

Universidad Tecnológica de La Habana

“José Antonio Echeverría”



**Facultad de Ingeniería Informática**

**Diseño e implementación de pruebas automatizadas al módulo Carga y Descarga del sistema de planificación de recursos de Habana Club International.**

*Informe prácticas profesionales 1*

**Autores:** Pedro Pablo Chevalier Pérez ([pchevalier@ceis.cujae.edu.cu](mailto:pchevalier@ceis.cujae.edu.cu))

Christian Peña González ([cpena@ceis.cujae.edu.cu](mailto:cpena@ceis.cujae.edu.cu))

**Tutor:** Dr. C. Raisa Socorro Llanes ([raisa@ceis.cujae.edu.cu](mailto:raisa@ceis.cujae.edu.cu))

La Habana, Cuba

Abril, 2025

Resumen

Abstract

Índice General

[Introducción 1](#_Toc194849866)

[1 Capítulo 1: Marco Teórico 5](#_Toc194849867)

[1.1 Procesos de las pruebas de software 5](#_Toc194849868)

[1.2 Tipos de pruebas de software 6](#_Toc194849869)

[1.3 Pruebas de software en sistemas de planificación de recursos empresariales 8](#_Toc194849870)

[1.4 Diferencias entre las pruebas manuales y las pruebas automatizadas 9](#_Toc194849871)

[1.5 Metodologías de pruebas de software 10](#_Toc194849872)

[1.6 Herramientas para las pruebas automatizadas 11](#_Toc194849873)

[1.7 Patrones de diseño para pruebas automatizadas 12](#_Toc194849874)

[1.8 Inteligencia Artificial en pruebas de software: 14](#_Toc194849875)

[2 Capítulo 2: Propuesta de solución 17](#_Toc194849876)

[2.1 Introducción al capítulo 17](#_Toc194849877)

[2.2 Análisis del módulo "Carga-Descarga" 17](#_Toc194849878)

[2.3 Complejidades técnicas identificadas 20](#_Toc194849879)

[2.4 Metodología para el diseño de casos de prueba 21](#_Toc194849880)

[2.5 Organización y gestión con Squash TM 22](#_Toc194849881)

[2.6 Arquitectura de la solución de automatización 23](#_Toc194849882)

[2.7 Estrategia de parametrización y gestión de datos 26](#_Toc194849883)

[2.8 Integración con el proceso de desarrollo 28](#_Toc194849884)

[2.9 Implementación de mantenimiento asistido por IA 30](#_Toc194849885)

[2.10 Conclusiones del capítulo 31](#_Toc194849886)

[Referencias Bibliográficas 34](#_Toc194849887)

# Introducción

En las últimas décadas, el software se ha consolidado como un pilar fundamental de la sociedad moderna, transformando radicalmente sectores como la salud, la educación, las finanzas y la comunicación[1]. Su influencia se extiende desde los dispositivos móviles que se utilizan diariamente hasta los complejos sistemas que controlan infraestructuras críticas, redefiniendo cómo las personas interactúan con la tecnología y entre sí mismas. Esta transformación digital ha generado un ecosistema donde el software ya no es una simple herramienta auxiliar, sino un componente esencial para el funcionamiento de la civilización actual.

La dependencia del software ha puesto de manifiesto la vital importancia de garantizar su calidad. Un error en un sistema bancario puede paralizar transacciones financieras en todo el mundo; una falla en equipos médicos podría comprometer vidas humanas; un defecto en sistemas de control industrial podría desencadenar catástrofes ambientales[2]. Las implicaciones económicas de estas fallas son igualmente alarmantes, con estimaciones que sitúan las pérdidas anuales en decenas de miles de millones de dólares[3].

Más allá del impacto financiero inmediato, el software defectuoso genera una cascada de consecuencias negativas: ciclos de desarrollo que se alargan indefinidamente por la necesidad constante de corregir errores; la erosión gradual pero implacable de la confianza de los usuarios; la frustración de los clientes traducida en abandono; y en casos extremos, disputas legales que pueden amenazar la supervivencia misma de las organizaciones[4, 5].

El ciclo de vida del desarrollo de software(SDLC por sus siglas en inglés) ha evolucionado para adaptarse a las necesidades actuales de la industria, priorizando la agilidad, la integración continua y la entrega rápida de software de calidad. En la actualidad, metodologías como DevOps, Agile y CI/CD(continuous integration and continuous delivery) han reemplazado en gran medida los enfoques tradicionales en cascada, promoviendo iteraciones frecuentes, retroalimentación constante y pruebas automatizadas en cada fase del desarrollo[6, 7]. Desde la planificación y el diseño hasta la implementación y el mantenimiento, el aseguramiento de la calidad (QA) juega un papel clave al integrarse de manera continua en el ciclo de vida[6].

El aseguramiento de la calidad del software representa un enfoque integral y proactivo que atraviesa todo el ciclo de vida del desarrollo. Lejos de ser una fase aislada que se ejecuta al final del proceso, constituye una filosofía orientada a prevenir defectos desde las etapas iniciales de conceptualización hasta el despliegue y mantenimiento. Este paradigma establece que la calidad no se puede "probar" posteriormente en un producto, sino que debe "construirse" desde su concepción[1, 2, 6].

En este contexto, las pruebas de software emergen como una disciplina fundamental cuyo objetivo principal no es demostrar la ausencia de errores, sino descubrir su presencia[2]. Esta distinción, aunque sutil, representa un cambio fundamental en el enfoque: las pruebas eficaces requieren una mentalidad orientada a la detección de problemas más que a la validación de funcionalidades ya implementadas.

Además de la estrategia de pruebas, es esencial contar con un equipo de pruebas capacitado y con experiencia. Los probadores deben tener un profundo conocimiento del dominio de la aplicación, así como habilidades técnicas para diseñar y ejecutar pruebas efectivas[2, 7]. La colaboración entre desarrolladores y probadores es crucial para asegurar que las pruebas sean integrales y que los defectos se identifiquen y solucionen rápidamente[8].

La automatización de las pruebas ha revolucionado esta disciplina, permitiendo la ejecución sistemática, consistente y repetible de casos de prueba sin intervención humana directa. No obstante, sería erróneo considerar la automatización como una panacea universal. Las pruebas manuales conservan un valor irreemplazable en aspectos como la evaluación de usabilidad, las pruebas exploratorias y aquellas situaciones donde la intuición y experiencia humanas resultan insustituibles. La estrategia óptima consiste en una combinación equilibrada de ambos enfoques, aprovechando las fortalezas de cada uno para crear un ecosistema de calidad robusto y eficiente[9, 10].

Esta transición hacia metodologías ágiles y DevOps en la industria del desarrollo de software ha evidenciado la necesidad crítica de integrar pruebas automatizadas como parte fundamental del ciclo de desarrollo. Sin embargo, múltiples organizaciones siguen operando sin incorporar estas prácticas de manera sistemática en sus proyectos, lo que genera ineficiencias operativas y riesgos potenciales para la calidad del producto final.

Este es precisamente el caso de **AVANGENIO**, una empresa de desarrollo de software que, hasta hace poco, llevaba a cabo sus proyectos sin contar con equipos dedicados a la automatización de pruebas. Los procesos de verificación y validación se realizaban predominantemente de forma manual, lo que implicaba limitaciones significativas en términos de cobertura, consistencia y capacidad de regresión. Esta situación, común en muchas empresas del sector, suele derivar de factores como la percepción de alto costo inicial de implementación, la escasez de personal especializado o la subestimación del retorno de inversión de las pruebas automatizadas.

Estas limitaciones en la eficiencia y la cobertura de las pruebas provocaron el retraso en entregas anteriores de los módulos del sistema de gestión para Habana Club International, tomándose la decisión de añadir un equipo para realizar pruebas automatizadas al nuevo módulo en desarrollo, llamado “Carga y Descarga”.

Partiendo de la **situación problemática** anteriormente descrita, es posible plantear el **problema a resolver**:

¿Cómo se puede diseñar e implementar un sistema de pruebas automatizadas para el nuevo módulo en desarrollo que sea compatible con las tecnologías utilizadas?

Para resolver la problemática planteada, se propone el siguiente **objetivo general**:

Desarrollar un plan de pruebas automatizadas para el nuevo módulo garantizando la compatibilidad con las tecnologías utilizadas para su desarrollo.

Para cumplir con el objetivo general planteado, se definen los **objetivos específicos** siguientes:

1. Identificar los elementos fundamentales para desarrollar un sistema de pruebas automatizadas compatible con las tecnologías utilizadas en la creación del sistema de gestión de recursos.
2. Diseñar los casos de prueba para el módulo “Carga y Descarga” que garanticen la explotación de las funcionalidades
3. Implementar los casos de prueba siguiendo patrones de diseño que favorezcan la reutilización y el mantenimiento de los conjuntos de prueba.
4. Documentar el proceso de diseño de casos de prueba para el módulo en cuestión.

# Capítulo 1: Marco Teórico

En el presente capítulo se abordan los conceptos fundamentales de las pruebas de software y las metodologías utilizadas para su integración en el desarrollo de software. Además, se presenta la automatización de las pruebas de software como una necesidad dentro del proceso de desarrollo de software. Por último, se hace un análisis de la aplicabilidad de los temas tratados a la solución de la problemática planteada.

## Procesos de las pruebas de software

La calidad de un producto de software es determinada por el grado de cumplimiento de las necesidades del cliente y los usuarios. Según [1] es *“proceso eficaz de software que se aplica de manera que crea un producto útil que proporciona valor medible a quienes lo producen y a quienes lo utilizan”.*

​Las **pruebas de software** comprenden un conjunto de procesos y técnicas diseñados para evaluar y verificar la calidad de un producto informático. Su propósito esencial es identificar y corregir defectos en el software antes de su implementación en producción, minimizando riesgos y asegurando su fiabilidad. Integrar estas pruebas en el ciclo de vida del desarrollo de software (SDLC) es crucial, ya que la detección temprana de anomalías puede reducir significativamente las iteraciones en el proceso de desarrollo y optimizar el uso de recursos[11].

El proceso de pruebas tiene cuatro etapas fundamentales: planificación, diseño, ejecución y análisis, cada una de ellas desempeña un papel fundamental en el propósito de garantizar la calidad del producto[1].

En la fase de planificación se definen los objetivos, el alcance y los criterios de aceptación del software. Además, se identifican los recursos humanos y tecnológicos necesarios, se asignan roles y responsabilidades, y se elabora un cronograma que contemple hitos y plazos. Una planificación meticulosa permite prever riesgos potenciales y establecer estrategias para mitigarlos, asegurando que las pruebas se realicen de manera organizada y eficiente[2, 12].

Una vez definidos los objetivos, se procede a la fase de diseño, donde se elaboran los casos de prueba y se determinan las condiciones y escenarios que deben evaluarse. Esto implica identificar los requisitos funcionales y no funcionales, así como definir los datos de entrada y los resultados esperados. Un buen diseño de pruebas debe ser exhaustivo y adaptable, permitiendo cubrir tanto situaciones comunes como escenarios excepcionales que podrían afectar el desempeño del software[2, 8].

A partir de los casos de prueba diseñados los equipos de testing ejecutan las pruebas de forma manual o automatizada, utilizando herramientas específicas según las necesidades del proyecto. Durante la ejecución se registra el comportamiento del sistema, se capturan evidencias y se documentan los resultados. Esta fase es crucial, ya que permite identificar defectos y fallos en el software, proporcionando la información necesaria para corregirlos antes de su implementación definitiva[2].

La última etapa consiste en analizar y evaluar los resultados obtenidos durante la ejecución de las pruebas. Se revisan los informes y se clasifican los defectos según su severidad e impacto en el sistema. Este análisis permite determinar si el software cumple con los estándares de calidad requeridos y si está listo para pasar a la siguiente fase del ciclo de vida del desarrollo. Además, se generan recomendaciones y se planifican acciones correctivas, lo que contribuye a mejorar tanto el producto final como los procesos de desarrollo y pruebas futuros[2].

Cada una de estas etapas es interdependiente y contribuye de forma decisiva a asegurar que el software no solo cumpla con los requisitos funcionales y de rendimiento, sino que también ofrezca una experiencia de usuario segura y confiable. Implementar un proceso de pruebas robusto y bien estructurado es, por tanto, una inversión clave en la calidad y éxito de cualquier proyecto de desarrollo de software.

## Tipos de pruebas de software

Existen diversos tipos de pruebas, cada uno diseñado para evaluar aspectos específicos del software y aplicable en distintas etapas del ciclo de vida del desarrollo. A continuación, se describen los principales tipos:

### Pruebas Unitarias:

Estas pruebas se centran en evaluar de forma individual cada componente o unidad del software, como funciones o métodos. El objetivo es asegurar que cada unidad funcione correctamente de forma aislada, permitiendo detectar errores en etapas muy tempranas del desarrollo[13, 14].

### Pruebas de Integración:

Una vez que se han verificado las unidades de forma individual, es necesario comprobar que interactúan correctamente entre sí. Las pruebas de integración se enfocan en la comunicación y el intercambio de datos entre módulos, detectando problemas en la interfaz y en la coordinación de componentes[13, 14].

### Pruebas Funcionales:

Estas pruebas evalúan si el software cumple con los requisitos especificados y realiza las funciones esperadas. Se realizan casos de prueba basados en escenarios de uso real, verificando la correcta ejecución de las operaciones, la manipulación de datos y la generación de resultados adecuados[13, 14].

### Pruebas de Sistema:

En esta etapa se examina el software en su totalidad, considerando la interacción de todos los módulos en un entorno que simula el ambiente de producción. Se evalúan tanto aspectos funcionales como no funcionales, garantizando que el sistema completo opere de manera coherente y estable[13, 14].

### Pruebas de Regresión:

Con cada nueva actualización o modificación del software, es fundamental asegurarse de que las funcionalidades existentes no se vean afectadas. Las pruebas de regresión se ejecutan de forma repetida para detectar la reaparición de errores o nuevos defectos introducidos por cambios recientes[13].

### Pruebas de Rendimiento:

Este tipo de pruebas se concentra en evaluar la capacidad del software para funcionar eficientemente bajo diversas cargas de trabajo. Se analizan aspectos como tiempos de respuesta, consumo de recursos y estabilidad cuando el sistema es sometido a altos volúmenes de transacciones o usuarios simultáneos[13].

### Pruebas de Seguridad:

Dado que el software puede estar expuesto a vulnerabilidades y amenazas, las pruebas de seguridad son cruciales. Estas evaluaciones identifican posibles brechas en la protección de datos, problemas en los mecanismos de autenticación y autorización, y otros riesgos que puedan comprometer la integridad y confidencialidad del sistema[13].

### Pruebas de Usabilidad:

Estas pruebas se enfocan en la experiencia del usuario, evaluando la interfaz, la navegabilidad y la facilidad de uso del software. Un sistema intuitivo y accesible es fundamental para lograr la satisfacción de los usuarios finales[13].

### Pruebas de Aceptación:

Finalmente, las pruebas de aceptación, ya sean internas o por parte del cliente (UAT, User Acceptance Testing), verifican que el software cumpla con las expectativas y requisitos del negocio. Es el último paso antes de la puesta en producción, donde se valida la funcionalidad desde la perspectiva del usuario final[13].

## Pruebas de software en sistemas de planificación de recursos empresariales

Las pruebas de software en los ERPs son fundamentales para garantizar la robustez y confiabilidad de sistemas que gestionan procesos críticos en las organizaciones. Debido a la complejidad de estos sistemas(que integran módulos de contabilidad, recursos humanos, inventarios, logística y más) la realización de pruebas exhaustivas se vuelve indispensable para asegurar que cada componente funcione de manera óptima, tanto de forma aislada como en conjunto

En este tipo de sistemas es esencial abordar las pruebas funcionales: se verifica que cada módulo cumpla con los requerimientos específicos y que los flujos de trabajo empresariales se ejecuten sin contratiempos. Estas pruebas se diseñan a partir de los procesos de negocio que el ERP debe soportar, asegurando que cada transacción y operación se procese correctamente.

Las pruebas de integración cobran especial relevancia en los ERPs, dado que los diferentes módulos deben comunicarse de forma fluida. Este tipo de pruebas ayuda a identificar posibles fallos en la interconexión de los componentes, garantizando la coherencia y sincronización de la información entre áreas tan diversas como ventas y finanzas o inventario y compras.

Las pruebas de rendimiento y escalabilidad son críticas, ya que los ERPs suelen manejar grandes volúmenes de datos y múltiples usuarios simultáneos. Evaluar la respuesta del sistema bajo carga permite identificar cuellos de botella y optimizar recursos para mantener la operatividad en entornos de alta demanda.

Las pruebas de seguridad son indispensables para proteger la integridad de los datos y asegurar que el sistema cumpla con normativas y estándares de protección de información. La detección temprana de vulnerabilidades evita posibles brechas que podrían comprometer la confidencialidad y disponibilidad de datos sensibles.

## Diferencias entre las pruebas manuales y las pruebas automatizadas

Las pruebas manuales y las pruebas automatizadas son dos enfoques fundamentales en el aseguramiento de la calidad del software, y cada uno ofrece ventajas específicas que, combinadas, permiten cubrir una amplia gama de escenarios y necesidades en el desarrollo.

Las pruebas manuales son realizadas por probadores sin ayuda de scripts o herramientas automatizadas. Este enfoque es especialmente valioso cuando se trata de evaluar la experiencia del usuario y la usabilidad, ya que permite a los evaluadores interactuar directamente con la aplicación, detectar problemas de navegación, inconsistencias en la interfaz o aspectos subjetivos que podrían pasar desapercibidos para una máquina. Además, en etapas tempranas o en procesos de exploración, las pruebas manuales facilitan la identificación de errores imprevistos, pues el evaluador puede adaptar su enfoque en función de lo que observa en tiempo real[15].

Las pruebas automatizadas consisten en la ejecución de scripts que verifican la funcionalidad del software de manera repetitiva y sistemática. Este método resulta indispensable en el SDLC, ya que permite:

Eficiencia y rapidez: Los tests automatizados se pueden ejecutar en cada integración o despliegue, reduciendo significativamente el tiempo de validación y permitiendo identificar errores de forma inmediata[16].

Consistencia: Al eliminar la variabilidad humana, se garantiza que cada prueba se realice de forma uniforme, lo que mejora la confiabilidad de los resultados[16].

Cobertura amplia: Permiten ejecutar un gran número de casos de prueba, incluyendo pruebas de regresión, de rendimiento y de seguridad, asegurando que las nuevas implementaciones no afecten funcionalidades ya existentes[16].

Integración en entornos CI/CD: La automatización se integra perfectamente en pipelines de integración y despliegue continuo, asegurando que cada cambio en el código sea validado de manera oportuna, lo que es esencial para mantener la calidad en proyectos de desarrollo ágil[16].

Si bien las pruebas automatizadas se han vuelto una necesidad dentro del ciclo de desarrollo moderno, especialmente en entornos de integración continua y despliegue rápido, las pruebas manuales siguen siendo cruciales para evaluar aspectos que requieren el juicio humano, como la experiencia del usuario y la interfaz gráfica. La combinación de ambos enfoques permite aprovechar lo mejor de cada uno: la precisión, rapidez y cobertura que ofrecen las pruebas automatizadas, junto con la capacidad de análisis detallado y contextual que brindan las pruebas manuales[17].

## Metodologías de pruebas de software

Históricamente, el modelo en cascada ha sido utilizado en proyectos donde se requiere una secuencia lineal de etapas. En este enfoque, las pruebas se realizan al final del proceso de desarrollo, lo que puede derivar en una detección tardía de errores y en altos costos de corrección[17]. Debido a esto, las metodologías de pruebas actuales se han diversificado para responder a las demandas de ciclos de desarrollo más cortos, la necesidad de integración continua y la creciente complejidad de las aplicaciones[2]. Aunque este método ha sido reemplazado en gran medida por enfoques más ágiles, sigue siendo útil en entornos con requisitos bien definidos y poca variabilidad.

La revolución de los métodos ágiles ha transformado la forma de abordar las pruebas de software. En este paradigma, las pruebas se integran desde las etapas iniciales del desarrollo y se ejecutan de manera iterativa junto con la codificación. Técnicas como el **Test Driven Development (TDD),** donde se escriben primero los casos de prueba para luego desarrollar el código que los satisfaga, y el **Behavior Driven Development (BDD),** que utiliza un lenguaje común entre desarrolladores y clientes para definir escenarios de comportamiento, permiten una mayor colaboración y una respuesta rápida ante cambios. Estas metodologías facilitan la integración continua, reducen el ciclo de retroalimentación y aseguran que el software cumpla con los requerimientos del usuario[18].

Dentro de los enfoques modernos la necesidad de integrar las pruebas automatizadas en ciclos de desarrollo acelerados han promovido la integración de estas pruebas en pipelines de CI/CD para detectar defectos en cada nueva integración al producto final[19].

## Herramientas para las pruebas automatizadas

Las herramientas para pruebas automatizadas son fundamentales para mejorar la eficiencia y la cobertura en el proceso de aseguramiento de calidad. Estas herramientas permiten ejecutar pruebas de forma repetitiva y sistemática, integrándose en entornos CI/CD. Las herramientas más conocidas y utilizadas en la industria son:

### Selenium WebDriver

Sin duda, una de las herramientas más reconocidas para la automatización de pruebas en aplicaciones web. Con soporte para múltiples lenguajes de programación y navegadores, Selenium permite simular la interacción del usuario y validar el comportamiento de la aplicación en diferentes entornos. Su flexibilidad y comunidad activa lo convierten en una opción robusta para proyectos de diversa envergadura[20].

### Cypress

Cypress está especialmente dedicada al ámbito de las aplicaciones modernas basadas en JavaScript. Se destaca por su fácil configuración, tiempos de ejecución rápidos y su capacidad para realizar pruebas de extremo a extremo, integrándose de manera fluida en pipelines de CI/CD. Su enfoque en el testing de front-end y su interfaz intuitiva hacen que sea una excelente opción para equipos que buscan resultados inmediatos y una curva de aprendizaje amigable[20].

### Playwright

Desarrollada por Microsoft, es otra herramienta emergente en el mundo de las pruebas automatizadas. Similar a Cypress en cuanto a su capacidad para realizar pruebas de aplicaciones web, Playwright ofrece la ventaja de soportar múltiples navegadores y dispositivos, permitiendo una mayor cobertura en entornos variados. Su diseño moderno y su integración con lenguajes como Python y JavaScript facilitan la creación de pruebas robustas y escalables[20].

### Appium

**Appium** se posiciona como una herramienta esencial para las pruebas de aplicaciones móviles. Permitiendo la automatización tanto de aplicaciones nativas como híbridas, Appium se integra con múltiples plataformas móviles y soporta diversos lenguajes de programación. Esto facilita la ejecución de pruebas de interfaz y funcionalidad en dispositivos reales o emulados, garantizando que las aplicaciones móviles cumplan con los estándares de calidad requeridos[20].

Además de estas herramientas especializadas en pruebas de interfaz, existen frameworks orientados a pruebas unitarias y de integración, como **JUnit** para aplicaciones en Java y **pytest** para Python. Estas herramientas son clave para validar el comportamiento de componentes individuales y asegurar que las integraciones entre ellos sean correctas, permitiendo una detección temprana de errores a nivel de código.

## Patrones de diseño para pruebas automatizadas

Los patrones de diseño para pruebas automatizadas representan soluciones estandarizadas a problemas comunes en la implementación de pruebas, proporcionando estructuras que mejoran la mantenibilidad, escalabilidad y reusabilidad del código de prueba[21]. Entre los patrones más utilizados se destacan:

### Page Object Model (POM)

Este patrón encapsula la estructura y comportamiento de cada página web en objetos separados, abstrayendo la interfaz de usuario de la lógica de prueba. El POM permite que los cambios en la UI afecten únicamente a las clases que representan las páginas, sin necesidad de modificar los casos de prueba, lo que reduce significativamente el esfuerzo de mantenimiento y mejora la legibilidad del código[22]. Cada página se representa como una clase que contiene los selectores de elementos y los métodos que interactúan con ellos, mientras que los casos de prueba simplemente utilizan estos objetos para simular las interacciones del usuario.

### Screenplay Pattern

Evolución del Page Object Model, este patrón se centra en las capacidades y tareas que un usuario puede realizar, más que en las páginas con las que interactúa. El Screenplay Pattern organiza el código en términos de actores, habilidades, tareas y preguntas, lo que mejora la expresividad y claridad de las pruebas, acercándolas al lenguaje de negocio[23]. Este enfoque resulta particularmente valioso en pruebas BDD, donde la alineación con el lenguaje de dominio es crucial.

### Data-Driven Testing

Este patrón separa la lógica de prueba de los datos utilizados, permitiendo ejecutar los mismos casos de prueba con diferentes conjuntos de entrada. Los datos pueden provenir de diversas fuentes como archivos CSV, bases de datos o APIs, lo que facilita la cobertura exhaustiva sin duplicar código[24]. El Data-Driven Testing es especialmente útil para validar funcionalidades que deben comportarse de manera consistente bajo diferentes condiciones de entrada.

### Factory Pattern

Aplicado a las pruebas automatizadas, este patrón simplifica la creación de objetos complejos necesarios para las pruebas, como entidades de dominio o datos de prueba. Las fábricas encapsulan la lógica de creación y proporcionan métodos para generar instancias con configuraciones predeterminadas o personalizadas, lo que mejora la legibilidad y mantenibilidad del código[24].

## Inteligencia Artificial en pruebas de software:

La integración de la Inteligencia Artificial (IA) en las pruebas de software representa uno de los avances más significativos en la evolución de la garantía de calidad de software en los últimos años. Esta convergencia tecnológica está transformando cada fase del ciclo de pruebas, desde la planificación hasta el análisis de resultados, optimizando procesos y mejorando la efectividad de la detección de errores.

### Generación automática de casos de prueba

Uno de los avances más prometedores es la aplicación de técnicas de aprendizaje automático para la generación de casos de prueba. Algoritmos basados en aprendizaje profundo pueden analizar el código fuente, requisitos funcionales y datos históricos de ejecución para crear conjuntos de pruebas exhaustivos con mínima intervención humana[24]. Estos sistemas son capaces de identificar áreas críticas del código y generar escenarios de prueba que maximizan la cobertura, incluyendo casos límite que podrían pasar desapercibidos en el diseño manual de pruebas.

### Pruebas autorreparables

Los frameworks de pruebas con capacidades de autoreparación representan un avance significativo en el mantenimiento de suites de pruebas automatizadas. Estos sistemas emplean algoritmos de aprendizaje por refuerzo para identificar cambios en los elementos de la interfaz y adaptar dinámicamente los selectores en tiempo de ejecución[23]. Cuando una prueba falla debido a cambios en la estructura del DOM o en identificadores de elementos, estas herramientas pueden generar selectores alternativos, reduciendo significativamente el tiempo dedicado al mantenimiento de scripts y aumentando la resiliencia de las pruebas automatizadas frente a cambios en la interfaz de usuario[25].

### Análisis predictivo de defectos

Los modelos predictivos basados en IA están revolucionando la forma en que se identifican potenciales áreas problemáticas en el software. Mediante el análisis de patrones en repositorios de código, historiales de cambios y datos de errores pasados, estos sistemas pueden predecir qué componentes tienen mayor probabilidad de contener defectos, permitiendo una asignación más eficiente de los recursos de prueba[25]. Esta capacidad predictiva facilita la implementación de estrategias de prueba focalizadas, concentrando los esfuerzos en áreas de alto riesgo y optimizando el tiempo dedicado a la verificación del software[26].

### Pruebas de comportamiento inteligentes

La aplicación de tecnologías de procesamiento de lenguaje natural (NLP) ha facilitado la creación de frameworks de pruebas que pueden interpretar requisitos expresados en lenguaje natural y traducirlos automáticamente en escenarios de prueba ejecutables[26]. Estas herramientas mejoran la colaboración entre equipos técnicos y no técnicos, permitiendo que analistas de negocio y usuarios finales participen directamente en la definición de criterios de aceptación, mientras que los sistemas de IA se encargan de generar y mantener los scripts de prueba correspondientes.

### Optimización de pruebas de regresión

Las técnicas de aprendizaje automático están siendo aplicadas con éxito para optimizar la selección de pruebas de regresión después de cada cambio en el código. Estos sistemas analizan la historia de ejecuciones previas, el impacto de los cambios y las dependencias del código para seleccionar el subconjunto mínimo de pruebas que proporcionará la máxima cobertura de los componentes afectados. Esta optimización reduce significativamente el tiempo de ejecución de las pruebas en pipelines de CI/CD sin comprometer la calidad, facilitando ciclos de retroalimentación más rápidos en entornos de desarrollo ágil.

### Análisis visual automatizado

En el ámbito de las pruebas de interfaz de usuario, los algoritmos de visión por computadora están transformando la validación visual de aplicaciones. Estas herramientas pueden detectar automáticamente inconsistencias visuales, problemas de renderizado y elementos solapados que afectan la experiencia del usuario. A diferencia de las pruebas tradicionales basadas en selectores DOM, los sistemas de análisis visual pueden identificar problemas que escapan a la validación programática, como errores de diseño, problemas de espaciado o elementos visuales incorrectos[26].

### Desafíos y perspectivas futuras

A pesar de los avances significativos, la integración de IA en las pruebas de software enfrenta desafíos importantes relacionados con la interpretabilidad de los resultados, la confiabilidad de las predicciones y la necesidad de datos de entrenamiento de alta calidad[26]. El desarrollo de modelos más transparentes y explicables representa una prioridad para facilitar la adopción generalizada de estas tecnologías en entornos críticos.

Las tendencias emergentes apuntan hacia sistemas híbridos que combinan la experiencia humana con las capacidades analíticas de la IA, creando entornos de prueba colaborativos donde los profesionales de QA son asistidos por agentes inteligentes que automatizan tareas repetitivas y proporcionan insights basados en análisis avanzados[26]. Esta sinergia entre humanos e IA promete elevar las pruebas de software a un nuevo nivel de eficiencia y efectividad, fundamentales para satisfacer las demandas de calidad en el acelerado panorama actual de desarrollo de software.

## Conclusiones del capítulo

El análisis de los campos investigados nos proporciona la base de conocimiento necesaria para desarrollar el plan de pruebas automatizadas del módulo Carga y Descarga. En el alcance de este documento, se profundizará únicamente en el submódulo Choferes, debido a su alto nivel de completitud y a su papel fundamental en el funcionamiento del sistema.

Se aplicará una metodología BDD para complementar el desarrollo del módulo, asegurando la correcta validación de sus funcionalidades. Para implementar las pruebas, se utilizarán los frameworks Playwright y Pytest: el primero se empleará para la automatización de pruebas en la interfaz de usuario, y el segundo, para realizar pruebas unitarias en funciones específicas. Ambos se integrarán con Behave, un framework diseñado específicamente para metodologías BDD, que facilita la comprensión de las pruebas incluso a personas externas al equipo de implementación.

# Capítulo 2: Propuesta de solución

## Introducción al capítulo

Este capítulo presenta la estrategia integral adoptada para el desarrollo del sistema de pruebas automatizadas del módulo Carga y Descarga del sistema de planificación de recursos de Habana Club International. La propuesta de solución que se expone a continuación surge como respuesta a las necesidades identificadas en el contexto empresarial, donde la ausencia de metodologías sistemáticas de pruebas automatizadas ha generado ineficiencias operativas y riesgos potenciales para la calidad del producto final.

El enfoque adoptado integra aspectos metodológicos y técnicos para crear un marco de trabajo que garantice la calidad del software a través de pruebas exhaustivas, reproducibles y mantenibles. La solución propuesta no se limita a la implementación de herramientas tecnológicas, sino que contempla un cambio en la filosofía de desarrollo, donde la calidad se construye desde las primeras etapas del ciclo de vida del software.

En las siguientes secciones se detallan los componentes fundamentales de la solución, partiendo del análisis del módulo Carga-Descarga, la metodología para el diseño de casos de prueba, la arquitectura de la solución de automatización, hasta las estrategias de gestión de datos e integración con el proceso de desarrollo. Cada elemento ha sido cuidadosamente considerado para asegurar una implementación exitosa que agregue valor significativo al proceso de desarrollo de software en AVANGENIO.

## Análisis del módulo "Carga-Descarga"

El módulo "Carga-Descarga" constituye un componente crítico dentro del sistema de planificación de recursos de Habana Club International, encargado de gestionar los procesos logísticos relacionados con la entrada y salida de camiones, el control de peso y la coordinación de operaciones de movimiento de materiales.

### Caracterización funcional del módulo

El análisis funcional del módulo revela una estructura compuesta por diversos submódulos interrelacionados, entre los que destacan:

1. Gestión de tanques: Encargado del seguimiento y control de las capacidades de almacenamiento y niveles de ocupación.
2. Registro de vehículos: Administra la información relacionada con las chapas (matrículas) de los vehículos que interactúan con el sistema logístico.
3. Gestión de choferes: Mantiene un registro actualizado de los conductores autorizados para las operaciones de carga y descarga.
4. Laboratorio de calidad: Integra los procedimientos de control de calidad de los productos durante los procesos de carga y descarga.
5. Operaciones de carga: Coordina todas las actividades relacionadas con la salida de productos desde las instalaciones.
6. Operaciones de descarga: Gestiona los procesos de recepción de materiales y productos, incluyendo verificación, pesaje y almacenamiento.

Los sistemas de gestión logística requieren un enfoque particular de pruebas debido a la naturaleza crítica de sus operaciones y la necesidad de integración con sistemas externos. La complejidad inherente a estos sistemas demanda estrategias de prueba que contemplen tanto los aspectos funcionales como los no funcionales, especialmente aquellos relacionados con la consistencia de datos y la trazabilidad de operaciones. [41]

### Flujos principales de usuario

Los flujos de usuario en el módulo "Carga-Descarga" están estrechamente vinculados a los roles y autorizaciones asignados a cada usuario dentro del sistema. El acceso a las diferentes secciones depende de permisos específicos, lo que genera múltiples caminos de interacción basados en responsabilidades operativas.

Los principales flujos identificados incluyen:

1. **Flujo de administración**: El rol de Administrador tiene acceso completo a todas las secciones del módulo, incluyendo Chofer, Vehículo, Laboratorio de Calidad, Tanque, Carga y Descarga. Este flujo comprende la supervisión general y la capacidad de intervenir en cualquier punto del proceso.
2. **Flujo de gestión técnica**: El Técnico del Puesto de Dirección puede acceder a la mayoría de las secciones, con la notable excepción de la pestaña "Carga a Fabricación". Este flujo se centra en el monitoreo y control de las operaciones logísticas desde una perspectiva de dirección operativa.
3. **Flujo de fabricación**: Este flujo involucra a múltiples roles del área de fabricación:
   * **Jefe de Fabricación**: Puede acceder a la mayoría de secciones excepto la pestaña "Puesto de Dirección".
   * **Tecnólogo de Fabricación**: Enfocado en aspectos técnicos del proceso productivo.
   * **Fabricante**: Centrado en operaciones directas de producción.
   * **Jefe de Turno de Fabricación**: Supervisa las operaciones de carga y descarga durante su turno específico.
   * **Especialista de Economía de Fabricación**: Accede a información relacionada con aspectos económicos de los procesos.
4. **Flujo de control de calidad**: Este flujo incluye roles específicos para la verificación de calidad:
   * **Jefe de Laboratorio**: Tiene acceso exclusivo a la sección "Laboratorio de Calidad" junto con el Administrador.
   * **Especialista de Laboratorio de Fabricación**: Encargado de realizar pruebas y verificaciones de calidad específicas para los procesos de fabricación.

Cada uno de estos flujos incorpora secuencias de acciones específicas que reflejan las responsabilidades operativas de cada rol. La complejidad aumenta debido a las interdependencias secuenciales, donde la acción de un usuario (por ejemplo, el Jefe de Laboratorio verificando la calidad de un material) puede generar un punto de espera que requiere la intervención de otro usuario (como el Técnico del Puesto de Dirección autorizando la continuación del proceso) para avanzar en el flujo logístico.

El sistema de autenticación con contraseñas estandarizadas (TestErp2024\*+) para todos los usuarios facilita las pruebas en entornos controlados, mientras que la estructura de correos electrónicos con el formato test.erp.hci+[rol]@avangenio.com permite una clara identificación de los diferentes roles durante las pruebas.

La estructura de permisos segmentados asegura que los usuarios solo puedan interactuar con las secciones relevantes para sus responsabilidades, lo que contribuye a la integridad del proceso logístico, pero añade complejidad a la hora de diseñar pruebas automatizadas comprehensivas que cubran todos los posibles caminos de usuario.

## Complejidades técnicas identificadas

Durante el análisis se identificaron varios desafíos técnicos que impactan directamente en la estrategia de pruebas:

1. **Ausencia de sistemas externos de pesaje**: Actualmente no existe un sistema externo real para el pesaje, por lo que se utiliza un mecanismo de verificación manual (check "pesa rota") que debe ser contemplado en los escenarios de prueba.
2. **Dependencias entre roles en procesos secuenciales**: Los flujos de trabajo incluyen puntos de pausa donde se requiere la intervención de roles específicos para continuar con el proceso, lo que genera complejidades para la prueba automatizada.
3. **Validaciones complejas con dependencias de calendario**: Existen reglas de negocio que involucran múltiples variables para la toma de decisiones logísticas, incluyendo validaciones relacionadas con lotes que incorporan componentes de calendario juliano.

Estos factores de complejidad deben ser considerados explícitamente en el diseño de pruebas automatizadas para evitar falsos positivos o negativos durante la ejecución. [42]

## Metodología para el diseño de casos de prueba

La metodología adoptada para el diseño de casos de prueba se fundamenta en prácticas ágiles y principios de Behavior-Driven Development (BDD), buscando un enfoque sistemático que garantice la cobertura funcional completa del módulo.

### Proceso de identificación y documentación

Para el diseño de casos de prueba se implementó un proceso estructurado en tres fases principales:

1. Fase de análisis preliminar: Estudio de las historias de usuario y especificaciones funcionales para identificar escenarios de prueba relevantes. Esta etapa incluyó la creación de una matriz de historias de usuario en formato Excel, donde se documentaron los aspectos clave a verificar por cada funcionalidad.
2. Fase de diseño de casos: Transformación de los escenarios identificados en casos de prueba formales, siguiendo inicialmente el formato BDD y posteriormente evolucionando hacia la sintaxis Gherkin para aprovechar sus capacidades avanzadas.
3. Fase de validación: Revisión y aprobación de los casos de prueba diseñados, garantizando su alineación con los requerimientos funcionales y la completitud de los escenarios.

### Implementación del enfoque BDD y migración a Gherkin

Inicialmente, los casos de prueba fueron documentados siguiendo la estructura básica de BDD (Given-When-Then). Este enfoque permitió una comunicación clara de los escenarios a probar; sin embargo, a medida que avanzó el proyecto, se identificaron limitaciones en términos de reusabilidad y organización de pasos comunes.

Se realizó una migración progresiva hacia la sintaxis Gherkin, lo que permitió incorporar características avanzadas como:

1. Background: Para definir pasos comunes que se ejecutan antes de cada escenario.
2. Scenario Outline: Para parametrizar pruebas con conjuntos diferentes de datos.

Esta evolución metodológica optimizó significativamente la estructura de los casos de prueba, reduciendo la duplicación y mejorando la mantenibilidad del conjunto de pruebas. [43]

## Organización y gestión con Squash TM

La gestión de los casos de prueba se realizó a través de Squash TM, una herramienta especializada que permitió:

1. Organización jerárquica: Estructuración de casos de prueba según módulos y funcionalidades.
2. Trazabilidad: Vinculación directa entre requisitos funcionales y casos de prueba correspondientes.
3. Control de versiones: Gestión de cambios en los casos de prueba a lo largo del tiempo.
4. Exportación automatizada: Generación de archivos .feature en formato Gherkin para su posterior implementación.

Para mantener la sincronización entre los casos diseñados y su implementación, se estableció un sistema de referencia cruzada utilizando una matriz en Excel que incluía:

* Identificador de historia de usuario
* Enlace a la especificación en el sistema de gestión de requisitos
* Identificador de caso de prueba en Squash TM
* Estado de implementación
* Responsable de la implementación

Esta estrategia proporciona visibilidad y control sobre el proceso de diseño e implementación de casos. [44]

## Arquitectura de la solución de automatización

La arquitectura de la solución de automatización fue diseñada siguiendo principios de modularidad, reutilización y mantenibilidad, adoptando patrones de diseño establecidos en la industria para pruebas automatizadas.

### Framework de automatización seleccionado

Tras un análisis comparativo de diferentes herramientas, se seleccionó Behave como framework principal para la implementación de pruebas automatizadas. Esta elección se fundamentó en:

1. Compatibilidad con BDD/Gherkin: Soporte nativo para la sintaxis utilizada en la especificación de casos.
2. Integración con Python: Aprovechamiento del ecosistema Python para la implementación de los scripts de prueba.
3. Extensibilidad: Capacidad para integrar librerías adicionales según necesidades específicas.
4. Compatibilidad con Playwright: Facilidad de integración con la herramienta seleccionada para la interacción con interfaces web.

Behave representa una solución robusta para la implementación de pruebas BDD, proporcionando un equilibrio adecuado entre simplicidad de uso y capacidades avanzadas. [45]

Entiendo, voy a corregir la estructura para que solo los steps y los archivos .feature estén dentro de la carpeta tests, sin incluir los demás archivos que mencionaste:

### Implementación del patrón Page Object Model (POM)

La arquitectura de la solución se fundamenta en el patrón Page Object Model, una práctica recomendada en la industria que promueve la separación entre la lógica de pruebas y la interacción con la interfaz de usuario. Se implementó una estructura jerárquica de clases que representan las diferentes páginas y componentes del sistema [46]:

HCI-MANUFACTURING-QA-AUTOMATION/

├── .venv

├── config

├── docs

├── pages/

│ └── .gitkeep

├── tests/

│ ├── steps/

│ │ ├── common\_steps.py

│ │ ├── carga\_steps.py

│ │ ├── descarga\_steps.py

│ │ └── ...

│ ├── carga.feature

│ ├── descarga.feature

│ ├── tanques.feature

│ └── ...

├── \_\_init\_\_.py

├── conftest.py

├── \_init\_.py

├── .env.example

├── .gitignore

├── .python-version

├── package-lock.json

├── package.json

├── playwright.config.ts

├── README.md

├── requirements.txt

└── utils/

├── context\_utils.py

├── fixtures.py

└── ...

Este enfoque aporta múltiples beneficios:

1. Abstracción de la UI: Encapsula los detalles de la interfaz de usuario, facilitando el mantenimiento cuando ocurren cambios visuales.
2. Reutilización de código: Las acciones comunes se implementan una única vez en las clases correspondientes.
3. Legibilidad mejorada: Los scripts de prueba expresan intenciones de negocio en lugar de detalles técnicos.
4. Mantenibilidad: Los cambios en la interfaz afectan únicamente a las clases de páginas correspondientes.

### Gestión de contexto y estado

Un aspecto crucial en la arquitectura implementada es la gestión eficiente del contexto y estado durante la ejecución de pruebas. Behave proporciona un mecanismo nativo (el objeto "context") que fue aprovechado para:

1. Compartir información entre pasos: Transmisión de datos generados en un paso para su verificación en pasos posteriores.
2. Mantener referencias a objetos de página: Acceso consistente a las instancias de Page Objects a lo largo de la ejecución.
3. Almacenar configuraciones dinámicas: Adaptación del comportamiento de las pruebas según el entorno de ejecución.

Estas capacidades son particularmente relevantes en un sistema como el módulo Carga-Descarga, donde existen dependencias secuenciales entre operaciones y donde el estado anterior influye en el comportamiento esperado de pasos subsecuentes.

### Mecanismos de fixture para precondiciones

Para abordar las necesidades de preparación del entorno de pruebas, se implementaron fixtures utilizando las capacidades de hooks de Behave. Estos mecanismos permiten:

1. Autenticación automática: Establecimiento de sesiones autenticadas antes de cada escenario.
2. Preparación de datos: Creación de precondiciones necesarias para la ejecución de pruebas.
3. Limpieza posterior: Restauración del estado inicial del sistema tras la ejecución.

Este enfoque de "preparación automática" reduce significativamente la duplicación de código y mejora la confiabilidad de las pruebas al garantizar un entorno consistente para cada ejecución. [47, 48]

## Estrategia de parametrización y gestión de datos

La gestión eficiente de datos de prueba constituye un factor crítico para el éxito de cualquier iniciativa de automatización. La estrategia adoptada se fundamenta en principios de separación de datos y lógica, configurabilidad y reutilización.

### Fuentes de datos para las pruebas

La estrategia de gestión de datos para las pruebas automatizadas del módulo "Carga-Descarga" se fundamenta en un enfoque generativo y paramétrico, donde los propios especialistas en QA diseñan y proporcionan los conjuntos de datos necesarios para cubrir exhaustivamente los escenarios de prueba.

Las principales fuentes de datos utilizadas incluyen:

1. Datos parametrizados mediante Gherkin: La implementación de la sección "Examples" en los archivos .feature permite definir conjuntos de valores para variables específicas (como <nombre>, <matrícula>, etc.) que serán utilizados durante la ejecución de los escenarios. Esta técnica facilita la ejecución de un mismo escenario con múltiples combinaciones de datos sin duplicar código.
2. Datos de configuración estáticos: Información relativamente estable como credenciales de acceso, URLs de entorno, tiempos de espera y otros parámetros de configuración se almacenan en archivos de configuración separados, facilitando su modificación sin alterar el código de las pruebas.
3. Datos dinámicos capturados durante la ejecución: En escenarios donde ciertos valores son generados por el sistema (como identificadores únicos, timestamps o resultados de cálculos), estos son capturados durante la ejecución de un paso y almacenados en el contexto para su uso en pasos posteriores.
4. Datos generados programáticamente (planificación futura**)**: El Project Manager junto con el Jefe de los QA automatizados han propuesto implementar funciones generadoras que producirían datos válidos o inválidos según los requisitos del caso de prueba, especialmente para escenarios que requieren grandes volúmenes de datos o valores específicos como fechas en formato juliano para validaciones de lotes. Sin embargo, esta iniciativa aún se encuentra en fase de evaluación y no se ha confirmado su implementación definitiva.

El diseño de estos conjuntos de datos se basa en un análisis exhaustivo de las historias de usuario, buscando cubrir tanto escenarios típicos como casos límite y situaciones excepcionales que podrían ocurrir durante la operación real del sistema.

### Estrategia de aislamiento y repetibilidad

Para garantizar la confiabilidad y repetibilidad de las pruebas automatizadas, se ha implementado una estrategia de aislamiento que minimiza las dependencias entre diferentes ejecuciones de prueba y asegura un entorno consistente para cada ciclo de evaluación.

Los componentes clave de esta estrategia incluyen:

1. **Hooks de preparación y limpieza**: Mediante la utilización del archivo environment.py de Behave, se han implementado mecanismos que ejecutan acciones específicas en momentos críticos del ciclo de vida de las pruebas:
   * **before\_all**: Configuración global única al inicio de toda la ejecución
   * **before\_feature**: Preparación específica para cada archivo .feature
   * **before\_scenario**: Configuración del entorno para cada escenario individual
   * **after\_scenario**: Limpieza y restauración del estado después de cada escenario
   * **after\_feature**: Acciones finales tras completar todos los escenarios de un feature
   * **after\_all**: Limpieza general al finalizar toda la ejecución
2. **Datos independientes por escenario**: Cada escenario de prueba opera con datos específicos y aislados, evitando interdependencias que podrían comprometer la repetibilidad de las pruebas. Esto se logra mediante la parametrización explícita y la regeneración de datos para cada ejecución.
3. **Restauración de estado:** Para escenarios que modifican el estado del sistema (como la creación de registros o la actualización de información existente), se implementan mecanismos automáticos que devuelven el sistema a un estado conocido tras la ejecución, ya sea mediante operaciones inversas o reinicialización de datos.
4. **Manejo de precondiciones:** Los requisitos previos para cada escenario se gestionan mediante pasos de configuración explícitos o a través de la sección "Background" en los archivos .feature, asegurando que todos los componentes necesarios estén disponibles antes de la ejecución de los pasos de prueba.

Esta estrategia de aislamiento no solo mejora la confiabilidad de los resultados de las pruebas, sino que también facilita la depuración al reducir las posibles causas de fallos intermitentes relacionados con estados inconsistentes del entorno de pruebas.

## Integración con el proceso de desarrollo

La integración efectiva de las pruebas automatizadas en el flujo de desarrollo representa un factor determinante para maximizar su valor y garantizar su adopción por parte del equipo.

### Workflow de implementación

Se estableció un workflow estructurado para la integración de nuevos casos de prueba:

1. Diseño en Squash TM: Creación y documentación inicial del caso de prueba siguiendo la sintaxis BDD/Gherkin.
2. Revisión y aprobación: Validación del caso de prueba por parte de analistas funcionales para garantizar su alineación con los requisitos.
3. Exportación a archivo .feature: Generación del archivo correspondiente en la estructura del proyecto de automatización.
4. Verificación de disponibilidad de Page Objects: Análisis de la necesidad de crear nuevas clases de página o reutilizar existentes.
5. Implementación de steps: Desarrollo de los pasos específicos requeridos por el escenario.
6. Verificación inicial: Ejecución local para confirmar el funcionamiento correcto.
7. Integración en suite de pruebas: Incorporación al conjunto completo de pruebas automatizadas.

Este proceso secuencial garantiza la trazabilidad completa desde el requisito funcional hasta la implementación automatizada, facilitando el mantenimiento y evolución del sistema de pruebas.

### Integración con CI/CD

Aunque aún en fase inicial, se ha diseñado un plan para la integración del sistema de pruebas con la infraestructura de CI/CD, contemplando:

1. Ejecución automática: Configuración para la ejecución de pruebas tras cada integración de código.
2. Reporting automatizado: Generación de informes de resultados en formato estandarizado para facilitar su interpretación.
3. Gestión de fallos: Estrategia para la notificación y gestión de casos fallidos, incluyendo mecanismos de reintentos para identificar fallos intermitentes.
4. Análisis de tendencias: Almacenamiento histórico de resultados para identificar patrones de comportamiento y áreas problemáticas.

Este enfoque, alineado con las prácticas recomendadas por Humble y Farley [49], busca maximizar el valor de las pruebas automatizadas incorporándolas como elemento fundamental del pipeline de entrega continua.

## Implementación de mantenimiento asistido por IA

Como elemento innovador dentro de la propuesta, se ha incorporado un enfoque de mantenimiento asistido por inteligencia artificial para los scripts de prueba automatizados, lo que representa un avance significativo en la sostenibilidad del proyecto a largo plazo.

### Asistencia de Claude para actualización de código

Una de las principales desventajas de las pruebas automatizadas tradicionales es su fragilidad ante cambios en la estructura de la aplicación. Para mitigar este problema, hemos implementado un flujo de trabajo de mantenimiento asistido por IA que:

1. Proporciona actualizaciones contextuales: Cuando se producen cambios en el proyecto posteriores a la implementación inicial, proporcionamos a Claude el código existente junto con información sobre los cambios necesarios.
2. Genera código actualizado con coherencia estructural: Claude analiza la estructura actual del proyecto, comprende las modificaciones requeridas y propone implementaciones que mantienen la coherencia con los patrones y estándares establecidos.
3. Preserva la lógica de negocio: Al actualizar selectores o flujos de interacción, Claude mantiene intacta la lógica fundamental de las pruebas, enfocándose únicamente en los aspectos técnicos que requieren actualización.

Según experiencias documentadas por Alshahwan et al. [50], este enfoque puede reducir hasta en un 60% el tiempo dedicado al mantenimiento de pruebas automatizadas.

### Beneficios del mantenimiento asistido por IA

La implementación de este enfoque ofrece ventajas significativas:

1. Respuesta ágil ante cambios: Capacidad para adaptar rápidamente los scripts de prueba cuando la interfaz o comportamiento de la aplicación evoluciona.
2. Consistencia en la actualización: Garantía de que las modificaciones siguen patrones coherentes en todo el código base, evitando soluciones ad-hoc que puedan introducir inconsistencias.
3. Transferencia de conocimiento: Claude comprende la estructura y patrones del proyecto, lo que facilita la incorporación de nuevos miembros al equipo, quienes pueden interactuar con la IA para entender y modificar el código existente.

Este enfoque representa una dirección prometedora para organizaciones que buscan mantener un equilibrio entre la cobertura de pruebas automatizadas y los recursos necesarios para su mantenimiento, permitiendo que el equipo se concentre en actividades de mayor valor añadido.

## Conclusiones del capítulo

La propuesta de solución presentada en este capítulo constituye un enfoque integral para la automatización de pruebas del módulo Carga-Descarga, estableciendo una base sólida tanto metodológica como técnica que responde directamente a las necesidades identificadas en el contexto empresarial de Habana Club International.

Los elementos fundamentales que conforman esta propuesta incluyen:

1. **Enfoque BDD/Gherkin:** La adopción de un paradigma centrado en el comportamiento facilita la comprensión de las pruebas por todos los interesados, mejorando significativamente la comunicación entre equipos técnicos y de negocio, y permitiendo que las especificaciones funcionales se conviertan directamente en casos de prueba ejecutables.
2. **Arquitectura POM (Page Object Model):** La implementación de una estructura modular basada en Page Objects garantiza la mantenibilidad y escalabilidad de la solución, encapsulando los detalles de implementación de la interfaz y permitiendo que los cambios en la UI afecten mínimamente a los scripts de prueba.
3. **Gestión estructurada del proceso:** El establecimiento de procesos formales para el diseño, implementación y mantenimiento de casos de prueba mediante herramientas como Squash TM proporciona trazabilidad completa desde los requisitos hasta la implementación automatizada.
4. **Estrategia robusta de datos:** El enfoque de parametrización y aislamiento de datos garantiza la repetibilidad y confiabilidad de las pruebas, factor crítico en sistemas con flujos complejos como el módulo Carga-Descarga.
5. **Integración con el ciclo de desarrollo:** La incorporación planificada de las pruebas automatizadas en el flujo de CI/CD permitirá maximizar su valor y convertirlas en un componente esencial del proceso de aseguramiento de calidad.
6. **Innovación mediante IA:** La exploración de técnicas de mantenimiento asistido por inteligencia artificial representa un avance significativo hacia la sostenibilidad a largo plazo del esfuerzo de automatización.

La validez técnica de esta propuesta se fundamenta en su alineación con las mejores prácticas de la industria, como evidencian las referencias a trabajos de autores reconocidos en el campo del testing automatizado. Adicionalmente, el diseño responde específicamente a las complejidades identificadas en el análisis del módulo Carga-Descarga, incluyendo la estructura de roles, los flujos secuenciales y las dependencias entre procesos.

La implementación práctica de esta solución, que se abordará en detalle en el siguiente capítulo, constituirá una validación empírica de la efectividad del enfoque propuesto y proporcionará datos cuantitativos sobre sus beneficios en términos de eficiencia, cobertura y capacidad de detección de defectos, estableciendo así un precedente valioso para futuras iniciativas de automatización en la organización.

# Referencias Bibliográficas

1. Pressman, R.S., *Software engineering: a practitioner's approach*. 2005: Palgrave macmillan.

2. Myers, G.J., C. Sandler, and T. Badgett, *The art of software testing*. 2011: John Wiley & Sons.

3. Krasner, H., *The cost of poor software quality in the US: A 2020 report.* Proc. Consortium Inf. Softw. QualityTM (CISQTM), 2021. **2**: p. 3.

4. Pargaonkar, S., *A Study on the Benefits and Limitations of Software Testing Principles and Techniques: Software Quality Engineering.* International Journal of Scientific and Research Publications (IJSRP) 13(08) (ISSN: 2250-3153), 2023.

5. Carter, D.O., *Advancing Software Quality: A Comprehensive Exploration of Code Quality Metrics, Static Analysis Tools, and Best Practices.* Journal of Science & Technology, 2024. **5**(1): p. 69-81.

6. Pargaonkar, S., *A comprehensive research analysis of software development life cycle (SDLC) agile & waterfall model advantages, disadvantages, and application suitability in software quality engineering.* International Journal of Scientific and Research Publications (IJSRP), 2023. **13**(08): p. 345-358.

7. Crispin, L. and J. Gregory, *Agile Testing: A Practical Guide for Testers and Agile Teams*. 2009: Addison-Wesley.

8. Jorgensen, P.C., *Software Testing: A Craftsman’s Approach, Fourth Edition*. 2013: Taylor & Francis.

9. Wang, J., et al., *Software testing with large language models: Survey, landscape, and vision.* IEEE Transactions on Software Engineering, 2024.

10. Pargaonkar, S., *Synergizing Requirements Engineering and Quality Assurance: A Comprehensive Exploration in Software Quality Engineering* International Journal of Science and Research (IJSR), 2022. **12**(8).

11. Black, R. and J.L. Mitchell, *Advanced Software Testing - Vol. 3: Guide to the ISTQB Advanced Certification as an Advanced Technical Test Analyst*. 2011: Rocky Nook.

12. Stapp, L., A. Roman, and M. Pilaeten, *ISTQB® Certified Tester Foundation Level: A Self-Study Guide Syllabus v4.0*. 2023: Springer Nature Switzerland.

13. Spillner, A., T. Linz, and H. Schaefer, *Software Testing Foundations: A Study Guide for the Certified Tester Exam*. 2014: Rocky Nook.

14. Jamil, M.A., et al. *Software testing techniques: A literature review*. in *2016 6th international conference on information and communication technology for the Muslim world (ICT4M)*. 2016. IEEE.

15. Thant, K. and H.H.K. Tin, *THE IMPACT OF MANUAL AND AUTOMATIC TESTING ON SOFTWARE TESTING EFFICIENCY AND EFFECTIVENESS.* 2023. **3**: p. 88-93.

16. Kumar, D. and K.K. Mishra, *The Impacts of Test Automation on Software's Cost, Quality and Time to Market.* Procedia Computer Science, 2016. **79**: p. 8-15.

17. Homès, B., *Fundamentals of software testing*. 2024: John Wiley & Sons.

18. Beck, K., *Test driven development: By example*. 2022: Addison-Wesley Professional.

19. Spinellis, D., *State-of-the-Art Software Testing.* IEEE Software, 2017. **34**(5): p. 4-6.

20. Donvir, A. *A Comparative Analysis of JavaScript Unit and End-to-End Testing Frameworks: Enhancing Quality Assurance in Modern Web Applications*. in *2024 IEEE 16th International Conference on Computational Intelligence and Communication Networks (CICN)*. 2024.

21. Khankhoje, R., *An In-Depth Review of Test Automation Frameworks: Types and Trade-offs.* International Journal of Advanced Research in Science, Communication and Technology (IJARSCT), 2023. **3**(1): p. 55-64.

22. Schäfer, M., et al., *An empirical evaluation of using large language models for automated unit test generation.* IEEE Transactions on Software Engineering, 2023. **50**(1): p. 85-105.

23. Brahmbhatt, K.H., *Comparative analysis of selecting a test automation framework for an e-commerce website.* Tallinn University of Technology, 2023.

24. Pícha, P., et al. *Process anti-pattern detection–a case study*. in *Proceedings of the 27th european conference on pattern languages of programs*. 2022.

25. Bai, X., et al. *WSDL-based automatic test case generation for web services testing*. in *IEEE International Workshop on Service-Oriented System Engineering (SOSE'05)*. 2005. IEEE.

26. Mustafa, A., et al., *Automated test case generation from requirements: A systematic literature review.* Computers, Materials and Continua, 2021. **67**(2): p. 1819-1833.