo sobre el equipo de QA y acelera el ciclo de pruebas.
- **Facilidad de adopción en los diferentes equipos:** Al ser un enfoque intuitivo y sencillo de implementar, POM permite que tanto testers manuales como desarrolladores puedan contribuir a la automatización sin requerir extensos conocimientos previos. Todo esto, permite una adopción progresiva y minimiza la resistencia al cambio dentro del equipo.

El Page Object Model (POM) ha sido seleccionado como el patrón más adecuado para la automatización de pruebas en la DTIC debido a su facilidad de implementación, su alineación con las herramientas utilizadas y su capacidad de soportar el crecimiento del proyecto sin comprometer

la mantenibilidad. Además, representa una solución óptima para facilitar la transición desde las pruebas manuales, asegurando un proceso estructurado y eficiente.

5.3.3 Definición del proceso de automatización

Automatizar pruebas no es solo escribir código y ejecutar scripts. También, es un proceso estructurado que empieza con la selección de los casos de prueba adecuados y termina con el mantenimiento y la mejora continua de los scripts. En este apartado, se detalla cada paso de este proceso para ayudar a que la automatización realmente aporte valor al equipo y al proyecto.

5.3.3.1 Identificación de pruebas a automatizar. La automatización de pruebas es un proceso estratégico que busca optimizar la calidad del software y la eficiencia del equipo de pruebas. No todas las pruebas pueden o deben ser automatizadas; por ello, es fundamental realizar un análisis detallado para identificar aquellas que realmente aportarán valor al proceso de desarrollo. Este análisis se lleva a cabo en tres etapas clave:

1. **Análisis de Historias de Usuario asignadas:** El primer paso es realizar un estudio detallado de las Historias de Usuario (HU) incluidas en el sprint. Esto permite identificar las funcionalidades que serán implementadas o ajustadas y, a partir de ello, determinar qué pruebas pueden beneficiarse de la automatización. Para este análisis, se siguen los siguientes pasos:

   Revisión detallada de las HU

   - Se estudian todas las HU planificadas en el sprint, verificando su alcance y objetivos.
   - Se identifican las funcionalidades clave que impactan directamente al usuario final.

Identificación de criterios de aceptación

- Se analizan los criterios de aceptación definidos en cada HU, los cuales representan las condiciones mínimas que debe cumplir la funcionalidad para ser considerada completada.
- Se determinan las validaciones necesarias para verificar el correcto funcionamiento del sistema.

Detección de pruebas necesarias

- Se establece qué pruebas son obligatorias para validar el cumplimiento de los criterios de aceptación.
- Se identifican los escenarios que requieren validaciones repetitivas o que implican flujos de usuario críticos.

2. **Priorización de pruebas:** No todas las pruebas tienen la misma importancia dentro del proceso de desarrollo. Algunas tienen un impacto directo en la estabilidad del sistema y la experiencia del usuario, mientras que otras pueden ser menos críticas o incluso innecesarias en la fase de automatización. Para definir qué pruebas deben priorizarse dentro de la estrategia de automatización, se aplican los siguientes criterios:

Impacto en el sistema

- Se priorizan aquellas pruebas que validan funcionalidades críticas para el usuario y el negocio.
- Se consideran las pruebas que afectan módulos interconectados y cuyo fallo puede generar errores en cascada.

Frecuencia de ejecución (Repetitividad)

- Se priorizan los casos que se ejecutan de manera frecuente, como pruebas de regresión y pruebas de integración.
- Se identifican pruebas que requieren validaciones repetitivas, ya que su automatización reducirá significativamente la carga de trabajo manual.

Tiempo de ejecución manual

- Se priorizan pruebas cuyo tiempo de ejecución manual es alto y puede optimizarse con la automatización.
- Se evalúa el esfuerzo requerido para la ejecución manual versus el esfuerzo para automatizar la prueba.

Complejidad y estabilidad de la funcionalidad

- Se analiza la estabilidad del módulo o funcionalidad a probar.
- Se evitan automatizar pruebas sobre funcionalidades en constante cambio, ya que esto puede generar un mantenimiento excesivo en los scripts.

3. **Filtrado de Historias de Usuario candidatas:** Después de analizar y priorizar las pruebas, es necesario realizar un filtro final para descartar aquellas que no son viables para la automatización y seleccionar las que realmente aportarán valor.

Descartar pruebas inestables o de alto mantenimiento

- Se excluyen pruebas sobre funcionalidades en desarrollo o que están sujetas a cambios constantes.
- Se evitan automatizar casos con dependencias externas inestables (como integraciones con servicios de terceros no controlados por el equipo).

Eliminar pruebas con validaciones visuales complejas

- Se identifican casos donde la validación depende de elementos gráficos que pueden ser difíciles de automatizar con precisión, como cambios de color, posicionamiento dinámico de elementos o animaciones.
- Se priorizan pruebas funcionales sobre pruebas puramente estéticas.

Seleccionar los casos viables para automatización

- Se documentan las pruebas seleccionadas con detalles sobre su propósito, precondiciones y criterios de éxito.
- Se agrupan los casos de prueba según su nivel de automatización y su relación con los criterios de aceptación.

En general, la identificación de pruebas a automatizar es una fase fundamental dentro del proceso de automatización. Debido a que, un análisis detallado de las Historias de Usuario, junto con una priorización adecuada basada en impacto, repetitividad y esfuerzo manual, permite seleccionar los casos más adecuados para la automatización. Es por esto, que filtrar cuidadosamente las pruebas candidatas evita el desperdicio de recursos en la automatización de pruebas inestables o de difícil mantenimiento, dándole así mejor uso de la automatización.

5.3.3.2 Planificación de la automatización. Una vez identificadas las pruebas candidatas para la automatización, es fundamental planificar cómo integrarlas dentro del sprint sin afectar la entrega del proyecto. Para ello, se establecen tiempos adecuados, se configura el entorno de pruebas y se organizan los casos en un formato estructurado que facilite su desarrollo y mantenimiento.

1. **Adaptación de los tiempos de automatización al sprint:** La automatización de pruebas debe alinearse con los tiempos del sprint y la dinámica del equipo, por lo que es necesario:

- **Coordinar con el equipo:** Se establecen reuniones con desarrolladores y testers para definir el tiempo que se destinará a la automatización sin comprometer las pruebas manuales o la validación de nuevas funcionalidades.
- **Definir prioridades:** Se organiza el backlog de automatización según la criticidad de las funcionalidades y la disponibilidad de recursos.
- **Equilibrar esfuerzos:** Se distribuye el trabajo de pruebas entre automatización y validaciones manuales para evitar sobrecargas en el equipo de QA.

2. **Preparación del entorno de pruebas:** Para promover la correcta ejecución de los scripts automatizados, es necesario configurar adecuadamente el entorno, asegurando compatibilidad con el sistema bajo prueba.
    - **Instalación de dependencias y herramientas necesarias**
        - Se verifica que todas las herramientas requeridas para la automatización estén instaladas y actualizadas.
        - Instalación de Java en la versión compatible con el framework.
        - Instalación de Maven para la gestión de dependencias.
        - Configuración del entorno de desarrollo en IntelliJ IDEA o cualquier otro IDE utilizado.
    - **Configuración de WebDrivers y ejecución en distintos navegadores**
        - Instalación y configuración de WebDrivers para Chrome, Firefox u otros navegadores compatibles.
        - Validación del correcto funcionamiento de los navegadores en el entorno de prueba (develop).

    - Configuración de variables de entorno para que el sistema reconozca las rutas de los WebDrivers.
- **Validación de la versión de Java y compatibilidad con el framework**
    - Se verifica que la versión de Java sea compatible con el framework de automatización seleccionado (en este caso, Serenity BDD con Cucumber y Selenium).

3. **Configuración de archivos clave:** Los archivos de configuración permiten ajustar parámetros importantes de la automatización, por lo que se debe tener su correcta edición:
    - **Archivo serenity.properties**
        - Se ajustan parámetros como el navegador por defecto, la estrategia de reinicio entre escenarios y las rutas de ejecución.
        - Se establecen configuraciones de logs y reportes de prueba.
    - **Archivo pom.xml**
        - Se añaden o actualizan dependencias necesarias para el framework de automatización.
        - Se configuran plugins esenciales para la ejecución de pruebas.
        - Se ajustan perfiles de ejecución para correr pruebas de forma local o en un entorno de integración continua (CI/CD).
4. **Creación de casos de prueba en Gherkin:** Los casos de prueba automatizados deben estructurarse de manera clara y comprensible en lenguaje Gherkin, lo que permite que cualquier miembro del equipo (técnico o no técnico) pueda interpretar su funcionamiento.
    - **Definición de escenarios de prueba**

- Se redactan los escenarios en formato Feature y Scenario, asegurando que sigan la estructura Given-When-Then para facilitar su comprensión.
- Se verifica que cada escenario cubra los criterios de aceptación definidos en la Historia de Usuario.
- Se eliminan redundancias y se optimiza la escritura de pasos reutilizables para mejorar la mantenibilidad.

- **Organización dentro de la estructura del proyecto**
  - Los archivos .feature se almacenan en la carpeta resources/features dentro del repositorio del proyecto.
  - Se implementa una nomenclatura clara para los archivos (HU_XXXX_NombreFuncionalidad.feature) para facilitar su identificación y rastreo.
  - Se documentan los escenarios con comentarios explicativos cuando sea necesario.

Con esta planificación bien definida, se garantiza que el proceso de automatización se integre eficientemente en el sprint, optimizando tiempos y asegurando la calidad de los scripts generados.

5.3.3.3 Diseño de la solución de automatización. El diseño de la solución de automatización es una etapa relevante para promover que las pruebas sean escalables, organizadas y mantenibles a lo largo del tiempo. Esta fase establece la estructura de proyecto, la integración con herramientas de control de versiones y la organización de los diferentes componentes del código.

Para lograr una implementación eficiente, se adopta el patrón de diseño Page Object Model (POM), el cual permite una clara separación entre la lógica de prueba y la interacción con la interfaz de usuario. Gracias a este enfoque, se mejora la reutilización del código, la legibilidad y el mantenimiento del proyecto a medida que evolucionan los requerimientos.

1. **Creación del proyecto en IntelliJ IDEA:** El primer paso es la creación del proyecto en IntelliJ IDEA, utilizando Maven como gestor de dependencias para apoyar una correcta administración de librerías y configuraciones. Durante esta etapa se deben realizar las siguientes acciones:
    - Abrir el proyecto base configurado inicialmente, asegurando que tenga la estructura adecuada para la integración con Cucumber, Serenity BDD y Selenium.
    - Validar el archivo pom.xml, asegurando que todas las dependencias necesarias para la automatización se encuentren instaladas.
    - Definir la estructura de carpetas bajo un paquete principal estándar, como ***com.prueba.co***, para mantener un código bien organizado y de fácil acceso.
    - Validar la configuración de los archivos de propiedades (serenity.properties) para establecer parámetros de ejecución clave, como el navegador, la URL base y la estrategia de reinicio del navegador entre pruebas.

2. **Sincronización con Git:** Para tener un control de versiones efectivo, el proyecto se sincroniza con un repositorio en una plataforma de gestión de código como lo es GitLab. Este proceso involucra:
   - Inicialización del repositorio Git, vinculando el proyecto con un sistema de control de versiones para gestionar los cambios de manera estructurada.
   - Configuración del archivo .gitignore, evitando la inclusión de archivos innecesarios como ***target/, .idea/, *.log***, y otros que puedan afectar la limpieza del repositorio.
   - Creación de ramas de desarrollo, siguiendo una estrategia clara para separar funcionalidades (***feature/automatizacion_hu123***) y mantener una rama estable (main o develop).
   - Uso de pull requests para revisiones de código, asegurando la validación del código antes de fusionarlo con la rama principal.
3. **Organización del código según el POM:** El código se organiza siguiendo el patrón de diseño Page Object Model (POM), el cual permite estructurar la automatización de pruebas separando la lógica de prueba de la interacción con la interfaz de usuario. En este sentido, cada página del sistema bajo prueba se representa mediante una clase, encapsulando sus elementos y métodos de interacción. Esto facilita la reutilización del código y mejora la mantenibilidad a medida que se actualiza la aplicación.

   Dentro de ***com.prueba.co***, el código se divide en paquetes específicos según su función.

   3.1 Paquete principal: com.prueba.co

   Este paquete actúa como la raíz del proyecto e incluye todas las clases y archivos organizados en subpaquetes según su función.

3.2 Definitions (Definiciones de pasos de prueba)

Ubicado dentro de ***com.prueba.co.definitions***, este paquete contiene las clases donde se definen los pasos en Cucumber utilizando anotaciones Given, When y Then.

Aquí se encuentran los archivos .java que actúan como puente entre los escenarios escritos en Gherkin y la implementación de los pasos. Su propósito es:

- Definir la ejecución de las pruebas en función de los escenarios escritos en ***.feature***.
- Llamar a los métodos correspondientes dentro del paquete Steps.
- Facilitar la comprensión de cada paso dentro de la automatización.

3.3 Pages (Implementación del Page Object Model - POM)

Siguiendo el Page Object Model, este paquete (***com.prueba.co.pages***) contiene la representación de la interfaz gráfica del sistema bajo prueba, organizándose en dos subpaquetes:

- **test:** Define los campos y selectores usados en las acciones iniciales (Given y When).
- **validations:** Define los campos y selectores usados en la validación de resultados (Then).

Cada clase dentro de este paquete almacena los selectores de elementos web (por ejemplo, identificadores de botones, campos de entrada y mensajes de notificación) y métodos para interactuar con ellos.

3.4 Steps (Implementación de la lógica de prueba)

Ubicado en ***com.prueba.co.steps***, este paquete contiene la lógica de ejecución de las pruebas, implementando las acciones que interactúan con los elementos de la interfaz. Se divide en dos subpaquetes:

- **test:** Contiene los métodos que realizan interacciones dentro del sistema, como ingresar datos, hacer clic en botones o navegar entre pantallas.
- **validations:** Contiene los métodos que validan los resultados obtenidos después de la ejecución de una prueba.

Cada archivo dentro de este paquete agrupa la lógica de las pruebas de una funcionalidad específica.

3.5 Utilities (Utilidades y configuraciones adicionales)

Este paquete (***com.prueba.co.utilities***) agrupa todas las configuraciones generales y archivos auxiliares necesarios para la automatización. Algunas funciones clave de este paquete incluyen:

- Clase ***Website.java***: Configura el runner para la ejecución de pruebas, definiendo parámetros de inicialización y cierre del navegador.
- Archivos de configuración como ***serenity.properties***, donde se establecen valores por defecto para la ejecución de las pruebas.
- Gestión de datos de prueba, facilitando la reutilización de información en múltiples escenarios y evitando la dependencia de datos dinámicos generados en cada ejecución.

El diseño de la automatización se basó en el patrón Page Object Model (POM) para mantener el código organizado y fácil de mantener. Esta estructura permite gestionar mejor las pruebas y facilita futuras mejoras, asegurando un proceso más eficiente y escalable.

5.3.3.4 Implementación y desarrollo de script. Una vez definida la arquitectura del proyecto y organizada la estructura del código bajo el patrón Page Object Model (POM), se procede con la implementación de los scripts de prueba automatizada. Esta fase implica la programación de los métodos que permitirán la interacción con la interfaz del sistema y la ejecución de los escenarios de prueba definidos en Gherkin.

- **Programación de búsqueda de elementos en la interfaz:** Para que las pruebas automatizadas puedan interactuar con la aplicación, es necesario identificar los elementos dentro de la interfaz de usuario. Esta identificación se realiza a través de selectores como:
    - **ID:** Ideal cuando los elementos tienen identificadores únicos.
    - **Name:** Útil cuando los nombres de los campos son estáticos y únicos.
    - **XPath:** Utilizado para acceder a elementos cuando no tienen ID o Name, permitiendo rutas flexibles.
    - **CSS Selectors:** Más eficientes y rápidos que XPath en algunos casos.
    - **Class Name y Tag Name:** Útiles cuando se necesita seleccionar elementos por su tipo o clases compartidas.

    Estos selectores se documentan y almacenan en archivos específicos según su propósito dentro del proyecto:

    - ***pages.test:*** Contiene los selectores relacionados con los pasos de Given y When, es decir, los elementos con los que el usuario interactúa.

  - ***pages.validations:*** Contiene los selectores necesarios para realizar las validaciones de los resultados esperados en los pasos Then.

Este enfoque permite una clara separación de responsabilidades, facilitando el mantenimiento de los selectores cuando la interfaz de usuario cambia.

- **Desarrollo de métodos de interacción con el navegador:** Con los selectores definidos, se procede a desarrollar los métodos que permitirán la interacción con la interfaz. Estos métodos se encargan de:
  - Ingresar datos en campos de texto.
  - Hacer clic en botones y enlaces.
  - Seleccionar opciones en listas desplegables.
  - Esperar la carga de elementos dinámicos.
  - Capturar y comparar valores de la interfaz con los valores esperados.

  Estos métodos se organizan en la siguiente estructura dentro del código:
  - ***steps.test*:** Contiene los métodos que ejecutan acciones sobre los elementos de la interfaz, correspondientes a los pasos Given y When.
  - ***steps.validations*:** Contiene los métodos que validan el comportamiento esperado del sistema, usados en los pasos Then.

  Este enfoque modular permite reutilizar los métodos en múltiples escenarios de prueba, evitando la duplicación de código y facilitando la escalabilidad del framework de pruebas.

- **Llamado de métodos en Cucumber:** Para que los escenarios escritos en Gherkin puedan ejecutarse correctamente, es necesario establecer la conexión entre los pasos definidos en lenguaje natural y los métodos programados en Java. Esto se logra mediante el paquete ***definitions***, donde se implementan las llamadas a los métodos desarrollados en ***steps.test y***

steps.validations. Dentro de este paquete, los pasos en Gherkin son enlazados con los métodos correspondientes utilizando anotaciones de Cucumber como:

- **@Given:** Define el estado inicial de la prueba (ejemplo: "El usuario abre la página de inicio de sesión").
- **@When:** Representa una acción que el usuario realiza (ejemplo: "El usuario ingresa sus credenciales y presiona el botón de iniciar sesión").
- **@Then:** Verifica el resultado esperado (ejemplo: "El sistema muestra un mensaje de bienvenida").

De esta manera, cada escenario de prueba ejecutado en Cucumber llamará a los métodos correspondientes, permitiendo que la automatización refleje el comportamiento esperado de la aplicación.

5.3.3.5 Ejecución de las pruebas automatizadas. Una vez han sido implementados los scripts de automatización y se ha realizado el respectivo llamado de los métodos en Cucumber, verificando la relación de los casos de prueba con cada método de interacción con la interfaz, se debe ejecutar el proyecto. En esta fase se explora la correcta ejecución de casos de prueba automatizados anteriormente definidos y el uso del gestor Maven para la generación de informes con serenity.

- **Creación de la clase Runner:** Para facilitar la ejecución de un proyecto tan grande se debe crear una clase denominada ***Runner.java*** la cual contenga todas las instancias necesarias para ejecutar Cucumber en conjunto al subpaquete que contiene toda la lógica de las prueba ***com.prueba.co.definitions***
- **Ejecución de Maven:** Ya generada correctamente la clase runner, ahora se debe correr desde las opciones que brinda el entorno de desarrollo los respectivos comandos de ***Maven***,

en donde se desplegará un enlace al navegador para visualizar un informe detallado con los resultados de las pruebas ejecutadas.

- **Generación de informes:** Al usar herramientas como lo son Serenity BDD con su funcionalidad "living documentation" y Maven, que mediante sus opciones de gestor de informes facilita la generación de estos, es posible su posterior análisis para comprender los resultados de cada caso de prueba con una fácil lectura, además de ser casi instantáneo el reporte recortando el tiempo de documentación de las pruebas casi a un 100%.

5.3.3.6 Análisis y validación de los resultados. Teniendo ya la prueba generada y su respectivo informe el paso a seguir para el QA a cargo es dar análisis a estos resultados con el fin de reportar los hallazgos en la historia de usuario siendo esto una forma de validar cada caso de prueba teniendo 2 opciones como posible resultado, fallar o pasar.

- **Aprobación casos de prueba:** Al verificar la información de cada caso de prueba y no encontrar hallazgos en el comportamiento funcional de la HU se debe reportar este informe mediante un link en los comentarios del gestor de proyectos TAIGA y dar por aprobada la historia.
- **Casos de prueba fallidos:** Haciendo la correspondiente revisión del informe puede presentarse la situación de que la prueba automatizada haya fallado en casos específicos debido a una mala implementación en el desarrollo del sistema haciendo que no se cumplan reglas de negocio, criterios de aceptación o hasta la funcionalidad principal por ende este reporte se debe adjuntar en una conocida issue la cual se genera dentro del gestor de proyectos mencionado con toda la información posible para la reproducción del caso de prueba y para el mayor entendimiento del desarrollador responsable de esta issue.

- **Seguimiento de la prueba fallida:** Con el fin de corregir estos errores encontrados se debe hacer un correcto seguimiento a esta issue en acompañamiento del desarrollador para la verificación del funcionamiento adecuado de las pantallas, al reportarse las correcciones realizadas se debe recurrir a una prueba automatizada de regresión para validar que el código añadido no interfiera o modifique el comportamiento en los casos de prueba ya aprobado terminando así el proceso de aprobación de la issue.

Una vez los casos de prueba han sido ejecutados y validados, los reportes generados por la automatización se almacenan en una carpeta específica dentro del Drive del proyecto de la DTIC. Este repositorio permite centralizar los resultados de las ejecuciones realizadas, conservar un histórico de las pruebas automatizadas y facilitar la consulta de los informes por parte del equipo de QA y demás actores involucrados, sirviendo como soporte del proceso de validación y aprobación de las historias de usuario.

5.3.3.7 Mantenimiento de los scripts y mejora continua. Un paso importante para el futuro de estos scripts de pruebas ya programados es el mantenerlos funcionales a través del tiempo, es decir, que estas pruebas no se vuelvan obsoletas con el paso de los días y el avance de desarrollos en el mismo proyecto. Por este motivo, se debe hacer un correcto mantenimiento de dichos scripts ante alteraciones del software y buscando la mayor calidad del proceso al implementar las nuevas tecnologías que permitan optimizar con mayor efectividad el flujo de automatización.

Mantenimiento del scripts

El mantenimiento de los scripts es fundamental debido a los constantes cambios en el sistema, como nuevas funcionalidades, actualizaciones de herramientas o integración de módulos. Para evitar su obsolescencia, es necesario actualizarlos tras cada modificación del software y evaluar

los casos de prueba en Gherkin para determinar si requieren ajustes o siguen siendo válidos para pruebas de regresión. Además, la refactorización del código y de los casos de prueba es clave para optimizar su eficiencia, mejorar su legibilidad y mantener su compatibilidad con nuevas versiones, asegurando así la calidad y estabilidad del software a lo largo del tiempo.

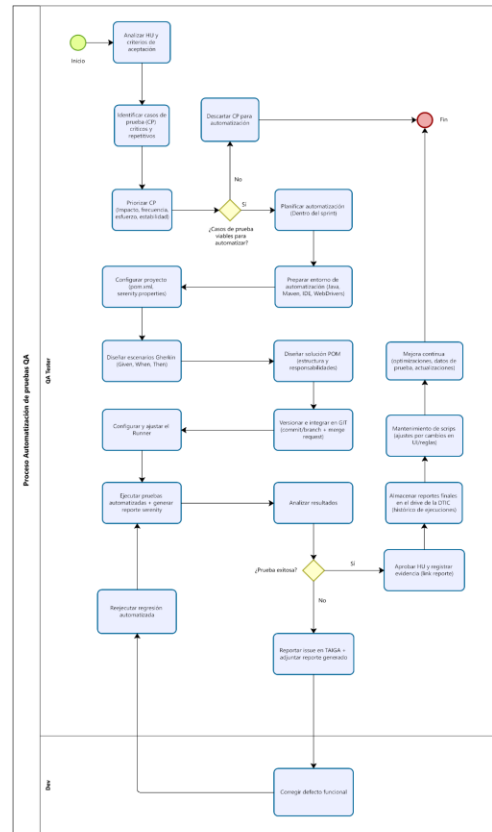
Mejora continua

La mejora continua en la automatización de pruebas es clave para mantener la calidad y eficiencia del testing a lo largo del desarrollo del software. Esto implica revisar el rendimiento de las pruebas, identificar fallos recurrentes y optimizar tiempos de ejecución. También es importante actualizar herramientas, mejorar la gestión de datos de prueba y paralelizar la ejecución para acelerar resultados.

5.3.3.8 Diagrama general del proceso de automatización de pruebas. En la *Figura 12* se presenta el diagrama del proceso de automatización de pruebas, el cual resume de forma general el flujo definido en las secciones anteriores. Este diagrama permite visualizar la secuencia de actividades que componen la automatización, desde la identificación y selección de los casos de prueba hasta la ejecución, validación de resultados y mantenimiento de los scripts.

Figura 12

Flujo de automatización de pruebas final



El flujo se enfoca en los roles de QA y Desarrollador, e incorpora puntos de decisión que permiten determinar la viabilidad de automatización de los casos de prueba y las acciones a seguir ante la aprobación o fallo de las pruebas ejecutadas. De esta manera, el diagrama funciona como un apoyo visual que facilita la comprensión del proceso sin entrar en el nivel de detalle ya abordado previamente.

5.4 Pruebas del plan de automatización propuesto

Aquí se presenta la aplicación práctica del plan de automatización propuesto, mediante la ejecución de las pruebas automatizadas definidas para el módulo de proveedores. A través de esta implementación, se valida el proceso establecido, permitiendo evaluar su funcionamiento,

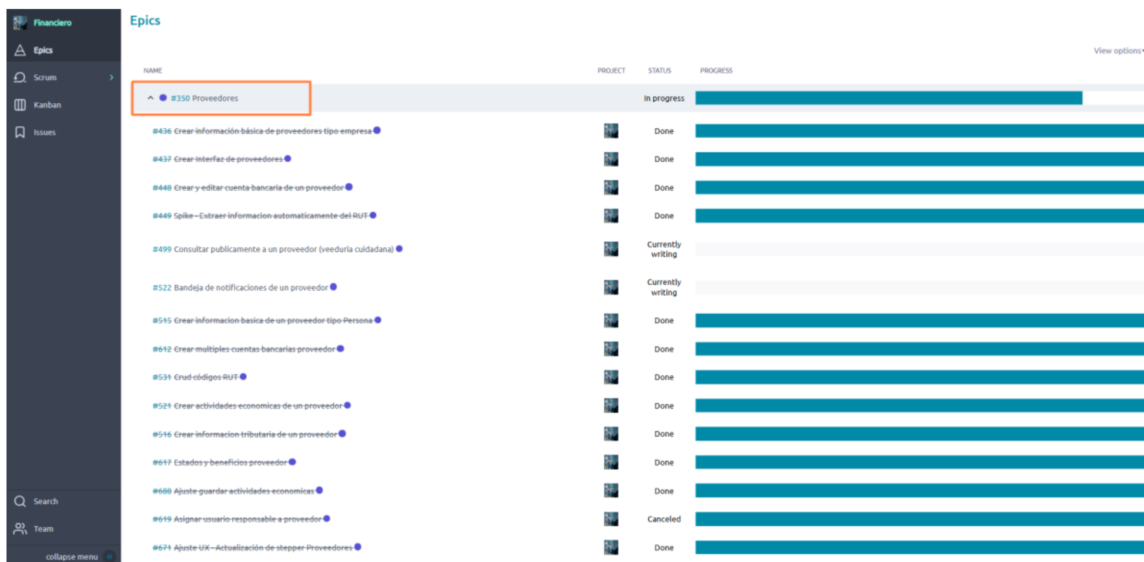
identificar ajustes necesarios y analizar los resultados obtenidos como insumo para la medición del impacto del enfoque automatizado frente al esquema de pruebas manuales.

5.4.1 Análisis y selección de casos de prueba seleccionados para automatizar

En esta fase se realizó el análisis de los casos de prueba asociados al módulo de Proveedores con el propósito de identificar aquellos que eran viables para ser automatizados. Dicho análisis se fundamentó en el estudio de las Historias de Usuario y de sus respectivos criterios de aceptación, lo que permitió identificar los flujos funcionales principales y alternos que requieren validaciones repetitivas durante el ciclo de desarrollo del software. En la *Figura 13* se presenta la épica del módulo de Proveedores, en la cual se visualiza un poco del conjunto completo de Historias de Usuario consideradas como base para el análisis y posterior selección de los casos de prueba candidatos a automatización.

Figura 13

Vista general de la épica de proveedores



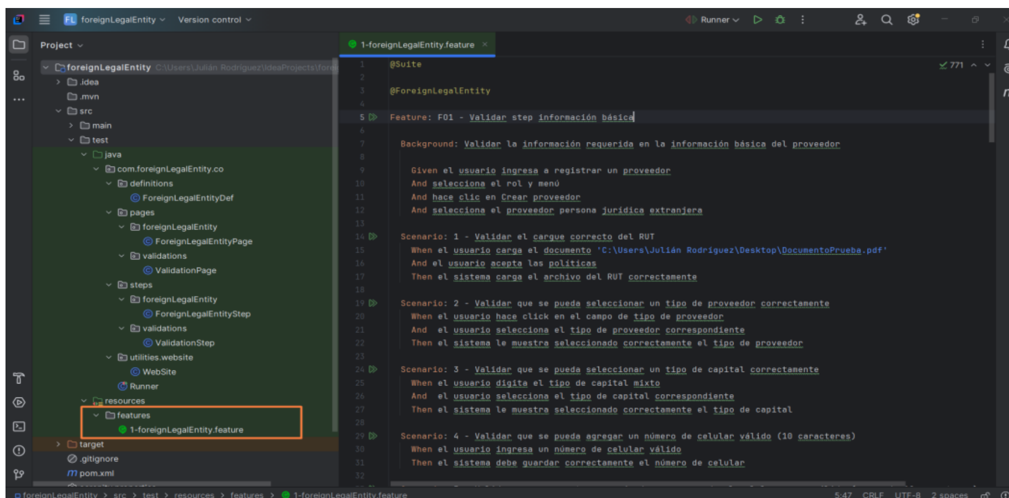
Durante este proceso se consideraron criterios como la criticidad de la funcionalidad, la frecuencia de ejecución de las pruebas, el esfuerzo requerido para su validación manual y la viabilidad técnica de automatización. La aplicación de estos criterios permitió descartar aquellos casos cuyo costo de mantenimiento sería elevado o cuyo aporte al proceso de automatización sería limitado. Como resultado, se seleccionaron 56 casos de prueba para persona natural, 86 casos de prueba para persona jurídica y 26 casos de prueba para persona jurídica extranjera.

5.4.2 Diseño de los casos de prueba automatizados

Una vez definidos los casos de prueba candidatos para automatización, se procedió con el diseño de los escenarios en lenguaje Gherkin bajo el enfoque de *Behavior-Driven Development* (BDD). Este diseño permitió describir el comportamiento esperado del sistema de manera clara, estructurada y alineada con los criterios de aceptación definidos. En la *Figura 14* se presenta un ejemplo de archivo .feature, en el cual se evidencia tanto la definición de la funcionalidad a evaluar (*Feature*) como la estructura de los escenarios asociados (*Scenario*), construidos siguiendo el patrón Given–When–Then.

Figura 14

Estructura de los archivos .feature



Esta estructura permitió establecer una diferenciación explícita entre el estado inicial del sistema, las acciones ejecutadas y los resultados esperados, facilitando la comprensión de los escenarios y promoviendo la reutilización de pasos comunes. Como resultado, se redujeron las redundancias y se mejoró la mantenibilidad de las pruebas automatizadas. El detalle completo de los escenarios diseñados, correspondientes a 56 escenarios para persona natural, 86 para persona jurídica y 26 para persona jurídica extranjera, se encuentra documentado en el **Apéndice A.**

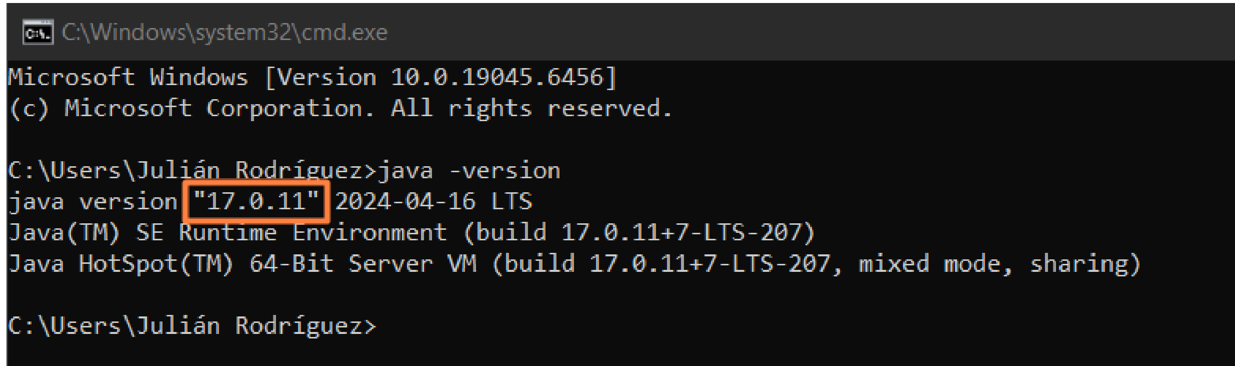
5.4.3 Configuración técnica del entorno de automatización

La configuración técnica del entorno de automatización constituyó una etapa esencial para promover la correcta ejecución de los scripts y la estabilidad del framework de pruebas. Esta fase incluyó la instalación de las herramientas base, la configuración del entorno de desarrollo y la validación de la compatibilidad entre las tecnologías utilizadas. Esto con el fin de mantener un entorno configurado, en la cual se evidencien los componentes necesarios para la ejecución de las pruebas automatizadas.

5.4.3.1 Instalación y configuración de herramientas. Inicialmente, se realizó la instalación del Java Development Kit (JDK) en una versión compatible con el framework de automatización, configurando las variables de entorno requeridas para su correcto funcionamiento. En la *Figura 15* se evidencia la verificación de la versión de Java instalada mediante consola, confirmando que el sistema reconoce adecuadamente la configuración realizada.

Figura 15

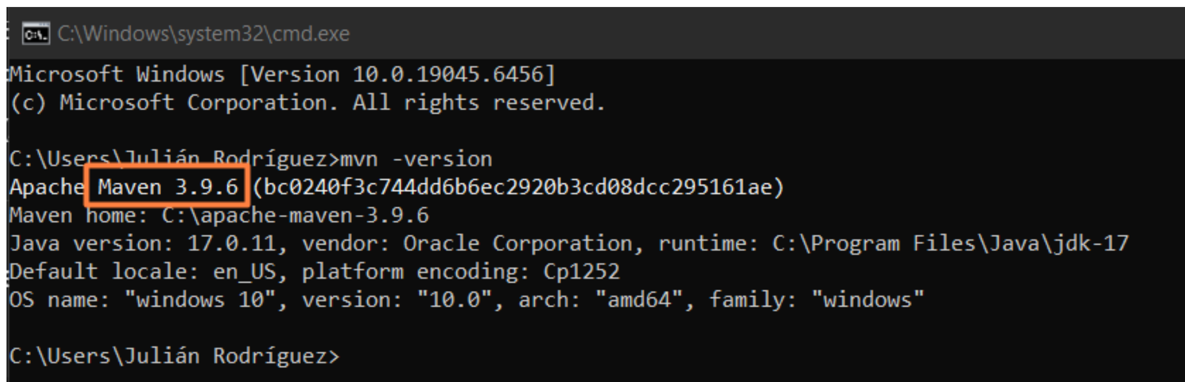
Verificación de versión de Java por consola



Posteriormente, se llevó a cabo la instalación de Maven como gestor de dependencias y herramienta de ejecución. Tal como se presenta en la *Figura 16*, la verificación de la versión instalada permitió confirmar que el entorno se encontraba preparado para la gestión del proyecto y la ejecución de las pruebas automatizadas.

Figura 16

Verificación de versión de Maven en consola



De manera complementaria, se configuró el entorno de desarrollo integrado (IDE), importando el proyecto Maven y sincronizando las dependencias definidas en el archivo pom.xml. En la *Figura 17* se observa la correcta carga de las dependencias asociadas al framework Serenity

BDD, Cucumber y Selenium. Finalmente, se validó la instalación del navegador objetivo y la disponibilidad del WebDriver correspondiente, lo cual se refleja en la *Figura 18*, permitiendo la correcta interacción con el sistema bajo prueba

Figura 17

Sincronización del proyecto Maven en el IDE

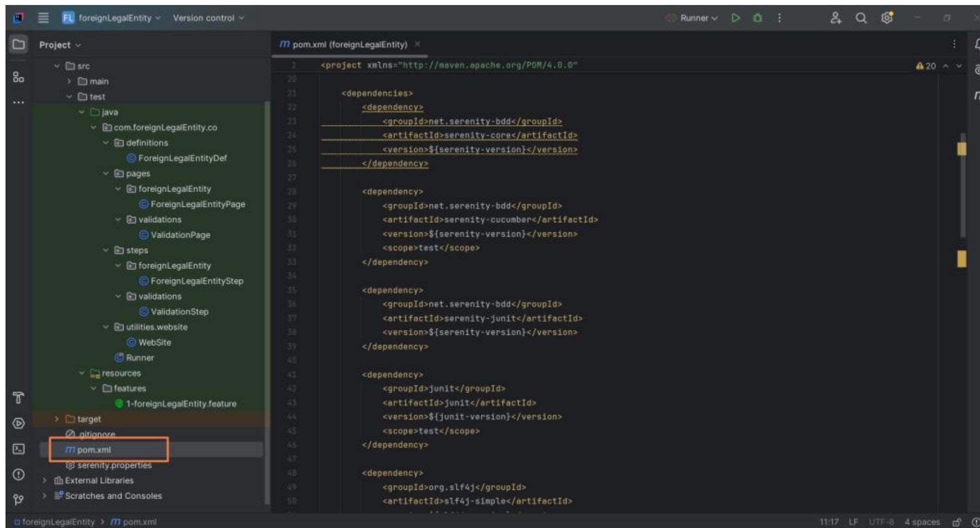
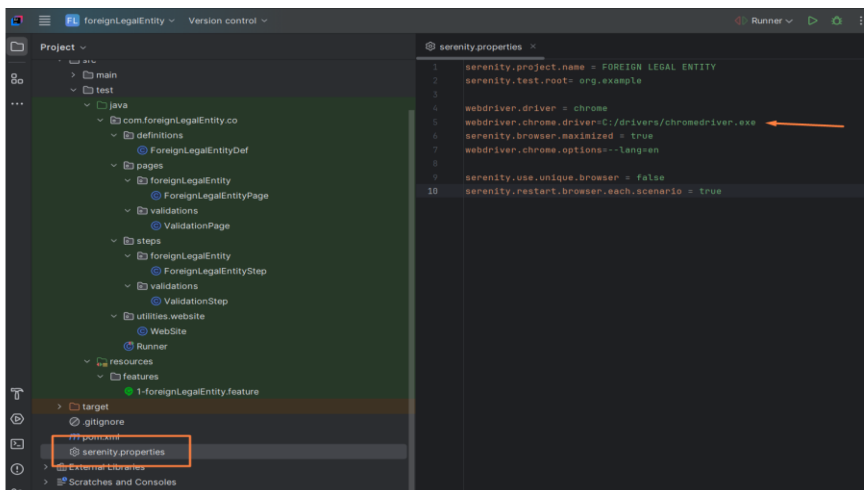


Figura 18

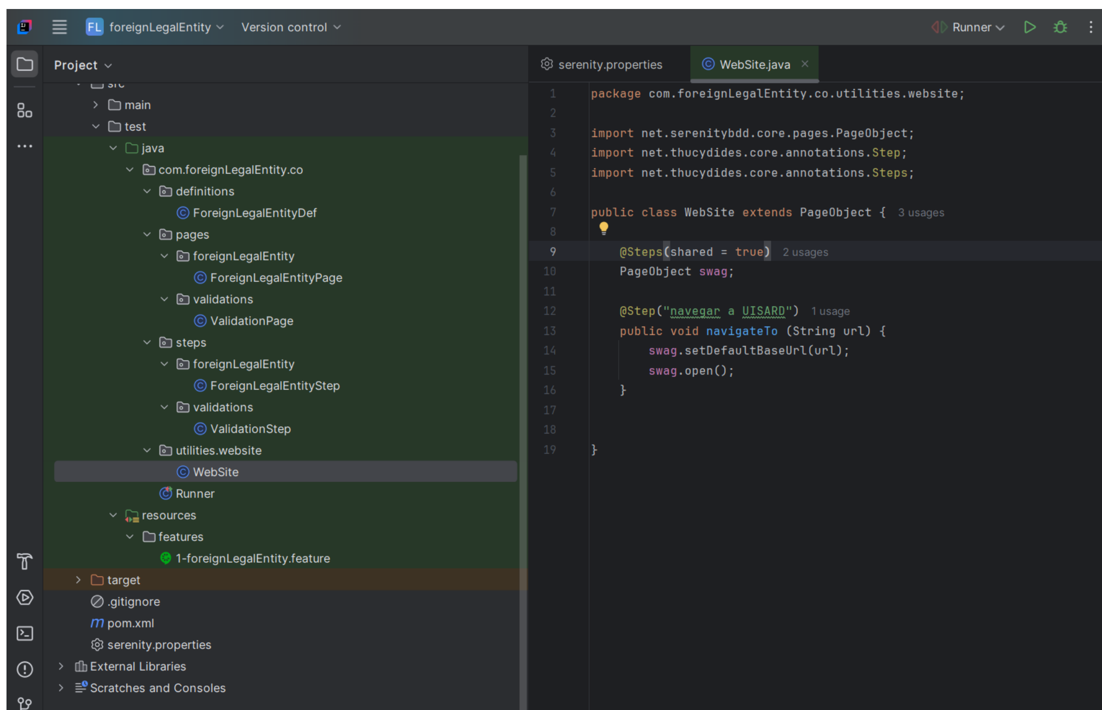
Configuración del WebDriver para el navegador



5.4.3.2 Configuración del entorno de pruebas. Una vez completada la instalación de las herramientas, se procedió con la configuración del entorno de pruebas a través de una clase centralizada encargada de gestionar el acceso inicial al sistema bajo prueba. Esta configuración se realizó mediante la clase WebSite, la cual permite definir dinámicamente la URL base del sistema y controlar la navegación inicial utilizada durante la ejecución de los escenarios automatizados. En la *Figura 19*, se presenta la implementación de dicha clase, donde se evidencia cómo, a partir de un método reutilizable, se establece la URL base de ejecución y se realiza la apertura del sistema, asegurando una inicialización controlada y consistente del entorno de pruebas para todos los escenarios automatizados.

Figura 19

Clase definida para navegar sitio web



De manera complementaria, se estableció una estrategia para el manejo de datos de prueba orientada a apoyar las ejecuciones repetibles y a reducir la dependencia de datos volátiles. Este enfoque permitió estandarizar la ejecución de las pruebas automatizadas y minimizar fallos asociados a condiciones externas.

5.4.4 Desarrollo y ejecución de los scripts de pruebas automatizadas

Una vez definidos los casos de prueba en lenguaje Gherkin y configurado el entorno técnico de automatización, se procedió con el desarrollo y la ejecución de los scripts de pruebas automatizadas. El objetivo de esta etapa fue materializar los escenarios diseñados en pruebas ejecutables, garantizando su correcta integración con el framework de automatización y asegurando la generación de evidencias confiables para el proceso de aseguramiento de la calidad.

5.4.4.1 Estructura del código de automatización. El desarrollo de los scripts de pruebas automatizadas se realizó siguiendo el patrón de diseño Page Object Model (POM), el cual permite separar la lógica de prueba de la interacción con la interfaz de usuario. Esta separación facilita la reutilización del código y mejora su mantenibilidad.

Implementación del patrón de diseño seleccionado

El paquete Definitions agrupa las definiciones de pasos que enlazan los escenarios escritos en Gherkin con la implementación en código, utilizando anotaciones Given, When y Then, como se evidencia en la *Figura 20*. Por su parte, el paquete Pages, presentado en la Figura X, encapsula los elementos y acciones asociados a cada pantalla del sistema, centralizando los selectores y reduciendo el impacto de cambios en la interfaz.

Figura 20

Ejemplo de clase en Definitions con anotaciones Cucumber

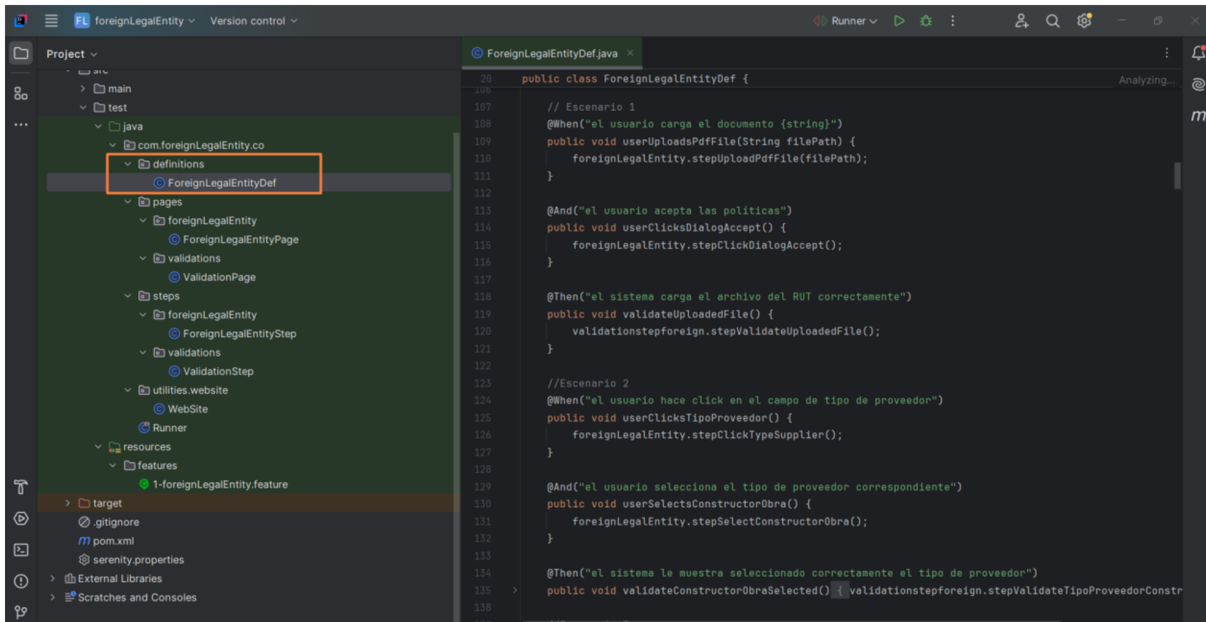
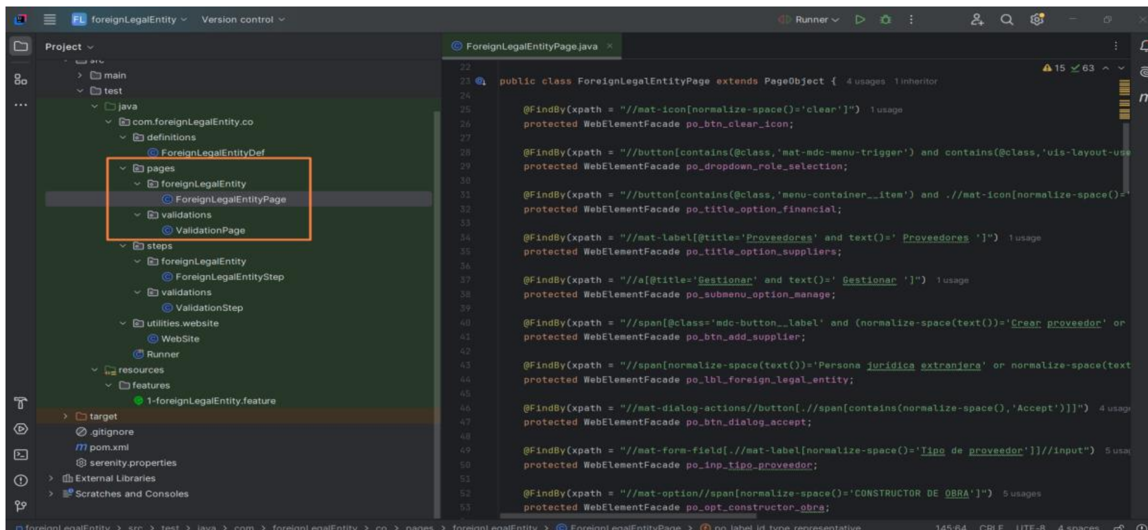


Figura 21

Ejemplo de Page Object Model con selectores



Asimismo, el paquete Steps, ilustrado en la *Figura 22*, implementa la lógica reutilizable que ejecuta las acciones y validaciones, mientras que el paquete Utilities concentra

configuraciones generales y componentes transversales necesarios para la ejecución del framework, especialmente se destaca la clase Runner, como se observa en la *Figura 23*.

Figura 22

Ejemplo de métodos reutilizables en Steps

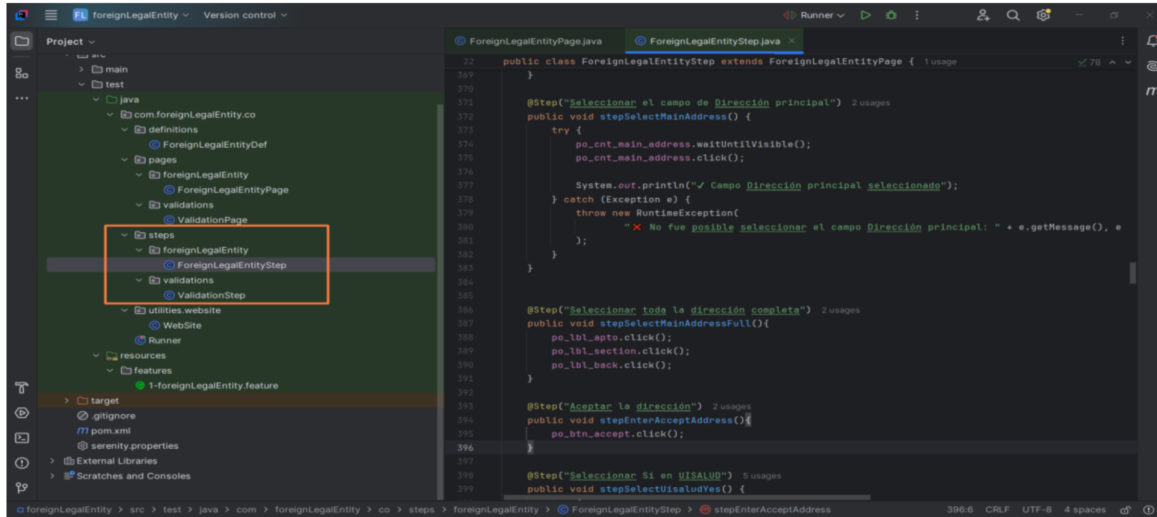
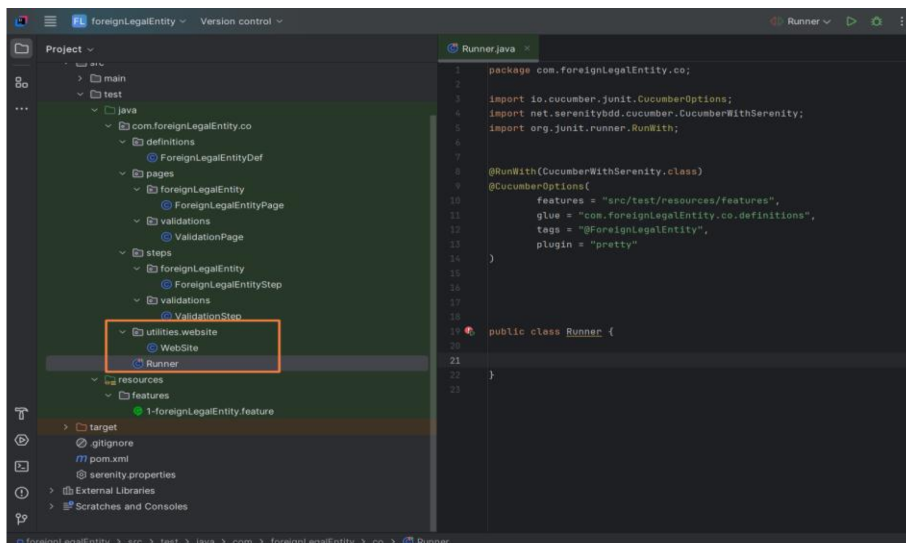


Figura 23

Estructura de la clase Runner

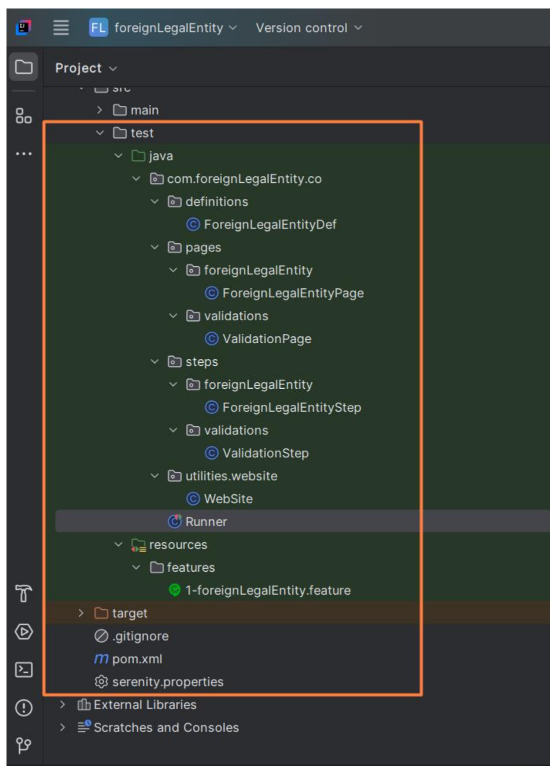


Estructura de carpetas y nomenclatura

Con el fin de mantener una organización clara del proyecto, se definió una estructura de carpetas y una nomenclatura estándar para los archivos. En la *Figura 24* se muestra cómo esta estructura facilita la navegación dentro del proyecto y asegura la trazabilidad con las historias de usuario.

Figura 24

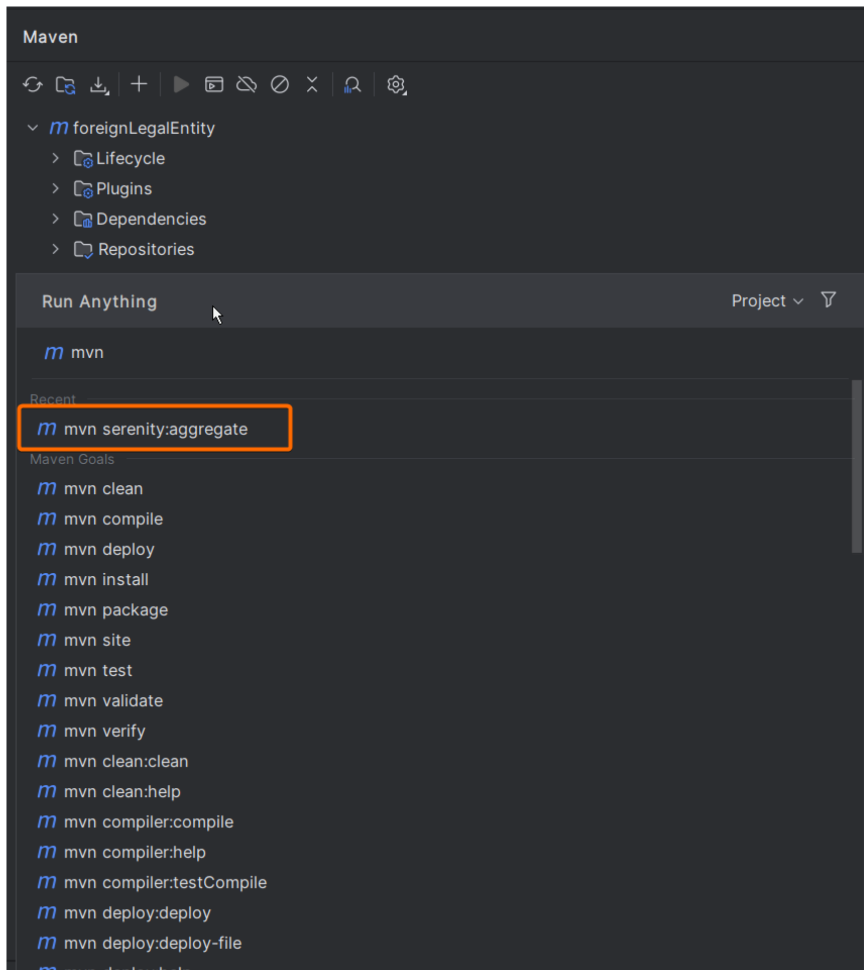
Estructura de carpetas del proyecto en el IDE



5.4.4.2 Ejecución y validación de los scripts. Una vez implementados los scripts, las pruebas automatizadas fueron ejecutadas mediante Maven. En la *Figura 25* se evidencia la ejecución de los comandos utilizados para correr las pruebas y generar los reportes correspondientes, sin embargo, el más destacado es *mvn serenity:aggregate.*

Figura 25

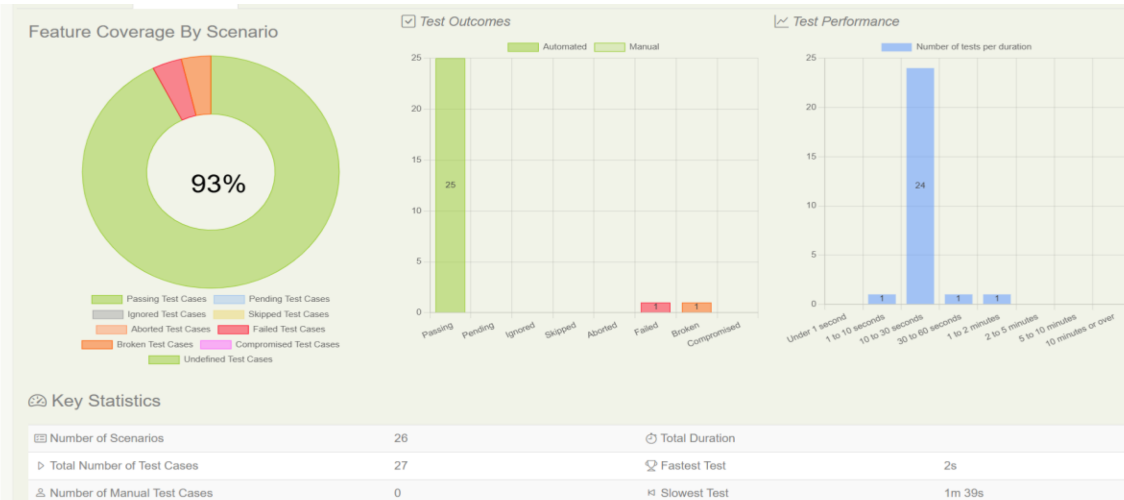
Comandos de Maven para generar el reporte



Posteriormente, Serenity BDD generó informes detallados que funcionan como documentación viva del comportamiento del sistema. En la *Figura 26* se presenta el resumen general de los reportes generados por Serenity BDD, donde se incluye información estadística sobre el total de escenarios ejecutados, así como los casos aprobados y fallidos y los tiempos de ejecución asociados.

Figura 26

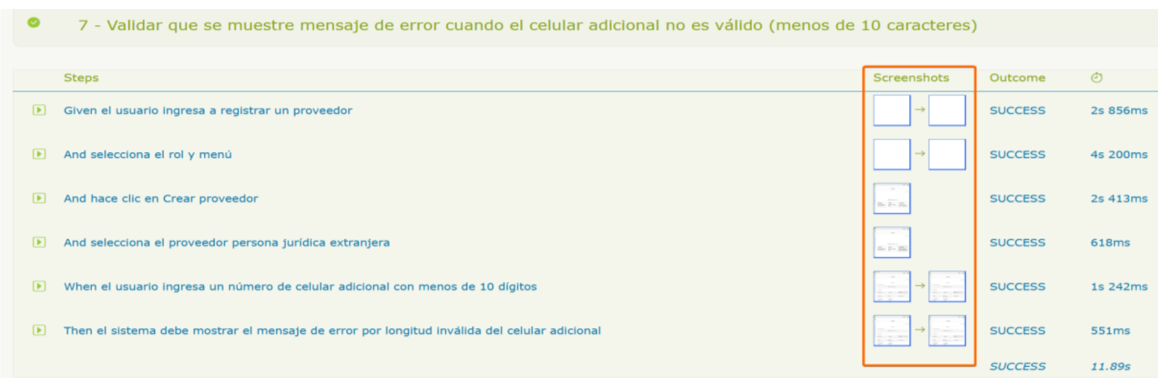
Resumen general de un reporte Serenity



De manera complementaria, en la *Figura 27* se muestra el detalle de los reportes por escenario, en el cual se evidencian las validaciones realizadas y las evidencias asociadas a cada caso de prueba ejecutado. Este enfoque elimina la necesidad de elaborar manualmente reportes.

Figura 27

Ejemplo del reporte detallado para cada escenario



5.4.4.3 Optimización, reutilización y mantenimiento del código. Posterior a la ejecución de las pruebas automatizadas, se realizaron actividades de optimización y mantenimiento del código con el propósito de mejorar su legibilidad y facilitar su reutilización en múltiples escenarios. En una primera etapa, algunos métodos presentaban una implementación funcional, pero con elementos que incrementaban el acoplamiento y afectaban la mantenibilidad, tales como validaciones embebidas y la definición de selectores directamente dentro del método. En la *Figura 28* se presenta una versión inicial de un método de carga de archivos que ilustra estas características.

Figura 28

Implementación inicial del método de carga de archivos

```
@Step("Cargar archivo en el campo de adjuntar del modal")  2 usages
public void stepUploadDocReps(String filePath) {

    // Selector directo en el método
    By uploadLocator = By.xpath(
            xpathExpression: "//div[contains(@class,'bottom-padding-modal-form-uis')]//input[@type='file' and @name='emitirArchivo']"
    );

    WebElement uploadInput = getDriver().findElement(uploadLocator);

    // Forzar visibilidad del input oculto antes de cargar el archivo
    JavascriptExecutor js = (JavascriptExecutor) getDriver();
    js.executeScript( script: "arguments[0].style.display='block';", uploadInput);

    // Validación simple
    if (!uploadInput.isDisplayed()) {
        throw new RuntimeException("El campo de carga no está visible para adjuntar el archivo.");
    }

    uploadInput.sendKeys(filePath);
}
```

Como resultado de la refactorización, la lógica se simplificó y se orientó a una implementación más concisa y reutilizable, manteniendo únicamente las acciones necesarias para facilitar la interacción con el campo de carga y la ejecución consistente del paso automatizado. En la *Figura 29* se presenta la versión refactorizada del método, la cual facilita su uso transversal en diferentes escenarios de prueba y contribuye a la estabilidad y evolución del framework conforme cambian los requerimientos del sistema.

Figura 29

Ejemplo de refactorización del método anterior

```
@Step("Cargar archivo {0} en el campo de adjuntar del modal")  2 usages
public void stepUploadDocReps(String filePath) {
    // Buscar el input oculto dentro del modal
    WebElement uploadInput = getDriver().findElement(po_btn_save_reps);

    // Hacer visible el input oculto
    ((JavascriptExecutor) getDriver()).executeScript( script: "arguments[0].style.display = 'block';", uploadInput);

    // Enviar la ruta del archivo
    uploadInput.sendKeys(filePath);
}
```

5.4.5 Entregables finales del proceso de automatización

Como resultado del proceso de automatización de pruebas descrito en las secciones anteriores, se generaron una serie de entregables que consolidan la implementación de la solución propuesta y permiten su uso, mantenimiento y evolución a lo largo del tiempo. Estos entregables constituyen evidencia tangible del proceso desarrollado e incluyen tanto los scripts automatizados versionados como la documentación técnica asociada, los cuales facilitan la trazabilidad, la reutilización del trabajo realizado y la transferencia de conocimiento dentro del equipo.

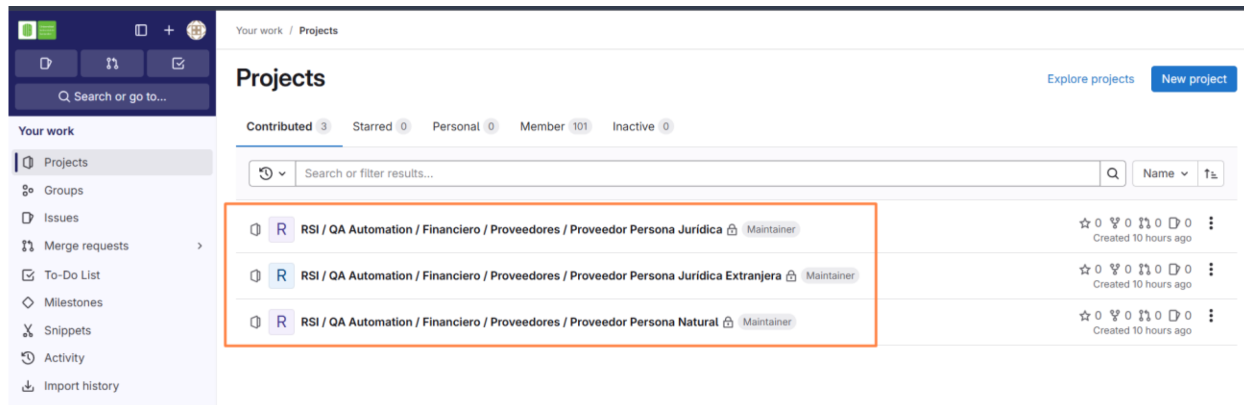
5.4.5.1 Publicación de los scripts automatizados en el repositorio. Los scripts automatizados desarrollados fueron versionados y publicados en un repositorio especializado de QA Automation, lo que permitió centralizar y gestionar de forma controlada el código asociado al proceso de automatización. Esta publicación garantiza la disponibilidad de los scripts para su ejecución, mantenimiento y evolución, además de facilitar el trabajo colaborativo y la trazabilidad de los cambios realizados.

En la *Figura 30* se presenta la estructura del repositorio de QA Automation, en la cual se observa el módulo correspondiente a Suppliers (Proveedores) y, dentro de este, los tres proyectos

de automatización asociados a los diferentes tipos de proveedor: persona natural, persona jurídica y persona jurídica extranjera. Esta organización permite una separación clara de los proyectos, facilitando su mantenimiento independiente y su escalabilidad futura.

Figura 30

Repositorio de QA Automation en donde se consolidaron los scripts



5.4.5.2 Estructura del framework de automatización. Como parte de los entregables del proceso de automatización, se definió y consolidó la estructura final del framework de pruebas automatizadas, la cual establece la organización de los componentes, las capas funcionales y la integración entre las herramientas utilizadas. Esta estructura permite comprender de manera global cómo se estructura la solución de automatización y cómo interactúan sus distintos elementos durante la ejecución de las pruebas.

En la *Figura 31,* se presenta el esquema de la estructura del framework de automatización implementado, donde se evidencian los componentes principales que lo conforman. En ella, se integran Selenium WebDriver para la interacción con la interfaz de usuario, Cucumber para la definición de escenarios en lenguaje Gherkin y Serenity BDD para la orquestación de la ejecución y la generación de reportes.

Asimismo, se muestra el uso de Maven como herramienta de construcción y ejecución, así como la organización de los paquetes que soportan la definición de escenarios, la lógica de prueba y las configuraciones generales del framework. Además de consolidar todo en un repositorio de GIT.

Figura 31

Estructura final del framework de automatización de pruebas

Nota. Adaptado de *Automation test with Serenity BDD*, por Moneyforward.Vn (s. f.). https://careers.moneyforward.vn/blog/automation-test-with-serenity-bdd

Esta representación refleja la estructura final adoptada para la automatización de pruebas y sirve como referencia del diseño implementado, evidenciando una separación clara de responsabilidades y una integración coherente de las herramientas seleccionadas.

6. Análisis de resultados

En esta sección se analizan los efectos prácticos de la automatización en términos de tiempo de ejecución, cobertura funcional, hallazgos detectados y generación de reportes, así como su impacto dentro de un contexto real de adopción, donde las pruebas automatizadas se integran a un sistema previamente validado mediante pruebas manuales.

Es importante señalar que la comparación entre pruebas manuales y pruebas automatizadas no se realizó sobre un sistema completamente nuevo, sino sobre un sistema previamente probado y estabilizado mediante ejecuciones manuales. En consecuencia, varios defectos fueron identificados y corregidos antes de la ejecución de las pruebas automatizadas. No obstante, esta condición refleja un escenario real de adopción de la automatización en proyectos existentes, donde las pruebas automatizadas se incorporan como una estrategia de regresión y validación continua, y no como un reemplazo inmediato de las pruebas manuales iniciales.

6.1 Resultados de la ejecución de pruebas automatizadas

Como resultado del proceso de automatización, se implementaron y ejecutaron 168 casos de prueba automatizados, distribuidos entre los diferentes tipos de proveedor: persona natural, persona jurídica y persona jurídica extranjera. Estos casos fueron seleccionados por su estabilidad, impacto funcional y necesidad de validación recurrente dentro del módulo de Proveedores.

La ejecución de las pruebas automatizadas evidenció diferencias en los tiempos de ejecución según el tipo de proveedor, asociadas principalmente a la complejidad de los flujos y a la cantidad de validaciones involucradas en cada caso. En la *Figura 32*, se presenta el tiempo de ejecución correspondiente a los casos automatizados para proveedores persona natural, en la

Figura 33 se muestra el tiempo de ejecución para proveedores persona jurídica, y en la *Figura 34* se evidencia el tiempo de ejecución asociado a proveedores persona jurídica extranjera.

Figura 32

Tiempo de ejecución de pruebas automatizadas para proveedor persona natural

Runner (com.naturalPerson.co) 18 min 38 sec
FO1 - Validar primer step 7 min 1 sec
FO2 - Validar segundo step 4 min 18 sec
FO3 - Validar tercer step 2 min 22 sec
FO4 - Validar cuarto step 4 min 57 sec

Figura 33

Tiempo de ejecución de pruebas automatizadas para proveedor persona jurídica

Runner (com.legalEntity.co) 20 min 2 sec
FO1 - Validar primer step 12 min 53 sec
FO2 - Validar segundo step 5 min 15 sec
FO4 - Validar tercer step 1 min 53 sec
FO5 - Validar casos adicionales y finales 808 ms

Figura 34

Tiempo de ejecución de pruebas automatizadas para proveedor persona jurídica extranjera

Runner (com.foreignLegalEntity.co) 8 min 3 sec
FO1 - Validar step información básica 8 min 3 sec

De manera global, la ejecución completa del conjunto de pruebas automatizadas presentó un tiempo total aproximado de 20 minutos, incluyendo la inicialización del entorno y la generación

automática de reportes. Este resultado demuestra que, independientemente del tipo de proveedor, la automatización permite validar de forma consistente y rápida los flujos críticos del sistema, habilitando su ejecución frecuente dentro de ciclos de regresión sin afectar los tiempos del proyecto.

6.2 Comparación entre pruebas manuales y pruebas automatizadas

Con el fin de evaluar el impacto real del proceso de automatización, se realizó una comparación entre los resultados obtenidos mediante pruebas manuales y pruebas automatizadas, considerando métricas de tiempo de ejecución, cobertura funcional y hallazgos detectados. Esta comparación se efectuó reconociendo que ambas estrategias cumplen roles complementarios dentro del aseguramiento de la calidad del software.

6.2.1 Análisis comparativo del tiempo de ejecución

Durante la fase de pruebas manuales del módulo de Proveedores, se ejecutaron 218 casos de prueba, lo que implicó un esfuerzo aproximado de 60 horas de trabajo, considerando la ejecución de los casos, la validación de resultados y la documentación de hallazgos.

En contraste, la ejecución de los 168 casos de prueba automatizados presentó un tiempo total cercano a los 45 minutos, equivalente a aproximadamente 0,75 horas. La reducción del tiempo de ejecución se calculó mediante la siguiente expresión:

$$Reducción\ de\ tiempo\ (\%)\ =\left(1-\frac{T_{auto}}{T_{manual}}\right)\ x\ 100$$

Donde:

- $T_{manual} =\ 60\ horas$
- $T_{auto} =\ 0{,}75\ horas$

Entonces,

$$Reducción\ \ de\ tiempo\ \approx\ 98{,}75\%$$

Este resultado evidencia una disminución sustancial del tiempo requerido para ejecutar pruebas de regresión, lo que permite al equipo realizar validaciones frecuentes sin afectar los tiempos de entrega del proyecto.

6.2.2 Análisis comparativo de la cobertura de pruebas

Las pruebas manuales cubrieron un total de 218 casos de prueba, incluyendo escenarios principales, alternos y validaciones específicas. A partir de este conjunto, se seleccionaron 168 casos viables para automatización, representando aproximadamente:

$$Cobertura\ automatizada\ (\%)\ =\ \left(\frac{CP_{auto}}{CP_{manual}}\right)\ x\ 100$$

Donde:

- $CP_{manual}\ =\ 218$
- $CP_{auto}\ =\ 168$

Entonces,

$$Cobertura\ automatizada\ \ \approx\ 77\%$$

Este porcentaje refleja que una parte significativa de la funcionalidad del módulo fue cubierta mediante pruebas automatizadas, manteniendo los flujos críticos y repetitivos. Los casos restantes continuaron siendo validados de forma manual debido a su dependencia de validaciones visuales, escenarios poco estables o flujos exploratorios, lo cual reafirma el carácter complementario de ambas estrategias de prueba.

6.2.3 Análisis comparativo de hallazgos y defectos

Durante la ejecución de pruebas manuales se reportaron 38 incidencias, mientras que la ejecución de las pruebas automatizadas permitió identificar 25 incidencias. Esta diferencia no responde a una disminución en la capacidad de detección de defectos por parte de la automatización, sino al estado del sistema al momento de su ejecución, el cual ya había sido parcialmente estabilizado mediante pruebas manuales previas.

Para analizar esta relación, se calculó el índice de detección relativa de defectos, definido como:

$$índice\ de\ detección\ (\%)\ =\ \frac{D_{auto}}{D_{manual}}$$

Donde:

- $D_{manual} =\ 38\ defectos\ en\ pruebas\ manuales$
- $D_{auto} =\ 25\ defectos\ en\ pruebas\ automatizadas$

Entonces,

$$índice\ de\ detección\ \approx\ 0{,}65\%$$

Este valor indica que las pruebas automatizadas lograron identificar aproximadamente un 66 % de defectos a comparación de los encontrados en las pruebas manuales, aun cuando el sistema se encontraba en una versión más estable.

6.3 Análisis de reportes automatizados por tipo de proveedor

Con el fin de evitar redundancias con la sección de implementación, los resultados relacionados con los reportes automatizados se analizan desde una perspectiva funcional y comparativa.

En la *Figura 35* se presenta un ejemplo del reporte generado para proveedores persona natural, donde se evidencian los escenarios ejecutados, los resultados obtenidos y las evidencias asociadas. De manera similar, en la *Figura 36* se muestra el reporte correspondiente a proveedores persona jurídica, y en la *Figura 37* se presenta el reporte asociado a proveedores persona jurídica extranjera.

Figura 35

Reporte automatizado de ejecución para proveedor persona natural

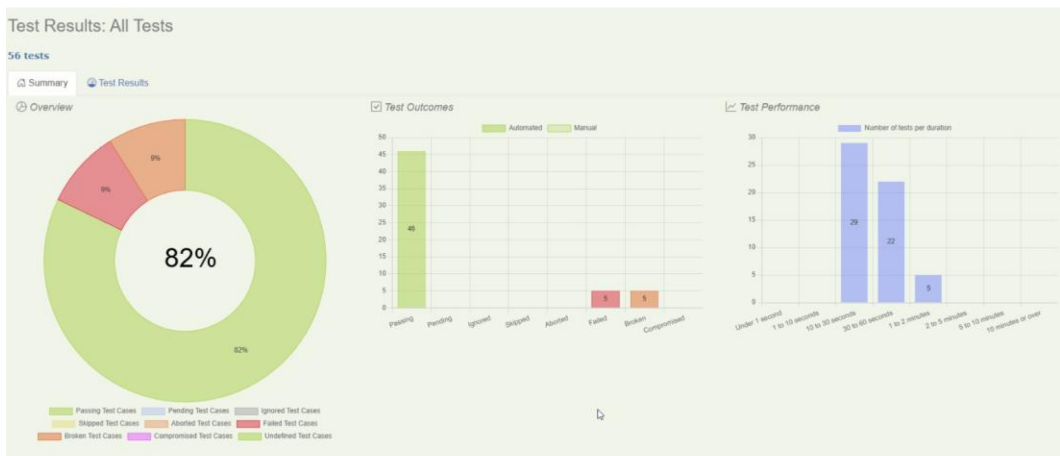


Figura 36

Reporte automatizado de ejecución para proveedor persona jurídica

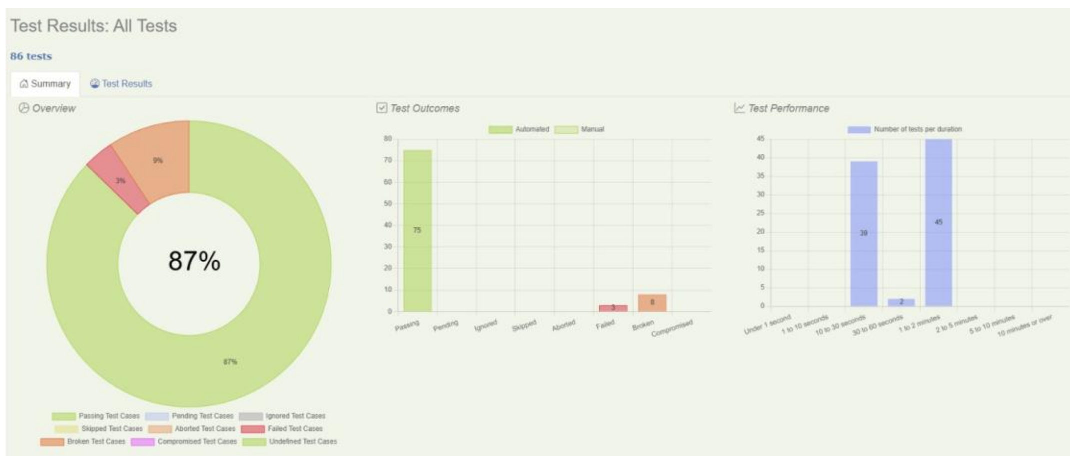
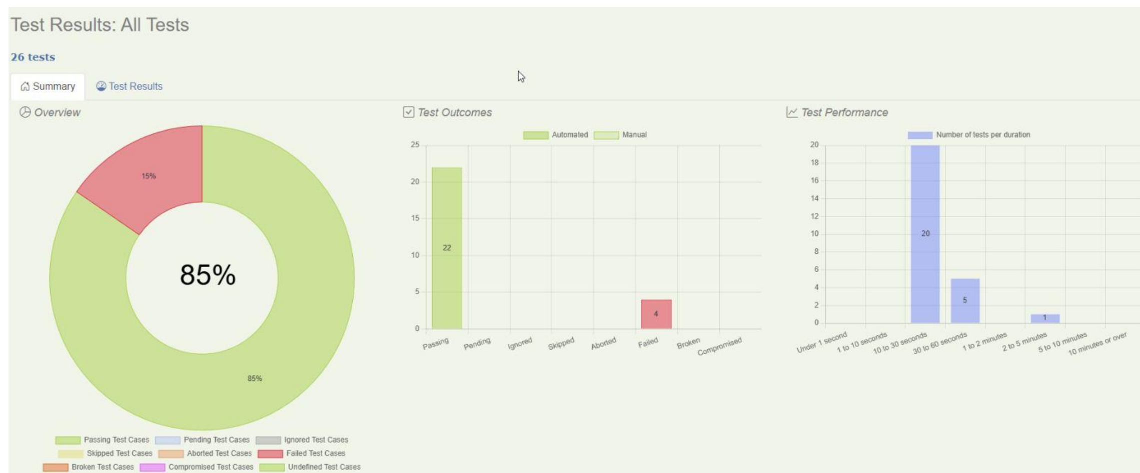


Figura 37

Reporte automatizado de ejecución para proveedor persona jurídica extranjera



El análisis de estos reportes permitió identificar diferencias en la cantidad de escenarios ejecutados, el número de validaciones involucradas y la complejidad de los flujos por tipo de proveedor. Asimismo, estos informes facilitaron la comparación entre ejecuciones y la validación de correcciones aplicadas, consolidándose como una herramienta clave para el seguimiento del estado del sistema durante los ciclos de regresión. Cabe mencionar, que como el acceso al contenido de los informes es privado, podrán consultarse a detalle con la debida autorización de la DTIC.

6.4 Gestión y validación de incidencias derivadas de la automatización

Como resultado de la ejecución de las pruebas automatizadas, se identificaron incidencias funcionales que fueron reportadas en nuevas Historias de Usuario, orientadas al rediseño y ajuste de funcionalidades del módulo de Proveedores. En la *Figura 38*, se presentan algunas de estas historias de usuario, por medio de las cuales se les dio solución a los issues encontrados.

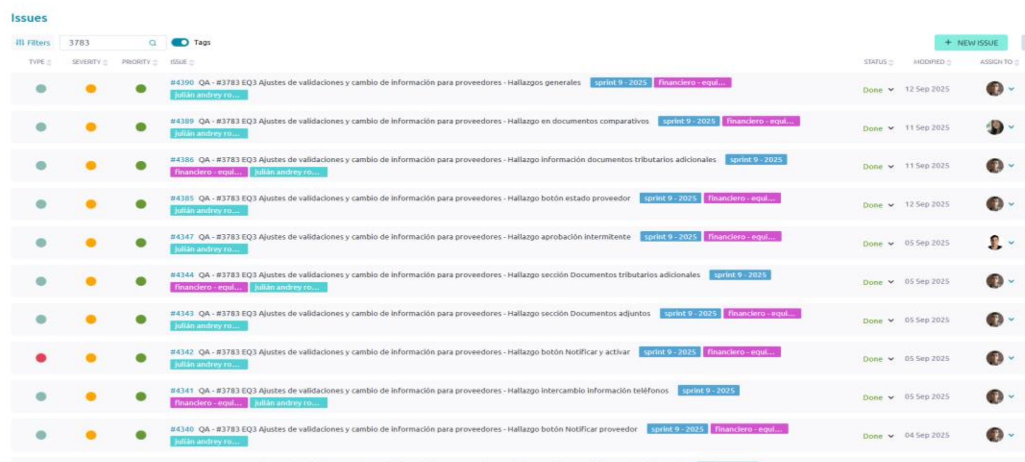
Figura 38

Historias de usuario relacionadas al rediseño de proveedores

Una vez implementadas las correcciones por parte del equipo de desarrollo, se realizó el seguimiento correspondiente y se ejecutaron nuevamente las pruebas automatizadas para validar los ajustes realizados. En la *Figura 39* se evidencia el registro y seguimiento de las incidencias reportadas. Luego de ello se comprobó que todo había sido resuelto en su posterior validación exitosa mediante la ejecución de las pruebas automatizadas.

Figura 39

Ejemplo de algunos issues reportados en la herramienta de gestión



Este proceso permitió cerrar el ciclo de aseguramiento de la calidad de forma más eficiente, utilizando la automatización no solo como mecanismo de detección de defectos, sino también como herramienta de verificación de correcciones y control de regresión frente a cambios funcionales introducidos en el sistema.

7. Conclusiones

La implementación del proceso de automatización de pruebas en el módulo de Proveedores del Sistema de Información Financiero (SIF) permitió construir y evaluar un enfoque de aseguramiento de la calidad pensado para un entorno real de desarrollo. A lo largo del proyecto se evidenció que la automatización, cuando se aplica de forma consciente y alineada con las necesidades del sistema, se convierte en una herramienta clave para mejorar la calidad del software y hacer más eficientes los procesos de validación.

Uno de los aportes más importantes de este trabajo fue la definición de un proceso de automatización que supera la idea de simplemente ejecutar scripts. Dicho proceso integra de manera ordenada actividades como el análisis y selección de casos de prueba, el diseño de escenarios en lenguaje Gherkin, la configuración técnica del entorno, la ejecución de pruebas, el análisis de resultados y la gestión de incidencias. Gracias a este enfoque, la automatización logró aportar valor al equipo de QA y dejó de ser un ejercicio aislado o meramente experimental.

Los resultados obtenidos muestran una reducción considerable en los tiempos de ejecución al comparar las pruebas manuales con las pruebas automatizadas. Mientras que la validación manual del módulo de Proveedores implicó un esfuerzo significativo en tiempo y recursos humanos, la automatización permitió ejecutar un conjunto representativo de pruebas de regresión

en cuestión de minutos. Esto facilitó su repetición frecuente y su incorporación natural dentro del ciclo de desarrollo del software.

De igual forma, la automatización demostró su utilidad como mecanismo de validación continua, incluso en un escenario donde el sistema ya había sido previamente estabilizado mediante pruebas manuales. La aparición de nuevas incidencias confirma que la automatización no solo apoya las fases iniciales de prueba, sino que cumple un papel fundamental en la detección de regresiones y comportamientos inesperados derivados de ajustes funcionales o rediseños del sistema.

Otro beneficio relevante fue la generación automática de reportes, los cuales permitieron consolidar los resultados de las pruebas de manera clara, organizada y trazable. Esto eliminó la necesidad de realizar procesos manuales de documentación, optimizando el tiempo del equipo de QA y facilitando tanto el análisis de resultados como la comunicación de hallazgos con los diferentes actores del proyecto.

Finalmente, el trabajo permitió consolidar un framework de automatización escalable y mantenible (Presentado en la *Figura 31*), construido sobre buenas prácticas y el uso de herramientas ampliamente conocidas en la industria. Esto no solo respalda la solución implementada, sino que también abre la puerta a la expansión del proceso de automatización hacia otros módulos del sistema y otros equipos de la organización, contribuyendo al fortalecimiento del aseguramiento de la calidad dentro de la DTIC.

8. Trabajo futuro

Aunque el proceso de automatización implementado cumplió los objetivos propuestos para el módulo de Proveedores del Sistema de Información Financiero (SIF), el trabajo realizado

permitió identificar oportunidades claras para ampliar y fortalecer el impacto de la automatización de pruebas dentro de la DTIC.

Una línea de trabajo futuro consiste en extender la automatización a otros módulos e interfaces del SIF, tomando como referencia el proceso definido en este trabajo. Esto permitiría estandarizar la automatización, aumentar la cobertura de pruebas y reforzar los ciclos de regresión en funcionalidades adicionales.

De igual forma, se identifica como una oportunidad relevante la adopción de este proceso por parte de otros equipos de la DTIC, como por ejemplo equipos de Talento humano o Gestión documental, lo cual contribuiría a unificar criterios de calidad y a reducir la dependencia de pruebas exclusivamente manuales.

También se plantea la exploración de la automatización de pruebas en aplicaciones móviles, considerando los retos propios de este tipo de entornos y la necesidad de adaptar las herramientas y estrategias utilizadas.

Como proyección adicional, resulta clave avanzar hacia una integración más profunda de la automatización dentro de flujos de integración continua (CI/CD), de manera que las pruebas automatizadas se conviertan en un componente habitual del proceso de despliegue del software, apoyando la detección temprana de errores y la estabilidad del sistema.

Referencias Bibliográficas

Adzic, G. (2011). Specification by example: How successful teams deliver the right software. Manning Publications.

Agile Alliance. (2014). Agile 101. http://www.agilealliance.org/

Allure Report. (s. f.). Documentación de Allure Report. https://allurereport.org/es/docs/

Apache Maven. (s. f.). Maven project. Apache Software Foundation. https://maven.apache.org/

Astels, D. (2003). Test-driven development: A practical guide. Prentice Hall.

Beck, K. (2003). Test-driven development: By example. Addison-Wesley.

Behave. (s. f.). Behave documentation. Read the Docs. https://behave.readthedocs.io/en/latest/

Bosch, J. (2014). Continuous software engineering: Agile and beyond. Springer.

BrowserStack. (s. f.). Screenplay pattern approach in Selenium. https://www.browserstack.com/guide/screenplay-pattern-approach-in-selenium

BrowserStack. (s. f.). Tutorial de pruebas automatizadas con SpecFlow. https://www.browserstack.com/guide/specflow-automated-testing-tutorial

Calidad y Software. (s. f.). Calidad. http://www.calidadysoftware.com/

Cohn, M. (2004). User stories applied: For agile software development. Addison-Wesley.

Cohn, M. (2009). An overview of user stories. Mountain Goat Software.

Coplien, J., & Bjørnvig, G. (2010). Lean architecture: For agile software development. Wiley.

Crispin, L., & Gregory, J. (2009). Agile testing: A practical guide for testers and agile teams. Addison-Wesley.

DevExpress. (s. f.). TestCafe: Automated end-to-end testing. https://testcafe.io/

Digital.ai. (s. f.). Automated testing tools. https://digital.ai/es/glossary/automated-testing-tools/

Digital.ai. (s. f.). Working with Serenity BDD framework: An overview. https://digital.ai/es/catalyst-blog/working-with-serenity-bdd-framework-an-overview/

Dookhun, A. S., & Nagowah, L. (2019). Assessing the effectiveness of test-driven development and behavior-driven development in an industry setting. In Proceedings of the IEEE international conference (pp. 365–370). IEEE. https://doi.org/10.1109/ICCIKE47802.2019.9004328

Freeman, S., & Pryce, N. (2010). Growing object-oriented software, guided by tests. Addison-Wesley.

Fuentes, J. R. L. (2015). Desarrollo de software ágil: Extreme programming y Scrum. IT Campus Academy.

García León, D., & Beltrán Benavides, A. (1995). Un enfoque actual sobre la calidad del software. Acimed, 3(3), 40–42.

Garg, N. (2014). Test automation using Selenium WebDriver with Java: Step by step guide.

GeeksforGeeks. (s. f.). Command pattern. https://www.geeksforgeeks.org/command-pattern/

Humble, J., & Farley, D. (2010). Continuous delivery: Reliable software releases through build, test, and deployment automation. Addison-Wesley.

IEEE. (1991). IEEE standard glossary of software engineering terminology (IEEE Std 610.12-1990).

ISO/IEC. (2011). ISO/IEC 25010:2011 systems and software engineering — Systems and software quality requirements and evaluation (SQuaRE) — System and software quality models.

ISTQB®. (s. f.). Welcome to ISTQB®. https://www.istqb.org/

JBehave. (s. f.). JBehave: Framework de desarrollo basado en historias. https://jbehave.org/

Keogh, L. (2014). Using examples in conversation to illustrate behaviour. Agile Journal.

Kovalenko, D. (2014). Selenium design patterns and best practices. Packt Publishing.

Leotta, M., Clerissi, D., Ricca, F., & Spadaro, C. (2013). Improving test suites maintainability with the page object pattern: An industrial case study. In 2013 IEEE sixth international conference on software testing, verification and validation workshops (pp. 108–113). IEEE.

Marick, B. (2003). Agile software testing: An overview. Agile Journal.

Matsinopoulos, P. (2020). Practical test automation: Learn to use Jasmine, RSpec, and Cucumber effectively for your TDD and BDD. Apress.

Meszaros, G. (2007). xUnit test patterns: Refactoring test code. Addison-Wesley.

Microsoft. (s. f.). ¿Qué es el control de versiones?. Microsoft Learn. https://learn.microsoft.com/es-es/devops/develop/git/what-is-version-control

Naik, K., & Tripathy, P. (2011). Software testing and quality assurance: Theory and practice. John Wiley & Sons.

North, D. (2009). Introducing BDD: A second-generation agile methodology. http://dannorth.net/

Pichler, R. (2012). Agile product management with Scrum: Creating products that customers love. Addison-Wesley.

Pragma. (s. f.). Patrón de diseño en la automatización con Screenplay. https://www.pragma.com.co/academia/lecciones/patron-de-dise%C3%B1o-en-la-automatizacion-con-screenplay

QA Touch. (s. f.). Page object model in Selenium. https://www.qatouch.com/blog/page-object-model-in-selenium/

QAlified. (s. f.). Framework de automatización de pruebas. https://qalified.com/

QAlified. (s. f.). Introducción a Selenium testing. https://qalified.com/es/blog/introduccion-a-selenium-testing/

Rahman, M. (2019). Behavior-driven development: Challenges and best practices. IEEE Software Journal.

Refactoring Guru. (s. f.). Command. https://refactoring.guru/es/design-patterns/command/

Roa, M. T. (s. f.). Guía del equipo QA. https://wikirsi.uis.edu.co/en/introQA

Serenity BDD. (s. f.). Living documentation en Serenity BDD. https://serenity-bdd.github.io/docs/reporting/living_documentation

Singular. (s. f.). Playwright: Primeros pasos y ejemplos de uso. https://www.sngular.com/es/insights/128/playwright-primeros-pasos-y-ejemplos-de-uso

Smart, J. F. (2017). BDD in action: Behavior-driven development for the whole software lifecycle. Manning Publications.

Smart, J. F., & Molak, J. (2023). BDD in action: Behavior-driven development for the whole software lifecycle. Simon & Schuster.

TIVIT. (s. f.). Herramientas de automatización: Qué son y cuáles son las mejores opciones. https://latam.tivit.com/blog/herramientas-de-automatizacion

Vitelli, J. (s. f.). ¿Qué es Extent Report?. https://jvitelli.com/portfolio/extent-report/

Wynne, C., & Hellesoy, A. (2017). The Cucumber book: Behaviour-driven development for testers and developers. Pragmatic Bookshelf.