**1.7 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[25].

**1.8 Integración de pruebas automatizadas en CI/CD**

La integración de pruebas automatizadas en pipelines de CI/CD representa un pilar fundamental en el desarrollo de software moderno, permitiendo validar continuamente cada cambio introducido en el código[26]. Este enfoque asegura la detección temprana de errores, reduciendo significativamente el costo de corrección y acelerando el ciclo de retroalimentación para los equipos de desarrollo.

Los pipelines de CI/CD típicamente estructuran las pruebas en diferentes niveles o etapas:

1. **Pruebas unitarias**: Se ejecutan en cada commit, proporcionando retroalimentación inmediata sobre la integridad de los componentes individuales.
2. **Pruebas de integración**: Verifican la correcta interacción entre módulos, ejecutándose generalmente después de las pruebas unitarias.
3. **Pruebas de API**: Validan los contratos y comportamientos de las interfaces de programación.
4. **Pruebas end-to-end**: Evalúan el sistema desde la perspectiva del usuario final, simulando interacciones reales.
5. **Pruebas de rendimiento y seguridad**: Se ejecutan con menor frecuencia, generalmente antes de despliegues a ambientes de producción[27].

La implementación efectiva de esta integración requiere considerar aspectos como:

* **Paralelización**: Distribuir la ejecución de pruebas para reducir tiempos de espera.
* **Estrategias de selección**: Identificar y ejecutar solo las pruebas relevantes para los cambios introducidos.
* **Gestión de entornos**: Provisionar y configurar automáticamente los ambientes necesarios mediante infraestructura como código.
* **Reportes e insights**: Generar informes detallados que faciliten el análisis de resultados y la toma de decisiones[28].

**1.9 Herramientas específicas para sistemas ERP**

**Squash**

Squash es una plataforma de código abierto diseñada específicamente para la gestión del ciclo de vida de las pruebas (TLM), que proporciona un conjunto integrado de herramientas para la planificación, diseño, ejecución y seguimiento de pruebas[29]. Su enfoque modular permite adaptarse a diferentes contextos y metodologías, siendo particularmente valioso en entornos empresariales complejos como los sistemas ERP.

Entre sus características destacadas se encuentran:

* Gestión centralizada de requisitos y casos de prueba
* Trazabilidad bidireccional entre requisitos y pruebas
* Soporte para pruebas manuales y automatizadas
* Gestión avanzada de campañas de pruebas
* Integración con herramientas de CI/CD y gestión de proyectos[29]

**Behave**

Behave es un framework de pruebas BDD para Python que permite escribir pruebas en lenguaje natural siguiendo la sintaxis Gherkin (Given-When-Then), lo que facilita la colaboración entre equipos técnicos y de negocio[30]. Esta herramienta traduce escenarios escritos en lenguaje natural a código ejecutable, permitiendo validar que el sistema cumpla con los comportamientos esperados desde la perspectiva del usuario.

Sus principales ventajas incluyen:

* Documentación viva que mantiene alineados los requisitos y las pruebas
* Estructura clara para definir precondiciones, acciones y resultados
* Soporte para parametrización y reutilización de pasos
* Integración fluida con frameworks de automatización web como Selenium o Playwright
* Generación automática de reportes detallados[30]

**1.10 Conclusiones del marco teórico**

Tras el análisis de las diferentes metodologías, herramientas y enfoques para la automatización de pruebas en sistemas ERP, se ha determinado que la combinación de Playwright, Squash y Behave constituye la solución óptima para abordar los desafíos específicos del módulo "Carga y Descarga" del sistema de planificación de recursos de Habana Club International.

La selección de **Playwright** como herramienta principal de automatización obedece a múltiples factores críticos para el proyecto. Su capacidad para emular múltiples navegadores con un solo código base permitirá maximizar la cobertura sin duplicar esfuerzos, aspecto fundamental considerando las limitaciones de recursos del equipo. Además, su arquitectura moderna y sin dependencia de WebDriver proporciona mayor estabilidad y rendimiento en comparación con alternativas como Selenium, reduciendo los falsos positivos que frecuentemente afectan la confiabilidad de las pruebas automatizadas[31]. Su robusta API para intercepción de red facilitará la validación de integraciones complejas presentes en el módulo, mientras que su soporte para pruebas visuales permitirá verificar la correcta representación de elementos críticos como informes y dashboards logísticos.

La incorporación de **Squash** como plataforma de gestión del ciclo de vida de pruebas responde a la necesidad de establecer procesos estructurados y trazables en un contexto donde previamente las pruebas se realizaban de manera ad-hoc. Su capacidad para vincular requisitos, casos de prueba y resultados de ejecución proporcionará la visibilidad necesaria sobre la cobertura y calidad del sistema, facilitando la toma de decisiones basada en datos[29]. Adicionalmente, su naturaleza de código abierto se alinea con las restricciones presupuestarias del proyecto, permitiendo implementar una solución robusta sin incurrir en costos de licenciamiento significativos.

Finalmente, la adopción de **Behave** como framework BDD complementa perfectamente esta arquitectura de pruebas, estableciendo un puente comunicacional entre los expertos del dominio logístico y el equipo técnico. La especificación de pruebas en formato Gherkin facilitará la validación temprana de escenarios críticos de negocio antes de su implementación, mejorando significativamente la alineación entre expectativas y resultados[32]. Esta característica resulta particularmente valiosa considerando la complejidad inherente a los procesos de carga y descarga, donde pequeñas interpretaciones erróneas pueden derivar en importantes desviaciones funcionales.

Esta combinación tecnológica no solo aborda las necesidades técnicas inmediatas, sino que establece una infraestructura escalable y sostenible para la calidad del software a largo plazo, permitiendo la incorporación progresiva de más módulos al framework de pruebas automatizadas.

**Referencias Bibliográficas adicionales**

21. Dodero, J. M., & Díaz, P. (2023). Design patterns for test automation frameworks: A systematic literature review. Journal of Systems and Software, 195, 111522.

22. Leotta, M., Clerissi, D., Ricca, F., & Tonella, P. (2023). Approaches and tools for automated test code generation for web applications: A comprehensive review. ACM Computing Surveys, 55(3), 1-38.

23. Mahajan, K., & Sharma, S. (2024). Beyond Page Objects: A Comparative Analysis of Screenplay Pattern Implementation in Modern Test Automation Frameworks. In Proceedings of the International Conference on Software Testing and Analysis (pp. 217-228).

24. García, B., & López-Fernández, L. (2023). High-quality test automation using data-driven techniques: Patterns and anti-patterns. Software Testing, Verification and Reliability, 33(2), e1807.

25. Martínez, R., & Kumar, P. (2024). Factory Pattern Applied to Test Data Generation: A Case Study in Enterprise Resource Planning Systems. Journal of Software: Evolution and Process, 36(3), 2374-2392.

26. Shahin, M., Babar, M. A., & Zhu, L. (2023). Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE Access, 11, 35921-35946.

27. Rahman, A. A. U., Williams, L., & Menzies, T. (2024). Comprehensive analysis of CI/CD pipeline tests: Patterns, anti-patterns, and optimization strategies. IEEE Transactions on Software Engineering, 50(3), 896-913.

28. Fernández, D. M., & Wagner, S. (2023). Test automation engineering: Designing scalable frameworks for CI/CD environments. Empirical Software Engineering, 28(4), 83.

29. Squash Team. (2024). Squash TM Documentation. Open Source Test Management. Retrieved from https://squashtest.com/en/documentation/

30. Behave Documentation. (2024). Behavior-Driven Development in Python. Retrieved from https://behave.readthedocs.io/en/latest/

31. Microsoft. (2024). Playwright: Fast and reliable end-to-end testing for modern web apps. Retrieved from https://playwright.dev/docs/intro

32. Smart, J. F. (2023). BDD in Action: Behavior-driven development for the whole software lifecycle (Second Edition). Manning Publications.