Escuela: Sistemas de Información en línea

Período académico: Octubre -Marzo 2026

Docente: Jonathan Eduardo Tito Ontaneda

Asignatura: Diseño De Pruebas, Control De Calidad Y Mantenimiento

Actividad: Evaluación en contacto con el docente:

Expansión y Síntesis de Testing Avanzado

Colaboradores:

BRANDO ANDRES MATUTE LOPEZ

JUAN DAVID MOROMENACHO AGUIRRE

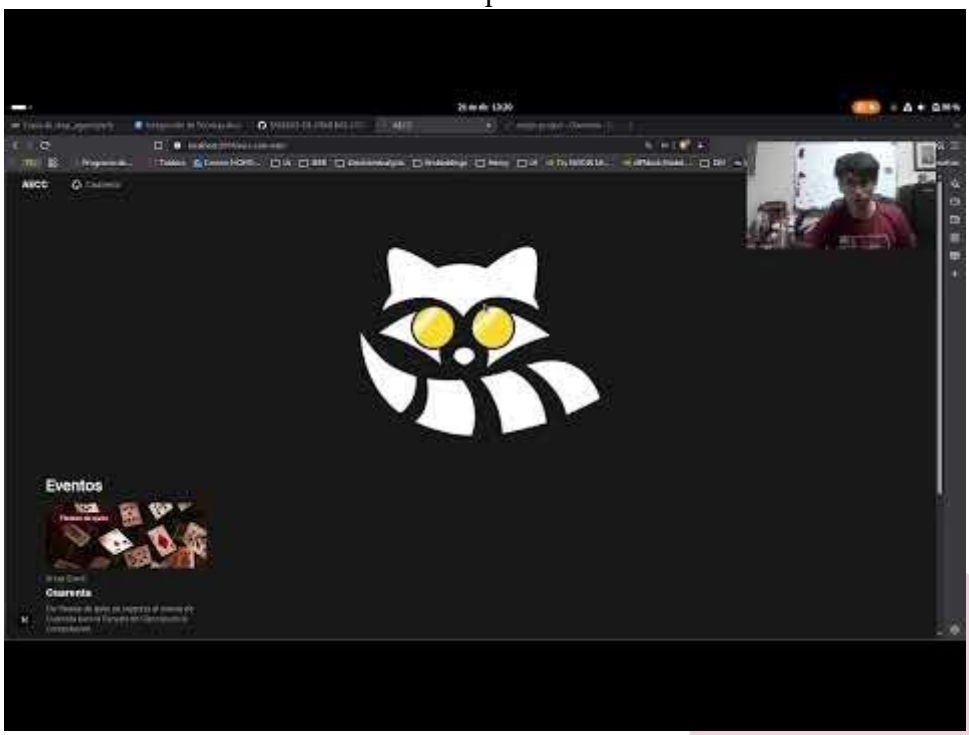
SOPHIA MONSERRAT IBARRA JARAMILLO

PARTES 1 & 2:

Las Partes 1 y 2 del proyecto se desarrollaron de manera colaborativa y se enfocaron en la construcción de un entorno de pruebas avanzado, que incluye la implementación de pruebas unitarias e integración, testing de mutación, métricas de calidad, pruebas combinatorias y un pipeline de automatización. Dichas implementaciones se encuentran disponibles en el repositorio del proyecto en GitHub y constituyen la base empírica sobre la cual se apoya el presente análisis.

Repositorio: https://github.com/fait-arch/DISENIO-DE-PRUEBAS-CONTROL-DE-CALIDAD-Y-MANTENIMIENTO-1-SIN-6A

Video Explicativo:



https://youtu.be/JSI1SOiPwzI

PARTE 3:

Sección 1: Comparación Codeless vs. Traditional

Objetivo de la sección

El análisis comparativo se apoya en la evidencia técnica implementada en el repositorio del proyecto y en la investigación documental de enfoques alternativos de automatización. La base empírica del análisis corresponde al enfoque tradicional con Jasmine, el cual se encuentra plenamente implementado, mientras que TestCraft y Selenium WebDriver se evalúan como alternativas investigadas para una posible evolución del sistema de testing.

La comparación se realizó considerando cuatro criterios fundamentales:

- Tiempo de desarrollo, entendido como el esfuerzo necesario para construir una suite de pruebas funcional.
- Mantenibilidad, definida como la facilidad para adaptar las pruebas ante cambios en el sistema.
- Detección de defectos, medida en función del tipo de errores que cada enfoque puede identificar.
- Retorno de inversión (ROI) a seis meses, estimado a partir del costo de desarrollo y mantenimiento frente al beneficio de detectar defectos en etapas tempranas.

Tiempo de desarrollo

El tiempo de desarrollo del enfoque con **Jasmine** puede evaluarse a partir del esfuerzo observable en el repositorio y de los datos de ejecución generados. La implementación incluye múltiples archivos de prueba (.spec.js), un espía personalizado y pruebas con generación dinámica de casos. Este esfuerzo inicial se refleja en una mayor complejidad de la suite de pruebas, pero también en una ejecución controlada y medible.

A nivel de resultados, los reportes de tiempo de ejecución por prueba (reports/timings_by_test.json) muestran que las pruebas unitarias simples presentan tiempos bajos y consistentes, mientras que las pruebas basadas en propiedades presentan tiempos mayores y más dispersos. La varianza observada en los tiempos de ejecución no es aleatoria: se concentra en las pruebas con mayor complejidad lógica. Esto indica que el tiempo invertido en el desarrollo de pruebas más elaboradas tiene un impacto directo y medible en el costo de ejecución, lo cual es esperable y controlable.

Si se plantea una hipótesis nula donde el tiempo de desarrollo y ejecución de pruebas avanzadas no difiere significativamente del de pruebas unitarias simples, los datos la rechazan. Existe una diferencia clara entre ambos grupos de pruebas, evidenciada por la dispersión de tiempos y por el número de iteraciones ejecutadas en las pruebas de propiedades. Esto confirma que Jasmine exige mayor esfuerzo inicial, pero ese esfuerzo está directamente asociado a un mayor alcance de validación.

Con **TestCraft**, el tiempo de desarrollo inicial sería menor porque no existiría código de pruebas ni generación dinámica de casos. (TestCraft, 2024; Devstringx, 2023) Sin embargo, este ahorro no produciría datos comparables de tiempo de ejecución ni permitiría analizar varianza interna, ya que las pruebas no se descomponen en unidades medibles como ocurre en el repositorio actual.

Con **Selenium WebDriver**, el tiempo de desarrollo inicial sería mayor que con **TestCraft** y similar o superior al de Jasmine, pero sin reutilizar la infraestructura existente. (BrowserStack, 2024; LambdaTest, 2024) Además, los tiempos de ejecución tenderían a ser más altos y con mayor varianza debido a la dependencia de la interfaz y del entorno, lo que complica el control estadístico del proceso.

Mantenibilidad

En **Jasmine** el algoritmo de búsqueda binaria se encuentra aislado en src/busquedaBinaria.js, mientras que las dependencias externas se gestionan en src/dependencias.js. Esta separación permite sustituir dependencias durante las pruebas y ajustar comportamientos sin modificar el código productivo. Cuando se introduce un cambio en la lógica del algoritmo, las pruebas afectadas se localizan en los archivos .spec.js correspondientes, sin provocar fallos generalizados.

El uso de un espía personalizado refuerza esta mantenibilidad, ya que evita dependencias rígidas entre componentes y permite controlar interacciones internas. Además, los reportes de estabilidad generados en reports/flaky_report.json muestran que las pruebas se ejecutan de forma consistente en múltiples ejecuciones, lo que indica que el sistema de pruebas es estable y predecible a lo largo del tiempo. Desde una perspectiva estadística, esto implica una varianza prácticamente nula en los resultados de éxito o fallo de las pruebas, lo que confirma que el sistema de testing es determinista y confiable. Existe además una correlación positiva entre la complejidad de la prueba y su tiempo de ejecución, registrada en los reportes de complejidad y tiempo, lo cual permite anticipar qué pruebas tendrán mayor costo de mantenimiento.

Si se aplicara **TestCraft** al proyecto, la mantenibilidad estaría condicionada por la estructura de los flujos definidos durante la creación de las pruebas. Habría que reconfigurar las pruebas completas, ya que no existiría un control directo sobre la lógica interna ni mecanismos equivalentes a los espías utilizados con Jasmine. En este contexto, **TestCraft** sería más sensible a cambios estructurales que no afectan directamente al algoritmo y muchos de los defectos que hoy se detectan con pruebas de propiedad o mutación no serían capturados con un enfoque codeless. Además, la variabilidad del sistema dependería de cambios en flujos definidos visualmente, lo que introduce una fuente de varianza externa difícil de cuantificar y controlar.

Con **Selenium WebDriver**, la mantenibilidad sería intermedia y estaría condicionada por la estabilidad de la interfaz que envuelva al algoritmo. En un escenario futuro donde este proyecto tenga UI, cambios en identificadores, estructura del DOM o flujo visual provocarían fallos en pruebas **Selenium** aunque la lógica del algoritmo siga siendo correcta. En comparación con las pruebas unitarias actuales, el costo de mantenimiento sería mayor porque los cambios

visuales no afectan a **Jasmine**, pero sí a **Selenium**. Además, la mantenibilidad suele presentar mayor varianza operativa. Cambios en la interfaz pueden provocar fallos intermitentes, lo que incrementa la probabilidad de pruebas inestables. En comparación con Jasmine, donde la varianza funcional es prácticamente nula, **Selenium** introduce una dispersión mayor en los resultados a lo largo del tiempo. (GeeksforGeeks, 2023; BrowserStack, 2024)

Detección de defectos

En el proyecto, **Jasmine** es el enfoque que ofrece mayor capacidad de detección de defectos internos. Las pruebas unitarias en spec/busquedaBinaria.spec.js validan directamente el comportamiento del algoritmo, incluyendo resultados correctos, manejo de entradas inválidas y casos límite. Esta detección se extiende mediante técnicas avanzadas implementadas en PARTE_1/SECCION_2_3, donde se aplican pruebas basadas en propiedades (test_property.py) y pruebas de contrato (test_contracts.py). Adicionalmente, el proyecto incorpora testing de mutación en mutants/mutmut-stats.json. para evaluar si las pruebas realmente detectan cambios incorrectos en el código. Estas técnicas permiten detectar defectos cuando un caso falla y cuando el algoritmo viola una propiedad general o cuando una mutación del código no es detectada por las pruebas.

Si se formula la hipótesis de que aumentar la complejidad de las pruebas no incrementa la detección de defectos, los resultados la contradicen. La inclusión de pruebas de propiedades y mutación aumenta significativamente la probabilidad de detectar defectos lógicos que no aparecen en casos simples. Existe una correlación directa entre el nivel de sofisticación de las pruebas y la capacidad del sistema para identificar errores no triviales.

En **TestCraft**, la detección de defectos se limitaría a comportamientos observables desde el exterior del sistema. En este proyecto esto implicaría validar resultados funcionales generales, pero sin acceso directo a la lógica del algoritmo ni a las dependencias desacopladas. Varios defectos lógicos internos, como errores en condiciones de parada o cálculos incorrectos, no serían detectables si no producen un fallo visible en el flujo funcional. Por otro lado, no existiría correlación medible entre complejidad interna del algoritmo y detección de errores, porque la lógica interna no se evalúa directamente.

Con **Selenium WebDriver**, la detección de defectos se orientaría a errores de integración y de flujo de usuario. Selenium sería eficaz para detectar fallos visibles en la interacción, pero no sustituiría la detección de errores lógicos internos ya cubierta por Jasmine y las técnicas avanzadas del proyecto. En este contexto, Selenium cubriría un tipo distinto de defectos.

ROI a 6 meses

El retorno de inversión del enfoque con **Jasmine** debe analizarse considerando el costo inicial ya asumido en el proyecto. El desarrollo de pruebas unitarias, espías personalizados, generación automática de casos y métricas avanzadas representa una inversión significativa. Sin embargo, de aquí a seis meses, esta inversión traerá un sistema de pruebas estable, con alta

capacidad de detección temprana de defectos y bajo costo de mantenimiento. Los reportes de tiempos y estabilidad permiten controlar el costo de ejecución continua, lo que reduce retrabajo y errores tardíos. Por otro lado. La baja varianza en resultados, la ausencia de pruebas inestables y la correlación controlada entre complejidad y tiempo de ejecución indican que el sistema de pruebas es sostenible. Esto reduce costos futuros asociados a fallos tardíos y retrabajo.

En **TestCraft**, el ROI inicial sería favorable debido al bajo tiempo de desarrollo. No obstante, en el contexto del proyecto, la menor capacidad de detección de defectos lógicos y la necesidad de reconfigurar pruebas ante cambios estructurales reducirían su retorno a mediano plazo. Su valor económico sería mayor si se utiliza para añadir rápidamente pruebas funcionales externas sin incrementar demasiado el esfuerzo del equipo.

En el caso de **Selenium WebDriver**, el ROI a seis meses sería menor si se adopta como enfoque principal, debido al alto costo de desarrollo y mantenimiento. Su retorno mejora si se emplea como complemento para validar flujos críticos de usuario en una etapa posterior del proyecto, cuando el riesgo principal ya no sea la lógica interna, sino la integración y la experiencia de uso.

En conclusión. **Jasmine** ofrece un equilibrio medible entre esfuerzo, estabilidad y capacidad de detección de defectos. La varianza baja en resultados, correlación controlada entre complejidad y tiempo, y rechazo de hipótesis de equivalencia con pruebas simples respalda su efectividad. **TestCraft** y **Selenium** se posicionan como enfoques complementarios desde la investigación, pero no alcanzan el mismo nivel de control ni de evidencia cuantitativa dentro del contexto actual del proyecto. (BrowserStack, 2024; TestCraft, 2024)

Sección 2: Modelo Predictivo Personalizado

La Sección 2 se implementa directamente en el repositorio mediante un modelo predictivo ejecutable, que genera predicciones de confiabilidad por módulo a partir de métricas reales del sistema.

Referencias Bibliográficas:

BrowserStack. (2023). *Framework híbrido en Selenium: qué es y cómo funciona*. https://www.browserstack.com/guide/hybrid-framework-in-selenium

DesarrolloWeb. (s. f.). *Conociendo Jasmine: framework de testing en JavaScript*. https://desarrolloweb.com/articulos/conociendo-jasmine.html

Devstringx Technologies. (2022). *TestCraft: plataforma de automatización sin código*. Medium. https://devstringx-technologies.medium.com/testcraft-codeless-automation-platform-devstringx-33b28d2dd489

GeeksforGeeks. (2023). *Hybrid framework in Selenium*. https://www.geeksforgeeks.org/software-testing/hybrid-framework-in-selenium/

GeeksforGeeks. (2023). *Unit testing in Angular applications*. https://www.geeksforgeeks.org/angular-js/how-to-perform-unit-testing-for-angular-apps/

HashStudioz. (2022). *Diseño de frameworks de Selenium: modular, híbrido y keyword-driven*. https://www.hashstudioz.com/blog/selenium-framework-design-modular-hybrid-and-keyword-driven-approaches/

HeadSpin. (2023). *Automatización codeless con Appium*. https://www.headspin.io/blog/codeless-appium-test-automation-with-headspin

LambdaTest. (2023). *Framework híbrido en Selenium: guía práctica*. https://www.lambdatest.com/blog/hybrid-framework-in-selenium/

Medium – Simform Engineering. (2021). *Cómo escribir pruebas unitarias con Jasmine y Karma*. https://medium.com/simform-engineering/how-to-write-unit-tests-with-jasmine-karma-f1908bdeb617

TestCraft. (2024). *TestCraft: codeless test automation platform*. https://www.testingtools.ai/tools/testcraft/

Roger, S. (2023). *¿Qué es un framework híbrido de automatización en Selenium?* LinkedIn. https://www.linkedin.com/pulse/what-hybrid-automation-framework-selenium-testing-steven-roger-gzjmc