UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

TÍTULO: Automatización de Pruebas de Regresión

AUTORES: Juan Ignacio Borio - Ricardo Javier Paterno

DIRECTOR: Claudia Pons

CODIRECTOR:

ASESOR PROFESIONAL:

CARRERA: Licenciatura en Sistemas

Resumen

La presente tesina tiene como objeto investigar e instrumentar un Proyecto de Testing automático que realiza pruebas de regresión. Se analizan aspectos singulares vinculados con la estructura del proyecto abordado, el contexto en que fue creado, los antecedentes de pruebas manuales y la participación que tuvimos en el desarollo del mismo, y que nos motivó a abordar presente trabajo. Se presenta una reseña de los principales antecedentes de pruebas de software y conceptos relacionados con automatización de pruebas. Se investiga el uso de Herramientas de testing automático, aplicadas a otro tipo de sistemas.

Palabras Clave

Testing automático – Pruebas de regresión – herramientas de testing automático – Clasificación de Pruebas - Automatización sobre aplicaciones web.

Conclusiones

El presente trabajo permite destacar las ventajas del testing automático. La automatización de pruebas de control reduce el esfuerzo de recursos humanos y los tiempos de despliegue, mitigan errores y optimizan la calidad.

Trabajos Realizados

Investigación, formulación e instrumentación de un proceso de testing automatizado enfocado en las pruebas de regresión e investigación de dos herramientas para la automatización de pruebas sobre aplicaciones web.

Trabajos Futuros

Ante la posible migración de módulos del proyecto presentado en el trabajo a tecnologías web, o desarrollo de nuevas aplicaciones de estas características, se contempla profundizar la investigación de herramientas para la automatización sobre aplicaciones web.

Fecha de la presentación: ABRIL 2021

RESUMEN

La presente tesina tiene como objeto investigar e instrumentar un Proyecto de Testing automático que realiza pruebas de regresión.

El proyecto abordado, está realizado en Java y montado sobre la herramienta WindowTester que se utiliza para hacer pruebas automáticas sobre el sistema de una empresa.

Se analizan aspectos singulares vinculados con la estructura del proyecto abordado, el contexto en el que fue creado, los antecedentes de pruebas manuales y la participación que tuvimos en el desarrollo del proyecto, en el área de testing automático y que nos motivó a abordar el presente trabajo.

Se presenta una reseña de los principales antecedentes de pruebas de software, y conceptos nodales relacionados con la automatización de pruebas.

Se investiga el uso de otras herramientas de testing automático, aplicadas a otro tipo de sistemas.

Agradecimientos:

Agradecemos gentilmente a la Directora de este trabajo, la Doctora Claudia Pons por el apoyo como tutora, por su permanente orientación, paciencia y sus correcciones para llevar adelante el presente trabajo, destacando muy especialmente en todo lo actuado su gran calidez humana.

A nuestros padres y madres por el apoyo brindado en la carrera Universitaria, buscando siempre inculcar los más altos valores educativos.

Agradecer también, a la Universidad de Informática de la ciudad de La Plata, por la excelencia en la formación académica que brindan a sus alumnos.

Por último, y de manera personal agradezco a mis hijas Lara, Maia y en especial a mi mujer Paula por motivarme, tenerme paciencia y comprensión durante estos años de carrera. **Ricardo**

ÍNDICE

1. Introducción

El presente trabajo propone investigar sobre la automatización de los test funcionales que realiza el sector de Control de Calidad para las pruebas de regresión de un sistema desarrollado en una empresa. Se trata también de indagar el potencial de las pruebas como herramienta eficaz en los mencionados procesos.

1.1 Objetivos

Objetivo principal:

- Investigar, formular e instrumentar un proceso de testing enfocado en la automatización de las pruebas de regresión que resulte beneficioso en tiempo y esfuerzo en comparación con la ejecución manual.

Objetivos específicos:

- Elaborar una introducción acerca de las pruebas de testing en los proyectos de software.
- Desarrollar un proceso de testing enfocado en la automatización de las pruebas de regresión, basado en aplicar las buenas prácticas y las experiencias obtenidas en nuestra actividad laboral actual.
- Mostrar un módulo de ejecución del proyecto en funcionamiento.
- Investigar y describir nuevas herramientas de testing automático.

1.2 Motivación

Las demandas de la sociedad actual, exigen a las tecnologías de la información, que cumplan con criterios de eficiencia y ahorro en sus tiempos de ejecución. Existen muchos recorridos realizados, en la búsqueda de herramientas que cumplan con tales requisitos.

La motivación que nos lleva a realizar este trabajo es destacar la importancia del testing automático en las pruebas de regresión, reduciendo el periodo de prueba de los productos de software y por ende el tiempo de despliegue. La posibilidad de analizar herramientas alternativas y comparar sus características y posibles beneficios, nos posibilitará seguir enriqueciendo el proyecto a la luz de las conclusiones que alcancemos.
El proyecto de testing automático abordado, automatiza los casos de pruebas diseñados para ser ejecutados como pruebas de regresión. La posibilidad de automatizar las pruebas surge con el propósito de disminuir los costos y tiempos, detectar patrones de errores y evitar su aparición en un futuro.

1.3 Estructura de la tesina

La presente tesina se desarrolla en 5 capítulos.

En el primer capítulo se presenta el objetivo principal del trabajo y los objetivos específicos. Se comparte la motivación que nos impulsó a desarrollarlo y finalmente se brinda una presentación de los distintos capítulos que componen este trabajo.

En el segundo capítulo, destinado al estado del arte, se brindan antecedentes históricos en relación al desarrollo de pruebas de software. Se analizan conceptos básicos inherentes a las mismas, tales como validación y verificación. Se presentan criterios de clasificación de las pruebas y finalmente se realiza una introducción sobre las pruebas de regresión, utilizadas en el proyecto que luego desarrollaremos.

En el tercer capítulo, dedicado al desarrollo del proyecto de testing automático, se relatan antecedentes y propósitos que llevaron a su creación, se amplían las características y las pruebas de regresión aplicadas. Finalmente se comparte la ejecución de un módulo, a fin de mostrar el funcionamiento de las pruebas automatizadas.

En el cuarto capítulo, se describen otras herramientas de automatización, como Selenium WebDriver y Cypress, aplicadas a otro tipo de sistemas.

En el quinto capítulo se presentan las conclusiones a las que arribamos y se abren posibles nuevas rutas de investigación.

Finalmente, se comparte la bibliografía consultada y los anexos.

2. ESTADO DEL ARTE

2.1 Las pruebas de software. Antecedentes

Las pruebas de software se aplican como una fase más del proceso de desarrollo de software, asegurando cumplir con las especificaciones requeridas y evitar posibles defectos que pudiera tener.
En un principio, los procesos de desarrollo tenían una instancia de pruebas más informal, y realizadas únicamente de forma manual.
La primera referencia a las pruebas de software puede ser rastreada en 1950, pero fue recién en 1957 cuando se diferencian del debugging. [Hetzel, 1988]
Myers, en su segunda edición del libro "The Art of Software testing" del año 2004, que fue publicado originalmente en 1979, describe la distancia que existe aún después de los años transcurridos entre una edición y otra, de que las pruebas de testing se constituyan como una ciencia exacta, presentándose como una "zona oscura" entre las etapas del desarrollo de software. [Myers, 2004]
Según el IEEE (Institute of Electrical and Electronic Engineers) estándar 610, define un caso de prueba como: *"conjunto de entrada de prueba, condición de ejecución y resultado esperado desarrollados para un objetivo en particular, como el ejercicio de una ruta de programa en particular o para verificar el cumplimiento de un requisito específico, y como documentación que especifique las entradas, los resultados previos, y un conjunto de condiciones de ejecuciones de un elemento de prueba."* [IEEE 610.12-1990]

Las pruebas de software intentan demostrar que un programa funcione como deba funcionar, así como descubrir defectos del mismo antes de usarlo. Para probar el software, se ejecuta un programa con datos artificiales. Hay que verificar los resultados de la prueba que se opera para buscar errores, anomalías o información de atributos no funcionales del programa. [Sommerville, 2011]
Cem Kaner, profesor de Ingeniería de software en el instituto tecnológico de Florida, las define como "Las pruebas de software son la investigación empírica y técnica realizada para facilitar a los interesados información sobre la calidad del producto o servicio bajo pruebas". [Kaner, 2008]

Edsger W. Dijkstra, científico de la computación, y uno de los primeros contribuyentes al desarrollo de la ingeniería de software afirmaba que “Las pruebas de software pueden ser una manera muy eficaz de mostrar la presencia de errores, pero son totalmente inadecuadas para mostrar su ausencia.” [Dijkstra, 1970]

Bohem, pionero de la ingeniería de software, en 1979 diferenciaba a las pruebas de software entre pruebas de validación y pruebas de verificación, a partir de la posibilidad de responder a los interrogantes de: “¿construimos el producto correcto?” y “¿construimos bien el producto?” para identificar conceptualmente a cada tipo de prueba.

En nuestro país, a partir del año 2009, el Instituto Nacional de Tecnología Industrial (INTI) comenzó a implementar el laboratorio de testing y aseguramiento de la calidad de software. Con ello el INTI tenía como objetivo garantizar el proceso de calidad de los productos de software. [Mera Paz, 2016].

También en [Scalone, F., 2006] se expresa que: “el principal instrumento para garantizar la calidad de las aplicaciones sigue siendo el plan de calidad, el cual se basa en normas o estándares genéricos y en procedimientos particulares. Los procedimientos pueden variar en cada organización, pero lo importante es que estén escritos, personalizados, adaptados a los procesos de organización y que sean cumplidos.” [Scalone, F., 2006].

Watts. S. Humphrey en su libro “Introducción al proceso de software personal”, dedica un capítulo especial al proceso de pruebas de calidad de software. [Humphrey, 2001]

En Suecia, en la Universidad de Gotemburgo en el año 2015, se realizó una investigación en la interactividad de las pruebas de software, basados en búsquedas bajo un experimento controlado. Se emplearon pruebas manuales y automatizadas, evidenciando que las últimas permiten encontrar un mayor número de fallos. Es a partir de este momento, que se inauguró un laboratorio de seguimiento, verificación y validación, implementando herramientas para diferentes análisis. [Mera Paz, 2016]

En España, durante el año 2011, en la Universidad de Sevilla, el ingeniero Rafael Ceballos Guerrero, en su Tesis Doctoral, resaltó que la detección de errores en

etapas tardías genera mayores costos, enfatizando la ventaja de utilizar técnicas automáticas para la detección de fallos en las pruebas de calidad de software. [Ceballos Guerrero, 2011]

En la actualidad las pruebas de software, constituyen una parte central de los proyectos, lo que generó el desarrollo de diversas metodologías.

La existencia de gran cantidad de lenguajes operativos, hardware, metodologías de software y sistemas operativos complejizan la necesidad de encontrar nuevas pruebas. En este contexto, la automatización pareciera presentarse como una opción valiosa al momento de desarrollar la etapa de testing.

La automatización de pruebas es cada vez más requerida dentro del ámbito de desarrollo de software, considerada clave para garantizar la calidad de un producto. Poner un mayor énfasis en la validación y verificación, determinó una tendencia a “tercerizar” las pruebas a compañías dedicadas específicamente a realizar las verificaciones. Sin embargo, desligar las empresas de pruebas de las de desarrollo pareciera no brindar los resultados esperados, no logrando la sinergia necesaria para mejorar la calidad.

La Ingeniería de software ha ido evolucionando en todos los aspectos relevantes, entre ellos las arquitecturas. Como proceso de desarrollo, se observa una ventaja, al incorporar a las pruebas como parte del mismo, a partir del conocimiento de su funcionalidad y cercanía con los desarrolladores.

A pesar del aumento considerable de pruebas automatizadas en los últimos años, pareciera sostenerse la necesidad de realizar pruebas manuales para probar la funcionalidad de algunos aspectos de los sistemas, como así también mantener la cohesión entre quienes diseñan la arquitectura y quienes la testean, de manera de lograr la retroalimentación entre las diferentes etapas del desarrollo.

2.2 Introducción a las pruebas de software

Las pruebas de software intentan demostrar que los sistemas funcionen con el propósito para el que fueron creados y encontrar errores, mediante la entrada de datos artificiales, antes de que el mismo salga a producción.

Existen algunos conceptos básicos para definir una prueba de software:

Una prueba es:

- Una actividad en la que un sistema o componente es ejecutado bajo condiciones especificadas, los resultados son observados o registrados, y una evaluación es hecha de algún aspecto del sistema o componente.
- Un conjunto de uno o más casos de prueba. [IEEE 610.12-1990].

Un caso de prueba (test case) es: un conjunto de valores de entrada, precondiciones de ejecución, resultados esperados y post condiciones de ejecución desarrollados con un objetivo particular o condición de prueba, tal como ejercitar un camino de un programa particular o para verificar que se cumple un requerimiento específico. [IEEE 610.12-1990].

Un procedimiento de prueba (test procedure) es:

- Instrucciones detalladas para la configuración, ejecución y evaluación de los resultados para un caso de prueba determinado.
- Un caso de prueba puede ser usado en más de un procedimiento de prueba. [IEEE 610.12-1990].

En Swebok (Software Engineering Body of Knowledge), documento creado por la Software Engineering Coordinating Committee, promovido por elIEEE se definen los conceptos de prueba, prueba de software, verificación dinámica y comportamiento esperado:

Prueba: actividad realizada para evaluar la calidad del producto y mejorarla, identificando defectos y problemas.

Prueba de software: verificación dinámica del comportamiento de un programa contra el comportamiento esperado, usando un conjunto finito de casos de prueba, seleccionados de manera adecuada desde el dominio infinito de ejecución.

Dinámica: implica que para realizar las pruebas hay que ejecutar el programa para los datos de entrada.

Comportamiento esperado: debe ser posible decidir cuando la salida observada de la ejecución del programa es aceptable o no, de otra forma el esfuerzo de la prueba es inútil. El comportamiento observado puede ser revisado contra las expectativas del usuario, contra una especificación o contra el comportamiento anticipado por requerimientos implícitos o expectativas razonables. [Swebok, 2004].

Desde el enfoque de la Ingeniería de software el proceso de pruebas tiene dos metas:

1. Demostrar que el software cumple los requerimientos para los que fue creado. Esto significa que debe tener pruebas para cada requerimiento.

2. Encontrar situaciones donde el software se comporte incorrectamente o no esté de acuerdo a las especificaciones. Las pruebas de defectos tienen la finalidad de erradicar el comportamiento indeseable: caída del sistema, cálculos incorrectos.

La primera meta conduce las pruebas de validación, en la cual se espera que el sistema se desempeñe correctamente de acuerdo a los casos de pruebas que reflejen el uso del sistema. Y la segunda se orienta a pruebas defectos, donde los casos de prueba se diseñan para presentar los defectos.

Los procesos de verificación y validación buscan comprobar que el software por desarrollar cumpla con sus especificaciones y brinde la funcionalidad deseada por el cliente. Estos procesos comienzan tan pronto como están disponibles los requerimientos y continúan en toda la etapa de desarrollo.

La definición dada por [IEEE 610.12-1990] de ANSI /IEEE 1990, de Verificación y Validación es la siguiente:

- Verificación: proceso de evaluación de un sistema o componente para determinar si un producto de una determinada fase de desarrollo satisface las condiciones impuestas al inicio de la fase.
- Validación: proceso de evaluación de un sistema o componente durante o al final del proceso de desarrollo para determinar cuándo se satisfacen los requerimientos especificados.

[CMMI02] define los propósitos de Verificación y validación de la siguiente manera:

- Propósito de Verificación: es asegurar que los productos internos seleccionados cumplen con su especificación de requerimientos. Los métodos de verificación pueden ser, entre otros: inspecciones, revisiones por pares, auditorías, recorridos, análisis, simulaciones, pruebas y demostraciones.
- Propósito de Validación: es demostrar que un producto o un componente de un producto cumple su uso previsto cuando es puesto en su ambiente previsto. Deben ser seleccionados los productos internos (por ejemplo: requerimientos, diseño, prototipos), que mejor indican cuán bien el producto y los productos internos deben satisfacer las necesidades del usuario.

No hay una frontera definida entre estos dos enfoques de prueba. Es posible descubrir defectos durante las pruebas de validación, como descubrir que el programa cumple con los requerimientos en las pruebas de defecto. En la figura 1, se observa el modelo de entrada y salida de una prueba de programa.

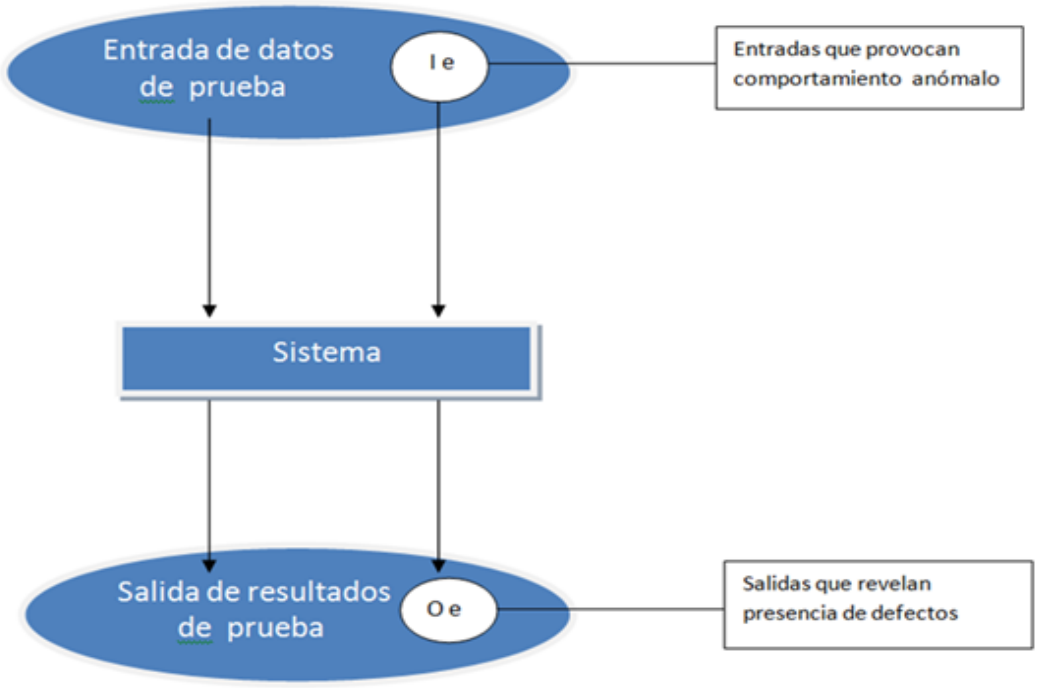

Figura 1. Modelo de entrada y salida de una prueba de Programa

Tanto las pruebas de validación como las de verificación se sostienen desde la aparición de los requerimientos y durante todas las etapas del proceso de desarrollo.

Las pruebas de un programa, donde el sistema se ejecuta a partir de un conjunto de casos simulados, son la principal técnica de validación.

2.3 Clasificación de pruebas de software

En función de las etapas del desarrollo, las pruebas pueden clasificarse en:

- Pruebas de desarrollo: el sistema se pone a prueba con el fin de descubrir defectos.
- Versiones de prueba: se realiza una prueba del sistema, antes de que llegue al usuario, a fin de descubrir si cumple con los requerimientos para lo que fue creado.

- Pruebas de usuario: en este caso pueden realizarse con usuarios reales o ficticios, para comprobar si es necesario ampliar el desarrollo.

Es factible que los errores de una etapa se hagan visibles en otra, como la detección de errores de sus componentes en el proceso, o problemas de interfaz cuando el sistema se integra. Es por ello, que el sistema es interactivo, con información retroalimentada desde etapas posteriores, hasta las partes iniciales del proceso. [Sommerville, 2011].

Los procesos de prueba en general requieren de pruebas manuales y otras automatizadas.

Las pruebas son procesos que se enfocan sobre la lógica interna del software y también sobre las funciones externas.

La prueba es un proceso de ejecución de un programa con la intención de descubrir un error que probablemente no fue previsto en las fases iniciales del desarrollo del software.

Según Whitaker, las pruebas pueden clasificarse según el tipo de prueba que se esté haciendo en funcional o estructural, y según como se realiza la prueba en unitaria, de integración o del sistema. [Whittaker, 2002]

- **Prueba funcional**: también conocida como **caja negra o** basada en la especificación**.** El objetivo es validar cuando el comportamiento observado del software probado cumple o no con sus especificaciones. Toma el punto de vista del usuario.
- **Prueba estructural**: también llamada **caja blanca** o basada en el código. El objetivo es seleccionar los caminos del programa a ejercitar durante las pruebas.
- **Prueba unitaria**: es el proceso de probar los componentes individuales de un programa, a fin de descubrir discrepancias entre la especificación de la interfase de los módulos y su comportamiento real. La prueba de unidad se centra en el módulo. Usando la descripción del diseño detallado como guía, se prueban los caminos de control importantes con el fin de descubrir errores dentro del ámbito del módulo. La prueba de unidad hace uso

intensivo de las técnicas de prueba de caja blanca. Este tipo de prueba la realiza, generalmente, el desarrollador luego de la codificación de un módulo.

- **Prueba de integración**: es el testing que intenta mostrar que, aunque funcionen los componentes de manera aislada, el sistema falla cuando se integran sus módulos. La integración puede ser realizada de diferentes maneras: de manera incremental, probando grupos de módulos que luego se unen en un sistema completo; o acorde a la arquitectura, uniendo módulos de acuerdo a la funcionalidad. El objetivo es tomar los módulos testeados en la prueba de unidad y construir una estructura de programa que esté de acuerdo con lo que dicta el diseño.
- **Prueba del sistema**: se evalúa el comportamiento del sistema entero, luego de probar sus componentes. Se evalúan en esta instancia los requerimientos funcionales del sistema como exactitud, seguridad, desempeño, confiabilidad, interfaces, ambientes. La prueba del sistema está constituida por una serie de pruebas diferentes cuyo propósito primordial es ejercitar profundamente el sistema basado en computadora. Algunas de estas pruebas son:
    - Prueba de validación: proporciona una seguridad final de que el software satisface todos los requerimientos funcionales y de rendimiento. Además, valida los requerimientos establecidos comparándolos con el sistema que ha sido construido. Durante la validación se usan exclusivamente técnicas de prueba de caja negra.
    - Prueba de recuperación: se fuerza a un fallo del software y se verifica que la recuperación se lleva a cabo apropiadamente.
    - Prueba de seguridad: verificar los mecanismos de protección.
    - Prueba de resistencia: se somete el sistema a operaciones anormales.
    - Prueba de rendimiento: prueba el rendimiento del software en tiempo de ejecución.
    - Prueba de instalación: se centra en asegurar que el sistema software desarrollado se puede instalar en diferentes

configuraciones de hardware y software y bajo condiciones excepcionales, por ejemplo, con espacio de disco insuficiente o continuas interrupciones.

Otro tipo de pruebas existentes son las pruebas de regresión, las pruebas de humo o las pruebas automatizadas.

2.4 Pruebas de regresión

Las pruebas de regresión son una estrategia de prueba en la cual las pruebas que se han ejecutado anteriormente se vuelven a realizar en la nueva versión modificada, para asegurar la calidad después de añadir la nueva funcionalidad. El propósito de estas pruebas es asegurar que los defectos identificados en la ejecución anterior de la prueba se hayan corregido, que los cambios realizados no introduzcan nuevos o viejos defectos. La prueba de regresión puede implicar la re-ejecución de cualquier tipo de prueba. Normalmente, las pruebas de regresión se llevan a cabo durante cada iteración, ejecutando otra vez las pruebas de la iteración anterior. [Maida - Pacienzia, 2015]

Las pruebas de regresión pueden clasificarse en:

- Regresión de defectos seleccionados: cuando se reporta un defecto y vuelve una nueva versión que supuestamente lo soluciona, el objetivo es probar que no fue solucionado.
- Regresión de defectos viejos. Se prueba que un cambio en el software causó que un defecto que ya había sido solucionado vuelva a aparecer.
- Regresión de efectos secundarios. Implica volver a probar una parte del producto. El objetivo es probar que el cambio ha causado que algo que funcionaba ya no funcione. [Kaner - Bach, 2001]

3. DESARROLLO DEL PROYECTO

3.1 Características del Proyecto. Antecedentes

La empresa en donde trabajamos, que por motivos de confidencialidad nombraremos como Security Inc., es una empresa dedicada a la gestión financiera. La figura 2 muestra el organigrama de la empresa.

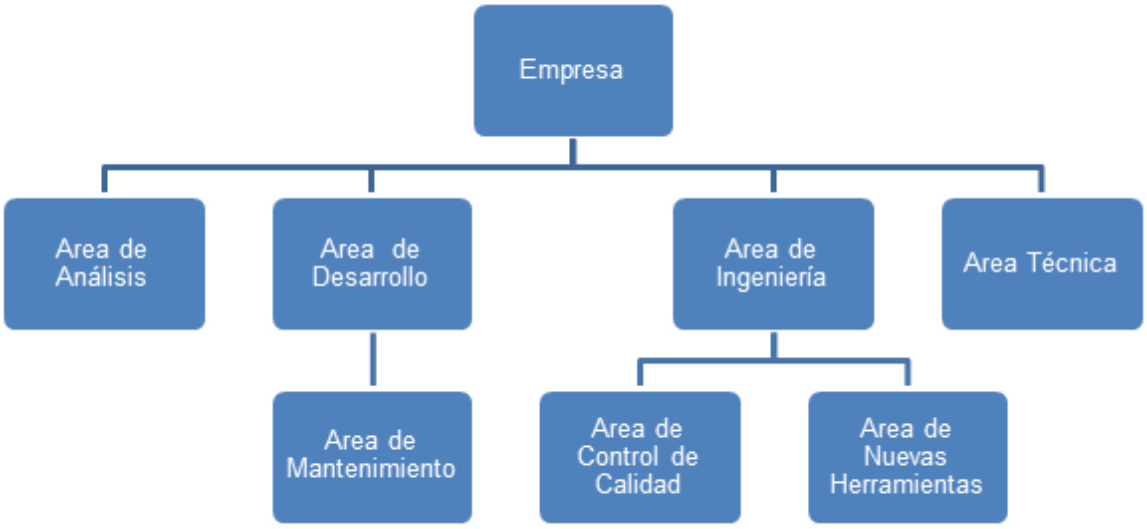

Figura 2. Organigrama de la empresa

La empresa Security Inc., cuenta con un sistema de gestión financiera y contable, dicho sistema gestiona comprobantes de distintos negocios (módulos), y está desarrollado en Java, por el área de Desarrollo, utilizando como framework para su diseño y desarrollo la interfaz gráfica SWT (Standard Widget Toolkit) (un conjunto de componentes para construir interfaces gráficas en Java), cada versión

de la aplicación es instalada en los entornos testing (Control de Calidad), para ser puesta bajo pruebas hasta su despliegue en producción.

En un principio las pruebas de control de calidad se realizaban en forma manual por el grupo de testing perteneciente al área de Gestión de calidad, lo cual demandaba mucho esfuerzo de recursos humanos y demoraba los tiempos de despliegue. Las pruebas de regresión se hacían manualmente y por su repetitividad se volvían monótonas, llevaban a cometer errores y a no prestarle tanta importancia a los casos de prueba.

A fin de optimizar la calidad, mitigar errores y reducir el tiempo de salida a producción de la aplicación, se creó un proyecto con el objetivo de automatizar los test funcionales que se realizaban para las pruebas de regresión. En la actualidad el tiempo que les demandaba a los testers realizar las pruebas de regresión manuales, es utilizado para otras tareas (diseñar casos de pruebas, analizar fallas, etc.)

Este proyecto es desarrollado con el fin de implementar una nueva forma de ejecutar las tareas de testing. Si bien no se reemplazan las tareas de testing manual (son las que se hacen normalmente al programar o las que ejecuta una persona con la documentación generada), se agrega una mayor cobertura de casos de prueba, agiliza los tiempos de testeo y permite la reutilización de los mismos. Para ello se usa un determinado software con el objeto de sistematizar las pruebas y obtener los resultados de las mismas.

Pruebas automáticas. Definición

Son pruebas funcionales que usan herramientas para automatizar la emulación de un usuario interactuando con la aplicación y verifican los pasos de las pruebas usando assertions (verificaciones de programación).

La figura 3 muestra algunos de los beneficios de los test automáticos

Figura 3. Beneficios de los test automáticos

El primer beneficio habla de que los test automáticos reducen costos y tiempos por ejemplo un caso de uso automatizado puede llegar a tardar entre 5 y 10 minutos en ejecutarse automáticamente dependiendo del mismo. Este mismo caso de uso en forma manual demandaría 2 días en hacer todas las verificaciones requeridas, este tiempo que se gana es aprovechado para realizar otras tareas.

El segundo beneficio. Mejorar pruebas ágiles en entornos de integración continua, no aplica en nuestro proyecto, pero habla de que en otras metodologías ágiles que utilizan pipeline se van probando en cada pipeline cada vez que hay un despliegue ciertos test a medida a que se desarrolla.

El tercer beneficio habla de ayuda de Control de calidad en los despliegues. La automatización de las pruebas detecta las fallas en forma más rápida ayudando a resolverlas en menor tiempo, disminuyendo por ende en tiempo de despliegue.

El cuarto beneficio habla del aumento de la velocidad de las pruebas, ya que al ser automáticas se ejecutan a mayor velocidad, y son repetibles y precisas, una

vez desarrolladas, se pueden ejecutar repetidas veces en cualquier momento y como son precisas evitan fallas humanas.

3.2 Desarrollo del Proyecto

El proyecto de testing automatizado, que llamamosTesting_automático, esun proyecto desarrollado en Java por un grupo de desarrolladores del área de Gestión de Calidad, del cual formamos parte, que está montado sobre la herramienta WindowTester y se utiliza para hacer pruebas automáticas sobre el sistema de administración financiera de la empresa nombrado anteriormente. El mismo utiliza varias herramientas, que describiremos en la siguiente sección.

3.2.1 Herramientas utilizadas

- **Eclipse**

  Herramienta de desarrollo. Es un entorno de desarrollo integrado de código abierto multiplataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido" (rcp), opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores. Es una potente y completa plataforma de programación, típicamente ha sido usada para desarrollar entornos de desarrollo integrados (del inglés IDE), como el IDE de Java llamado Java Development Toolkit (JDT) y el compilador (ECJ) que se entrega como parte de Eclipse (y que son usados también para desarrollar el mismo Eclipse).

- **CVS**

  Es un plugins de eclipse, conocido como **Concurrent Versioning System**, es una aplicación informática que implementa un sistema de control de versiones: mantiene el registro de todo el trabajo y los cambios en los archivos (código fuente principalmente) que forman un proyecto (de

programa) y permite que distintos desarrolladores (potencialmente situados a gran distancia) colaboren. CVS se ha hecho popular en el mundo del software libre. Sus desarrolladores difunden el sistema bajo la licencia GPL.

- **Java**

  Lenguaje de programación utilizado para desarrollar el proyecto, ya que el software que se testea está implementado en Java.

- **JUnit**

  Es un conjunto de bibliotecas utilizadas en programación para hacer pruebas unitarias de aplicaciones Java. JUnit está compuesto por clases (*framework*) que permiten realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente.

  JUnit es también un medio de controlar las pruebas de regresión, necesarias cuando una parte del código ha sido modificado y se desea ver que el nuevo código cumple con los requerimientos anteriores y que no se ha alterado su funcionalidad después de la nueva modificación.

- **WindowTester**

  Herramienta que permite crear test automáticos para aplicaciones gráficas construidas con swing o con SWT. La herramienta permite grabar todos los eventos de teclado y ratón realizados durante el testeo de una aplicación, y después es capaz de reproducir todos esos eventos nuevamente como test de JUnit. Esta automatización mejora la productividad tanto del desarrollador como del equipo de Control de calidad. Se brinda al

desarrollador la capacidad de crear sus propios tests, reduciendo el esfuerzo requerido. Los casos de test generados son JUnit estándares y por lo tanto pueden ejecutarse desde Eclipse o Ant. Para grabar el comportamiento de una aplicación normalmente se usa Eclipse y, para probar una aplicación desplegada, primero se la importa en el ambiente de trabajo. El proceso de grabación consiste en que la herramienta va registrando las acciones del usuario sobre la aplicación, brindando funcionalidades adicionales para el desarrollo de tests automáticos, siendo una de ellas la transformación automática a código Java.

WindowTester consiste principalmente de dos componentes: La componente *Runtime*, que tiene los paquetes necesarios para la ejecución de Tests Automáticos, y la componente *IDE*, que contiene los paquetes que proporcionan los accesorios para Eclipse, principalmente la Grabadora.

Esta herramienta es OpenSource primero perteneció a Instantations Inc. Y fue adquirida por Google en el año 2010.

3.2.2 Elementos del Proyecto

En la figura 4, se muestra el modelo del proyecto y a continuación se explica las partes que lo componen.

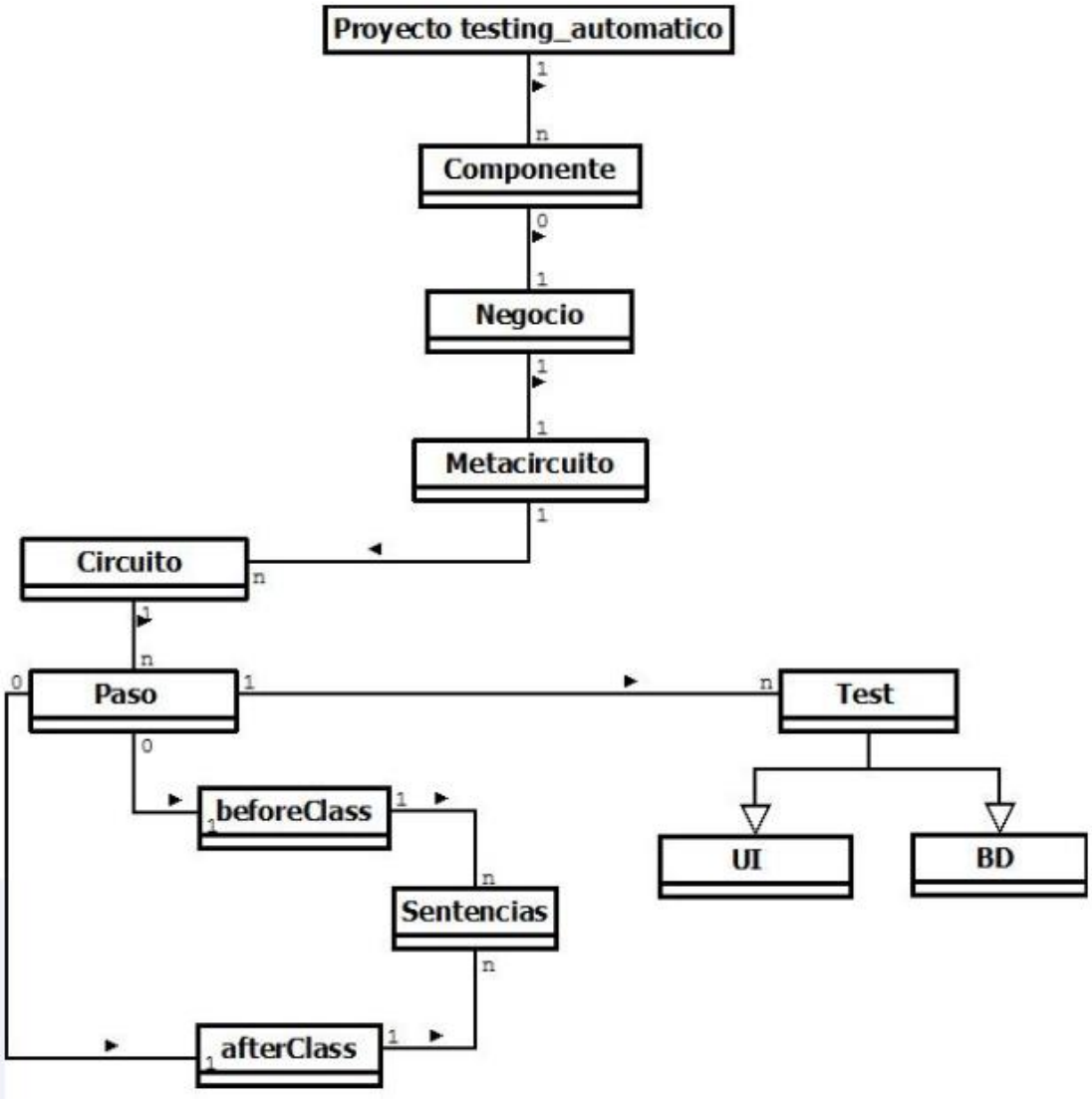

Figura 4. Modelo del proyecto

- **Componente negocio:**

  Los negocios (módulos) implementados en nuestro proyecto, son los distintos componentes que representa a los negocios del sistema administrativo testeado. Cada negocio en el sistema administrativo cuenta con un conjunto de comprobantes que son gestionados y transicionados

por los usuarios finales. En nuestro proyecto cada componente negocio, tiene una arquitectura del mismo, en la cual existen clases java que modelan los comprobantes que se utilizan para ser manipulados y para obtener los datos artificiales a popular en los mismos. Además, cada componente negocio cuenta con scripts, archivos de propiedades, archivos con datos artificiales a utilizar por las clases Test como ser los metacircuitos, circuitos, pasos y verificaciones. Los circuitos a ser ejecutados representan los casos de uso diseñados por el Área de Análisis como requerimiento y desarrollados en nuestro proyecto Testing_automático, para simular las acciones de los usuarios finales y comprobar mediante test implementados, que se cumplan las verificaciones requeridas en cada acción o paso de usuario.

- **Metacircuito:**

Un metacircuito está formado por un conjunto de circuitos de un negocio en particular, y la finalidad de este es que se ejecuten todos los circuitos que contenga. Son agrupadores de Circuitos. Los metacircuitos son clases que extienden implícitamente a Object y tienen un único método estático (de clase) llamado suite() (una suite es un conjunto de casos de prueba en este contexto es un conjunto de clases de tipo Circuito a ser ejecutados) que no recibe parámetros y retorna un objeto de tipo junit.framework.Test. El cuerpo de este método, instancia la clase SetupDecoratedTestSuiteWT clase de nuestro proyecto, cuyo constructor recibe el nombre de la clase del metacircuito y un arreglo de objetos tipo java.lang.Class que representan los circuitos.

Un metacircuito está formado por circuitos de un negocio en particular. Es un Test Suite (conjunto de casos de prueba) de Test Suite. En la figura 5 se puede observar un ejemplo de un metacircuito.

```
1  public class NombreMetacircuito {
2      public static Test suite() {
3          return new SetupDecoratedTestSuiteWT(NombreMetacircuito, new Class[] {
4              NombreCircuito1.class,
5              NombreCircuito2.class,
6              ...
7              NombreCircuitoN.class,
8          });
9      }
10 }
```

Figura 5. Ejemplo de metacircuito

- **Circuito:**

  Un circuito está formado por pasos. Un circuito es una suite, es un conjunto de pasos (clases) que se van a ejecutar. Es una clase que extiende implícitamente de Object y tiene un método estático (de clase) llamado suite() que no recibe parámetros y retorna un objeto de tipo junit.framework.Test. (Caso de prueba). El cuerpo de este método instancia la clase SetupDecoratedTestSuiteWT clase de nuestro proyecto cuyo constructor recibe el nombre de la clase del circuito y que internamente para los circuitos determina cuáles pasos va a ejecutar, en esta caso dependiendo de una condición llama al método getNacion() o getProvincia(), que nos devuelve un arreglo de clases pasos.

  Además, un circuito tiene una anotación @Circuito que brinda información adicional. Para internamente saber de qué negocio es el circuito, donde ir a buscar los datos a popular en el mismo y una descripción breve de que hace el circuito. La figura 6 muestra un ejemplo de un circuito.

```
@Circuito(negocio = Negocio.NEG, instancia = NroInstancia, descripcion="descripcionCircuito")
public class NombreCircuito {
    public static Test suite() {
        return new SetupDecoratedTestSuiteWT(NombreCircuito);
    }
    public static Class<?>[] getPasosProvincia() {
              return new Class<?>[] {
                  InvalidarCircuito.EjecucionIlegalEnPCIA.class,
              };
    }

    public static Class<?>[] getPasosNacion() {
              return new Class<?>[] {
                  Paso1.class,
                  Paso2.class,
                  ...
                  PasoN.class,
              };
    }
}
```

Figura 6. Ejemplo de circuito

Los circuitos pueden ser:

Circuitos normales: son los circuitos que siguen una secuencia normal de pasos de un negocio, pueden variar de acuerdo al diseño.

Circuitos Tratares: son los circuitos que contienen pasos tratares, los cuales intentan realizar acciones de usuario que arrojan errores de validación. Por lo general se dejan campos obligatorios sin rellenar y se trata de realizar una acción en la cual sean requeridos dichos campos, de modo de forzar mensajes de validación, los cuales se almacenan en variables, se cierran dichas ventanas de error y se realizan los test que verifican los mensajes esperados contra los mensajes obtenidos. Luego de los test, se ejecuta el método afterClass para realizar el ingreso correcto de datos de dichos campos obligatorios y de esta manera poder proseguir la secuencia del circuito con un nuevo paso. Otro ejemplo de un paso tratar, puede ser cambiar de usuario y tratar de que se realice una acción para la cual el usuario no tenga permisos o capacidades, ejemplo que el ítem del menú para esta acción con el usuario logueado se encuentre deshabilitado.

En la figura 7 se puede ver un ejemplo de un circuito tratar en donde en un comprobante no se ingresan campos obligatorios requeridos del solicitante y al salvarlo, se puede observar cómo se abre una ventana de error con los mensajes de validación, los cuales se capturan y se comparan en los test con los mensajes esperados.

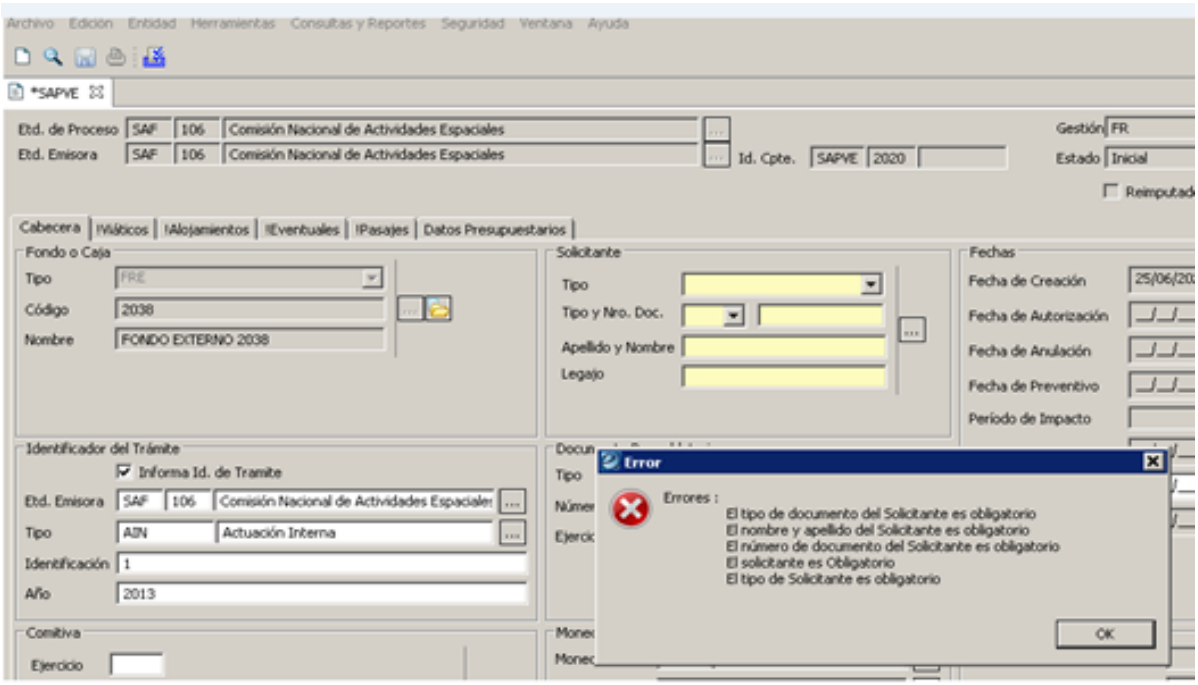

Figura 7. Ejemplo de circuito tratar

- **Paso:**

  Un paso simula una o más acciones que realiza un usuario. En nuestro proyecto es una clase que puede extender a com.windowtester.runtime.swt.UITestCaseSWT o junit.framework.TestCase según se requiera de la Interfaz Gráfica o no.

Un paso puede estar compuesto por:

Pasos de usuarios (métodos beforeClass y afterClass), métodos test y Subpasos que son inner class (clases internas) del paso. Estos subpasos se utilizan para heredar funcionalidad del mismo. En la figura 8 se muestra un ejemplo de cómo se define un paso.

```
1  public abstract class NombrePaso extends TAUITstCaseSWT {
2          public void beforeClass() throws Exception {
3                  IUIContext ui = getUI();
4                  Manejador.accionARealizar1();
5                  Manejador.accionARealizar2();
6                  ...
7                  Manejador.accionARealizarN();
8          }
9
10         public void testNombreTest1() throws Exception {
11                 ManejoTest.AssertEquals(esperado, actual);
12                 ManejoTest.AssertNotEquals(esperado, actual);
13         }
14
15         public void testNombreTest2() throws Exception {
16                 ManejoTest.AssertEquals(esperado, actual);
17                 ManejoTest.AssertNotEquals(esperado, actual);
18         }
19         ...
20         public void testNombreTestN() throws Exception {
21                 ManejoTest.AssertEquals(esperado, actual);
22                 ManejoTest.AssertNotEquals(esperado, actual);
23         }
24
25         public void afterClass() throws Exception {
26                 IUIContext ui = getUI();
27                 Manejador.accionARealizar1();
28                 Manejador.accionARealizar2();
29                 ...
30                 Manejador.accionARealizarN();
31         }
32
33         public static class PARA___NombreCircuito
34                 extends NombrePaso {
35                         ...
36         }
37 }
```

Figura 8. Ejemplo de un paso

Tanto los métodos beforeClass, afterClass o test pueden ser anotados con las siguientes annotations (anotaciones) de java:

@Bug: Es una anotación java que sirve para etiquetar los métodos que dan error, fuera cual fuese su naturaleza. Ejemplo: widget no encontrado, valor esperado difiere del valor actual, etc. En esta anotación figura el número de bug, desarrollador que detectó el mismo, y una descripción.

@Hack: Es una annotation java que sirve para etiquetar los métodos que, por el motivo que sea, difieren intencionalmente de su diseño. De este modo, el log de ejecución descubre la anotación y registra una advertencia que sirve para no perder de vista la situación. Otros usos que se le han dado hasta ahora fueron: cuando se modifican los test para sortear un obstáculo que impide la prueba o cuando se deshabilitan porciones del código.

Método beforeClass

Este método fue creado para nuestro proyecto es una extensión del método setUp() de JUnit3, se ejecuta una sola vez por paso y en nuestro proyecto lo utilizamos para desarrollar las acciones que realiza un usuario final de la aplicación en un paso particular a modo de simular su interacción con la misma, llegando de esta forma a un estado concreto de la aplicación. Es el primer método que se ejecuta de un determinado paso. En el cuerpo de este método, generalmente, una de las primeras sentencias es la obtención de la interfaz usuario, a través de la invocación del método getUI() heredado de la clase **com.windowtester.runtime.common.UITestCaseCommon** que retorna un objeto de una clase que implemente la interfaz **com.windowtester.runtime.IUIContext**. Entre otros usos integra a los elementos que conforman una verificación en la interfaz gráfica: la interfaz de usuario, la referencia al widget y la condición con el valor esperado.

IUIContext ui = getUI();

A este objeto UI se le pueden invocar las acciones:

click (ILocator)

wait(Condition)

enterText(String)

keyClick(char)

Ejemplo:

```
public void beforeClass() throws Exception  {
IUIContext ui = getUI();
ui.click(new UnLocator());
ui.wait(new unCondition());
ui.enterText("untexto");
}
```

Locator:

Es parecido a los punteros de otros lenguajes de programación en cuanto sirven para referirse a un widget, sin serlo ellos mismos; los describen y se emplean en su lugar. Se utilizan principalmente para buscar widgets a partir de alguna característica distintiva.

Por ejemplo:

```
new ButtonLocator("Aceptar");
```

Es capaz de ubicar un botón con el texto "Aceptar", sin importar su ubicación en la pantalla. Los conditions reciben Locators para referirse a widgets. En general los Locators que provee windowTester son capaces de ubicar cualquier elemento SWT pero no siempre sucede lo mismo con

nuestros widgets de nuestra aplicación y en esos casos tenemos que buscar alternativas. Una de ellas es usar NamedWidgetLocator (como técnica para encontrar widgets esquivos, windowTester sugierela posibilidad de asignarles manualmente un nombre. Esta alternativa debería considerarse un último recurso ya que obliga a modificar la aplicación, requiere el mantenimiento de un listado de nombres asignados, no es aplicable en casos como las grillas y hay veces en que bien puede sustituirse por locators menos intrusivos. Si el empleo de otros locators no es posible. De ser necesario, se solicita al Área de Diseño y desarrollo el nombre.

Condition:

Se comporta como expresión booleana que comúnmente define verificaciones.

Por ejemplo:

new ShellShowingCondition("Información"))

Puede usarse como argumento de un assert para averiguar si está visible una ventanade información. Varios tipos de condiciones se pueden asociar a un widget, como por ejemplo comprobar si está habilitado o si contiene cierto valor conocido, si tiene foco, si es visible, etc. Cada caso se implementa en una clase aparte, que hereda de ICondition y, si no se dispone de ninguna acorde a nuestra necesidad, generalmente la desarrollamos.

Ejemplos:

IsEnabledCondition

HasTextCondition

HasFocusCondition

IsVisibleCondition

Assertion

Es el mecanismo principal de windowTester para implementar verificaciones y consiste en evaluar el estado de una condición. A su vez, las condiciones comparan un valor obtenido contra un valor esperado y se separan en dos grupos: las que involucran a los widgets (elementos gráficos) y las que no. Las que los involucran se ejecutan mediante un contexto provisto por windowTester, que representa a la interfaz de usuario (UIContext), mientras que las demás recurren a los varios métodos assert * ofrecidos por JUnit.

Métodos Assert en junit

Static void	fail () Falla el test sin ningún mensaje
Static void	fail (java.lang.String message) Falla el test con el mensaje indicado.
Static void	assertEquals (boolean expected, boolean actual) Verifica que los valores proporcionados sean iguales.
Static void	assertEquals (byte expected, byte actual) Verifica que los valores proporcionados sean iguales.

Static void	assertEquals (char expected, char actual) Verifica que los valores proporcionados sean iguales.
Static void	assertEquals (double expected, double actual, double delta) Verifica que los valores proporcionados sean iguales tomando en cuenta la tolerancia indicada por el parámetro delta.
Static void	assertEquals (float expected, float actual, float delta) Verifica que los valores proporcionados sean iguales tomando en cuenta la tolerancia indicada por el parámetro delta.
Static void	assertEquals (int expected, int actual) Verifica que los valores proporcionados sean iguales.
Static void	assertEquals (long expected, long actual) Verifica que los valores proporcionados sean iguales.
Static void	assertEquals (java.lang.Object expected, java.lang.Object actual) Verifica que los valores proporcionados sean iguales.

Static void	assertEquals (short expected, short actual) Verifica que los valores proporcionados sean iguales.
Static void	assertEquals (java.lang.String message, boolean expected, boolean actual) Verifica que los valores proporcionados sean iguales. Si no lo son, envía el mensaje indicado por el parámetro message.
Static void	assertEquals (java.lang.String message, byte expected, byte actual) Verifica que los valores proporcionados sean iguales. Si no lo son, envía el mensaje indicado por el parámetro message.
Static void	assertEquals (java.lang.String message, char expected, char actual) Verifica que los valores proporcionados sean iguales. Si no lo son, envía el mensaje indicado por el parámetro message.
Static void	assertEquals (java.lang.string message, double expected, double actual, double delta) Verifica que los valores proporcionados sean iguales dentro de la tolerancia especificada por el parámetro delta. Si no lo son, envía el mensaje indicado por el parámetro message.

Static void	assertEquals (java.lang.string message, float expected, float actual, float delta) Verifica que los valores proporcionados sean iguales dentro de la tolerancia especificada por el parámetro delta. Si no lo son, envía el mensaje indicado por el parámetro message.
Static void	assertEquals (java.lang.String message, int expected, int actual) Verifica que los valores proporcionados sean iguales. Si no lo son, envía el mensaje indicado por el parámetro message.
Static void	assertEquals (java.lang.String message, long expected, long actual) Verifica que los valores proporcionados sean iguales. Si no lo son, envía el mensaje indicado por el parámetro message.
Static void	assertEquals (java.lang.String message, java.lang.Object expected, java.lang.Object actual) Verifica que los valores proporcionados sean iguales. Si no lo son, envía el mensaje indicado por el parámetro message.
Static void	assertEquals (java.lang.String message, short excepted, short actual) Verifica que los valores proporcionados sean iguales. Si no lo son, envía el mensaje indicado por el parámetro message.

Static void	assertEquals (java.lang.String expected, java.lang.String actual) Verifica que los valores proporcionados sean iguales.
Static void	assertEquals (java.lang.String message, java.lang.String expected, java.lang.String actual) Verifica que los valores proporcionados sean iguales. Si no lo son, envía el mensaje indicado por el parámetro message.
Static void	assertFalse (boolean condition) Verifica que el objeto boolean seleccionado sea falso.
Static void	assertFalse (java.lang.String message, boolean condition) Verifica que el objeto boolean seleccionado sea falso.
Static void	assertNotNull (java.lang.Object object) Verifica que el objeto proporcionado no sea null.

Static void	assertNotNull (java.lang.String message, java.lang.Object object) Verifica que el objeto proporcionado no sea null y envía un mensaje en caso de que sea cierto.
Static void	assertNotSame (java.lang.Object expected, java.lang.Object actual) Verifica que los objetos proporcionados no son los mismos.
Static void	assertNotSame (java.lang.String message, java.lang.Object expected, java.lang.Object actual) Verifica que los objetos proporcionados no son los mismos.
Static void	assertNull (java.lang.Object object) Verifica que el objeto proporcionado es null.
Static void	assertSame (java.lang.Object expected, java.lang.Object actual) Verifica que los objetos proporcionados son los mismos.

Static void	assertSame (java.lang.String message, java.lang.Object expected, java.lang.Object actual) Verifica que los objetos proporcionados son los mismos enviando un mensaje en caso de que no se cumpla la condición.
Static void	assertTrue (boolean condition) Verifica que el parámetro proporcionado true.
Static void	assertTrue (java.lang.String message, boolean condition) Verifica que el parámetro proporcionado sea true enviando un mensaje de error en caso de que no sea así.

Método afterClass

Este método pertenece a nuestro proyecto y es una extensión del método de JUnit3 tearDown(). Se implementan los pasos del usuario final de la aplicación, a modo de simular su interacción con la misma. Este método es ejecutado una sola vez por paso y se ejecuta después de que todos los tests hayan terminado. Generalmente se utiliza para cerrar vistas, deshacer un determinado proceso. Es el último método que se ejecuta en un determinado paso.

Ejemplo:

```
public void afterClass() throws Exception {
getUI().ensureThat(ViewLocator.forName(
"MensajesProblemas").isClosed());
}
```

Métodos Test

Son las verificaciones que se realizan en cada paso del diseño. Se compara un valor esperado contra un valor actual. Los tests se pueden clasificar en dos tipos:

Según la capa:

Sobre la interfaz usuario

Un ejemplo. Cuando en un paso de usuario en un beforeClass() se realice la acción de salvar un comprobante y se solicite verificar que el estado del mismo en la aplicación figure como “Inicial”.

Ejemplo:

```
public void testUI() {
getUI().assertThat(new NamedWidgetLocator(“estado”).
hasText(“Inicial”));
}
```

Sobre la base de datos

Ejemplo:

```
public void testBD() throws Throwable {
try {
assertEquals(true, object.exists());
} catch(Throwable t) {
throw new BD_TA_Exception(t.getMessage());
}
}
```

Según la multiplicidad:

Simples: son los métodos tests que no devuelven nada (void)

Anidados: son los métodos tests que devuelven un objeto Test. Son una estrategia que permite agrupar tests con el objetivo de aumentar el reúso. En un paso que necesite verificar múltiples valores esperados. Es necesario que el método sea público, devuelva un objeto Test y su nombre comience con “suite”.

Ejemplo:

```
public Test suiteAnidado() {
return new SetupDecoratedTestSuite(Verificaciones.class);
}
```

Test vacíos

Existen pasos en los cuales no existe la necesidad de verificar nada, solo son pasos del usuario. JUnit3, en la clase TestSuite (constructor), cuando empieza a organizar los "Test", verifica si el paso contiene o no métodos tests, en el caso de no existir, agrega un test al paso con el nombre "warning" que hace que falle el test:

```
if (Tests.size() == 0) addTest(warning("No tests found in "+
theClass.getName()));
```

Por esta razón, por convención en nuestro proyecto, adoptamos lo siguiente: **"Todo paso en que no haya que verificar nada, implementa en el mismo, un método cuya signatura sea "public void testVacio()"** y su cuerpo este vacío; es decir, no contenga ninguna sentencia Java.

```
public void testVacio() { }
```

Sub pasos:

Se implementan como clases internas que son una herramienta que provee java para definir clases dentro de otras clases, a veces útiles para representar situaciones especiales. Resultan prácticas para reducir la cantidad de archivos fuentes (.java) y reutilización de código aspirando a un fácil mantenimiento de los tests. Para una mayor legibilidad del código el nivel de jerarquía no debe superar los dos niveles, ya que resulta complicado entender un diseño en papel en el que cada paso refiere a otro varias veces antes de averiguar qué acciones y verificaciones se espera realizar efectivamente. Por este motivo se pide forzar los diseños y las implementaciones a que limiten la cantidad de veces que se utiliza la herencia. En concreto se espera que ninguna superclase extienda de un paso, se trate de inner classes o no.

3.2.3 Ciclo de vida del desarrollo de una prueba automática

El ciclo de vida de una Prueba es la secuencia de tareas en un orden específico para desarrollar la misma. Se compone de las siguientes tareas:

1. Diseño del caso de prueba automático (CDPA) con las herramientas Excel / TestLink.
2. Ejecutar la aplicación mediante un launcher “Eclipse Application” en modo Record.
3. Grabar interacciones del usuario con la aplicación.
4. Generar código en forma automática desde la vista “Recorder”.
5. Limpiar código ajustando a la modalidad de trabajo del proyecto Testing_automático.
6. Incorporación de validaciones test.

Diseño del caso de Prueba:

Estos diseños los realiza el Área de Análisis y es el insumo o sea el requerimiento para desarrollar un circuito, un caso de prueba diseñado es un circuito a desarrollar.

En la figura 9, se muestra el diseño de un caso de prueba que posee un nombre y un conjunto de pasos y cómo puede apreciarse en la imagen, en cada paso se describen las acciones del usuario y las verificaciones requeridas que se deben desarrollar, esto se traduce en nuestro proyecto a un Circuito que tiene un conjunto de pasos en donde cada paso es una clase con un método beforeClass (acciones de usuario descritas en el diseño), unas verificaciones (test) y posiblemente posea un método afterClass (acciones de usuario que se realizan luego de los test al final de un paso).

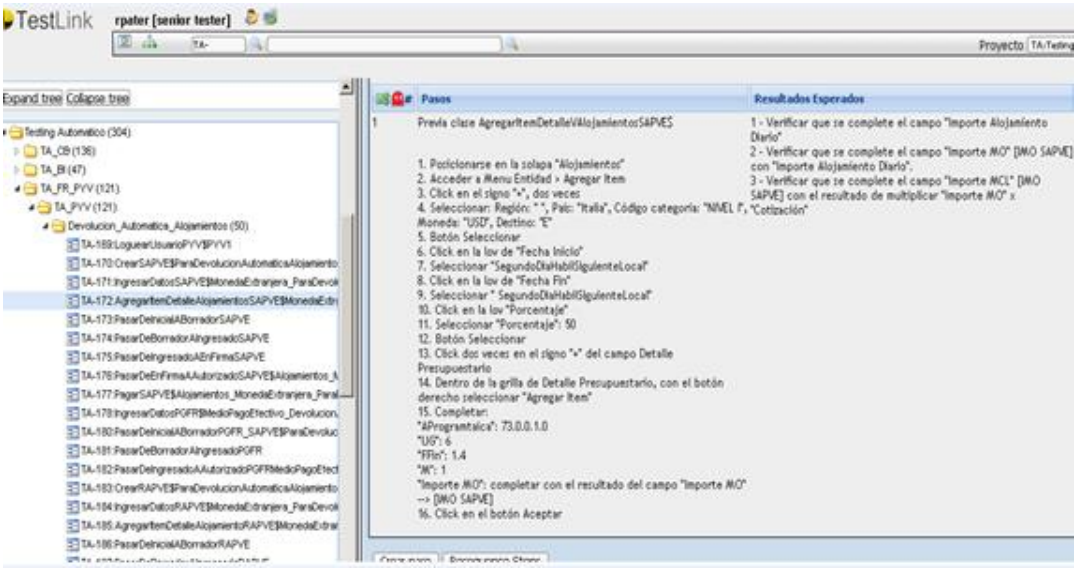

Figura 9. Ejemplo de Diseño caso de prueba

Uso de la Grabadora de la herramienta WindowTester

Para utilizar la misma se crea un launcher (lanzador) “Eclipse Application” en modo Record, que contenga los plugins de record de windowTester, se ejecuta y se va interactuando con la aplicación simulando las acciones del usuario, de acuerdo a la descripción de cada paso diseñado en el caso de uso, el objetivo es que la grabadora reconozca los widgets con los que se interactúa tales como cajas de texto, botones, etc. La consola de grabación va capturando dichos movimientos y registrándolos, una vez terminada las acciones a realizar, la consola genera una clase con dichos movimientos, luego se traduce el código generado a la modalidad del proyecto, utilizando las clases llamadas Manejadores. Estas clases fueron creadas en el proyecto para encapsular los locator y conditions, en ellas están los métodos más utilizados para acceder a los widgets de la UI de acuerdo al tipo de acceso, ejemplo de estas clases: ManejoBotonWT, ManejoMenuWT, ManejoTextoWT, ManejoVentanaWT, etc. La idea de tener estas clases es centralizar y reusar los métodos definidos en ellas cada vez que aparezca una acción de alguna clase en particular. A medida que surjan acciones distintas se van a agregar métodos nuevos en dichas clases.

Una vez realizado esto se crea la clase con el nombre del paso a desarrollar, todas estas acciones van a formar parte del beforeClass (pasos de usuario) y luego del beforeClass estarán los test assertions (verificaciones) requeridas en el diseño. En algunos casos es necesario también que esté el método afterClass que será el último método del paso.

En la figura 10 se puede observar la interfaz de la grabadora y en la figura 11 el diálogo para guardar la clase generada por la grabadora luego de interactuar con la misma.

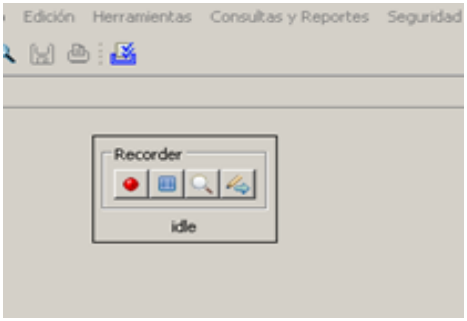

Figura 10. Interfaz de grabadora

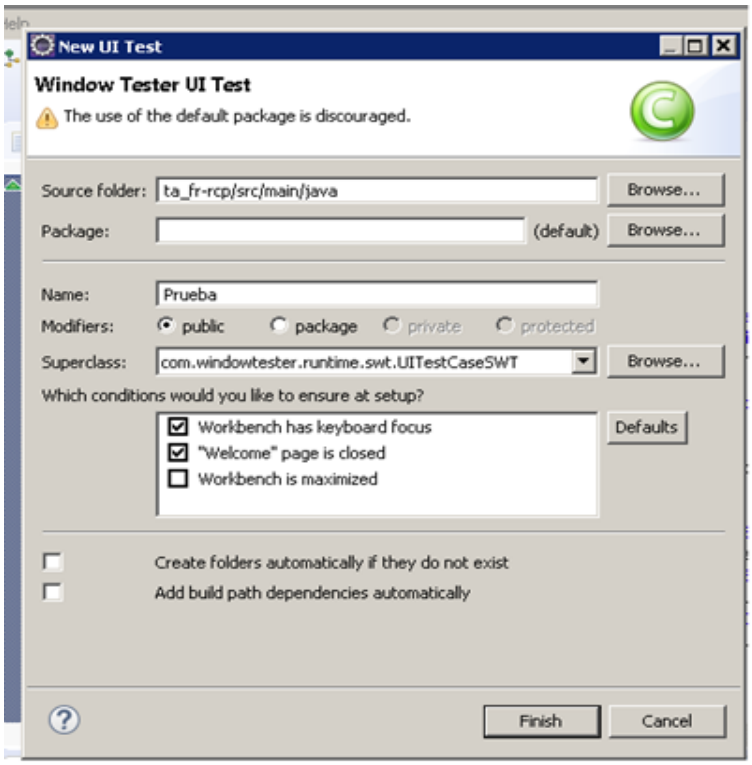

Figura 11. Clase a guardar luego de registrar los movimientos de usuario en un paso

Tal como se muestra en la Figura 12 y Figura 13, pueden observarse ejemplos de clases Manejos.

```
54
55 public class ManejoBotonWT {
56
57     // Button
58
59     public static void clickButton(IUIContext ui, String buttonText) throws Exception {
60         ui.click(new ButtonLocator(buttonText));
61     }
62
63     public static void clickButtonAceptarConInternacionalizacion(IUIContext ui) throws Exception {
64         ManejoBotonWT.clickButton(ui, Messages.NLS_ACEPTAR);
65     }
66
67     public static void clickButtonAceptarSinInternacionalizacion(IUIContext ui) throws Exception {
68         ManejoBotonWT.clickButton(ui, "Aceptar");
69     }
70
71     public static void clickButtonCancelarConInternacionalizacion(IUIContext ui) throws Exception
72         ManejoBotonWT.clickButton(ui, Messages.NLS_CANCELAR);
73     }
74
75     public static void clickButtonCancelarSinInternacionalizacion(IUIContext ui) throws Exception
76         ManejoBotonWT.clickButton(ui, "Cancelar");
77     }
```

Figura 12. Ejemplo de clases Manejos. Manejo Botón.

```
public class ManejoVentanaWT {

    public static void clickShell(IUIContext ui, String title) throws Exception {
        ui.click(new ShellLocator(title));
    }

    public static void maximizeWorkbench(IUIContext ui) throws Exception {
        ui.ensureThat(new WorkbenchLocator().isMaximized());
    }

    public static void maximizeShell(IUIContext ui) throws Exception {
        ui.keyClick(WT.ALT, ' ');
        ui.keyClick('x');
    }

    public static void showShell(IUIContext ui, String title) {
        ui.wait(new ShellShowingCondition(title));
    }

    public static void showShell(IUIContext ui, String title, long timeout) {
        ui.wait(new ShellShowingCondition(title), timeout);
    }

    public static void disposeShell(IUIContext ui, String title) {
        ui.wait(new ShellDisposedCondition(title));
    }
```

Figura 13. Ejemplo de clases Manejos. Manejo Ventana.

3.2.4 Metodología de trabajo para el desarrollo y ejecución de una prueba

Cuando se recibe un diseño de caso de Prueba para automatizar un circuito de un negocio, tal como se explicó anteriormente, se va desarrollando cada paso de acuerdo a la especificación del diseño.

Usando la grabadora de windowTester, para cada paso se genera la clase y se desarrollan el método beforeClass (pasos de usuario), los test o verificaciones requeridas en el caso de prueba, y si fuera necesario el método afterClass.

Un caso de prueba se representa en un circuito de varios pasos, una vez desarrollado el mismo, se ejecuta desde Eclipse desde un launcher (lanzador) de ejecución de JUnit Plug-in-Test, ya que un circuito se va a ejecutar como una clase JUnit3. Este launcher va a lanzar por un hilo la aplicación y por otro hilo nuestros test, para que interactúe sobre la misma emulando un usuario y realizando las pruebas. El launcher tendrá en los argumentos de la VM (Java Virtual Machine) unos parámetros, que representan la URL de la aplicación a testear y la base de datos donde se encuentra el modelo de la aplicación llevada a una base de datos relacional. A medida que son ejecutados los circuitos en distintas etapas de regresión, es esperable que se detecten fallos en circuitos que funcionaban correctamente en versiones anteriores, estos fallos o errores pueden ocurrir por diferentes motivos.

Fallo por error de windowTester

Este fallo suele ocurrir por ejemplo cuando un paso desarrollado de un circuito falla al no encontrar un widget, la causa podría ser un cambio de nombre del mismo, de una versión de la aplicación a otra, un cambio del tipo de elemento al cual se hacía referencia, ejemplo: en una versión había un button y en la nueva se cambió por un label, o un cambio en el comportamiento del paso por alguna modificación de la funcionalidad y arquitectura. Estos errores son analizados junto al equipo de testing manual y cuando se define si es correcto dicho comportamiento nuevo, se ajusta el diseño y se ajusta el código de nuestro proyecto para que el circuito funcione correctamente de acuerdo al nuevo

esquema. En la figura 14 se puede observar un fallo de este tipo, en donde no se encuentra un widget.

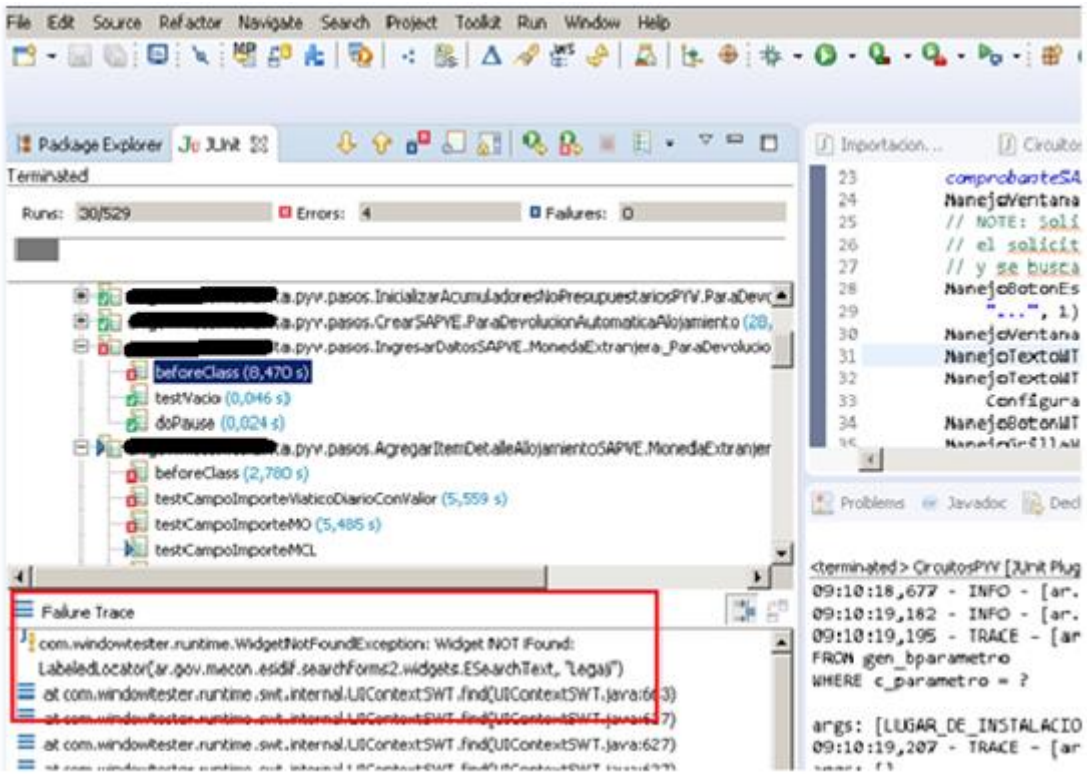

Figura 14. Ejemplo de Falla en pasos de usuario (beforeClass)

Falla por error inesperado

Ocurre cuando en medio del circuito aparece un fallo inesperado, se da aviso al grupo de testing manual (control de calidad) para que verifique manualmente si realizando las mismas acciones que realiza el circuito se comporta de la misma manera. Por lo general estos son bugs bloqueantes de la aplicación.

Fallo de test

Ocurre cuando falla una verificación (test) en la cual el valor que se espera difiere del valor actual, se verifica con testing manual para saber si cambió alguna regla de negocio o si es una falla del sistema. Según el caso se ajusta el test.

En los casos en que no sea un problema del proyecto de Testing_automático y fuese un problema de la versión de la aplicación, los tester manuales levantan un bug con un número y la descripción del problema, el mismo se anota en el código el proyecto Testing_automático con la anotación java @bug la cual tiene los campos número de bug, la descripción y desarrollador que lo identificó, cuando es ejecutado el circuito y falla el test en cuestión, se sabe cuál es la causa del fallo y se deja seguir ejecutando el circuito. En cambio, si el bug es bloqueante se dejará el circuito con dicha anotación hasta que el mismo sea resuelto. Cuando el área de desarrollo resuelve dicho bug, se prueba en la nueva versión en donde está corregido y si el funcionamiento es correcto, se ajusta el código del proyecto eliminando la anotación del mismo. En la figura 15 se puede ver un ejemplo de falla de test.

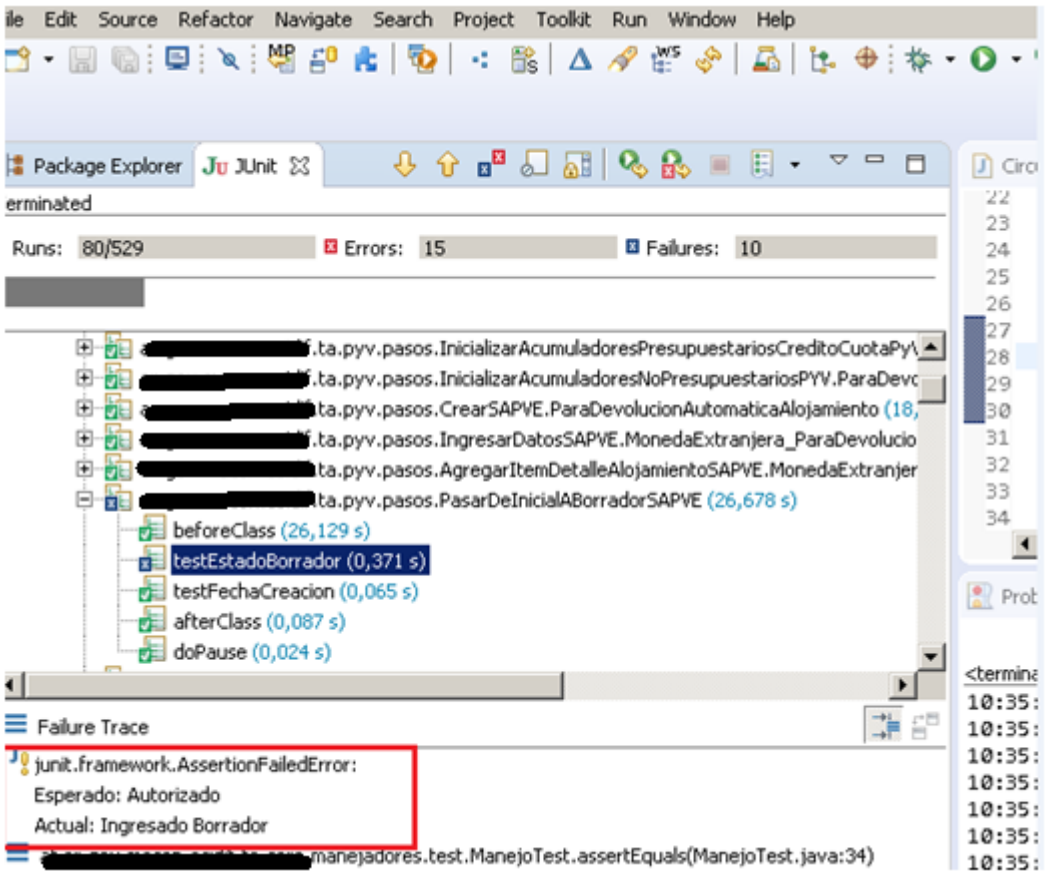

Figura 15. Ejemplo de fallo de test

Precondiciones

Para poder realizar la ejecución de los circuitos del proyecto de Testing_automático en un entorno o ambiente de prueba, es necesario que las bases de datos de estos entornos donde impactan las ejecuciones contengan una serie de datos esenciales para la ejecución de los mismos. Existen circuitos específicos en el proyecto que realizan la carga de estos datos. Estos circuitos son llamados PostRefresh y PreEjecucion.

Los circuitos PostRefresh deben ser ejecutados sobre una base de datos, cada vez que la misma, en donde se requieran las ejecuciones de los circuitos del proyecto, venga de un refresh (borrado de datos). Es muy importante dicha ejecución, ya que además de los datos necesarios de los negocios, también se requiere que existan los usuarios que se utilizan en el proyecto con sus correspondientes capacidades, ya que de lo contrario dichos circuitos no funcionarán, para esto existen circuitos de Creación de Usuarios por negocio.

Los circuitos de PreEjecucion por negocio deben ejecutarse antes que los circuitos del negocio sean ejecutados, ya que estos crean por aplicación datos a ser utilizados por los mismos.

Descripción de algunos aspectos de la Arquitectura

La funcionalidad básica del proyecto de Testing_automático es brindada por un componente principal llamado Core, este componente contiene las clases fundamentales para la ejecución de un circuito. Esto incluye la gestión de los parámetros de ejecución, manejo de la Base de Datos, gestión de la API del sistema operativo, acciones básicas y la lógica de logging en base de datos.

Cada componente negocio contiene una lógica del mismo, los metacircuitos, los circuitos y pasos y un conjunto de script, además de los datos propios del negocio que sirven para construir los datos a popular en cada circuito.

3.3 Pruebas de regresión aplicadas al Proyecto a testear

En el proyecto de Testing_automático existen varios negocios (módulos) automatizados que se corresponden con los módulos de la aplicación a testear, los cuales tienen varios circuitos, cada circuito contiene una cantidad importante de pasos con sus respectivas verificaciones, tanto de la interfaz gráfica, como de la base de datos, como se explicó anteriormente, los circuitos se corresponden con casos de usos diseñados por el Área de Análisis. Las pruebas de regresión son pruebas que se realizan para asegurarnos de que algún cambio introducido por pequeño que sea o una nueva funcionalidad en el sistema, no afecte el código que antes funcionaba. Al realizar estas pruebas se comprueba que los cambios se comporten como es esperado y no estén rompiendo el código que antes funcionaba.

La ejecución de los circuitos de los negocios del proyecto de Testing_automático sobre cada nueva versión de la aplicación testeada, cumple con el objetivo planteado en el testing, ya que al correr la batería de circuitos automáticos de los negocios se abarca una cobertura muy amplia de pruebas y se disminuyen los tiempos de testeo de forma significativa, reduciendo los tiempos de despliegue.

3.4 Simulación de funcionamiento de las pruebas.

En el video adjunto se visualiza la ejecución de un circuito que va creando y transicionando comprobantes utilizados en la gestión de un Banco: Solicitud de Alta, Solicitud de Modificación, Solicitud de Baja y Solicitud de Rehabilitación. Cuando comienza el circuito hay un logueo al Sistema y a posterior se da de alta un Banco por medio del comprobante de Solicitud de Alta. En dicho video, se puede apreciar cómo se van completando los campos del comprobante con datos artificiales de la cabecera, mostrado en la Figura 16, como ser: personería, denominación completa, denominación abreviada, contactos, etc. Domicilios como se muestra en la Figura 17, etc. simulando la entrada de datos de un usuario. A medida que se va transicionando el comprobante pasando de un estado a otro, se van realizando verificaciones (test). Ejemplo: test que verifica

que el estado del comprobante que figura en la interfaz gráfica, sea igual al estado que se espera, test que verifican que los datos ingresados en el comprobante, sean los esperados, que ciertas solapas del comprobante están habilitadas, etc. Luego del alta del Banco, se crea una Solicitud de Modificación, en donde se modifican ciertos datos del Banco creado. En las transiciones del comprobante se hacen verificaciones que comprueban dichos cambios. A continuación, se da de baja el Banco mediante el comprobante Solicitud de Baja verificando el estado del comprobante en sus transiciones, por último, se rehabilita el Banco con el comprobante de Solicitud de Rehabilitación.

Al finalizar la ejecución del circuito, se visualiza el resultado de los pasos y sus verificaciones. En el ejemplo de la Figura 24, se pueden observar para un paso, que el método Before de dicho paso (tiene color verde), lo que indica que no hubo ningún problema en los pasos de usuario (se encontraron todos los widgets con los que se interactúa ejemplo: botones, cajas de texto, etc.) y también se puede ver que todos los test (verificaciones) ejecutados para este paso también figuran en verde, esto indica que cada verificación arrojó resultado esperado.

Enlace al video de referencia

https://www.dropbox.com/s/yxmlzwaa06m2yp4/Video-Circuito.mp4?dl=0

Solicitud de Alta

En la figura 16 y 17 se puede apreciar un comprobante de Solicitud de Alta de un Banco con datos ficticios ingresados automáticamente por el sistema simulando ser cargados por un usuario

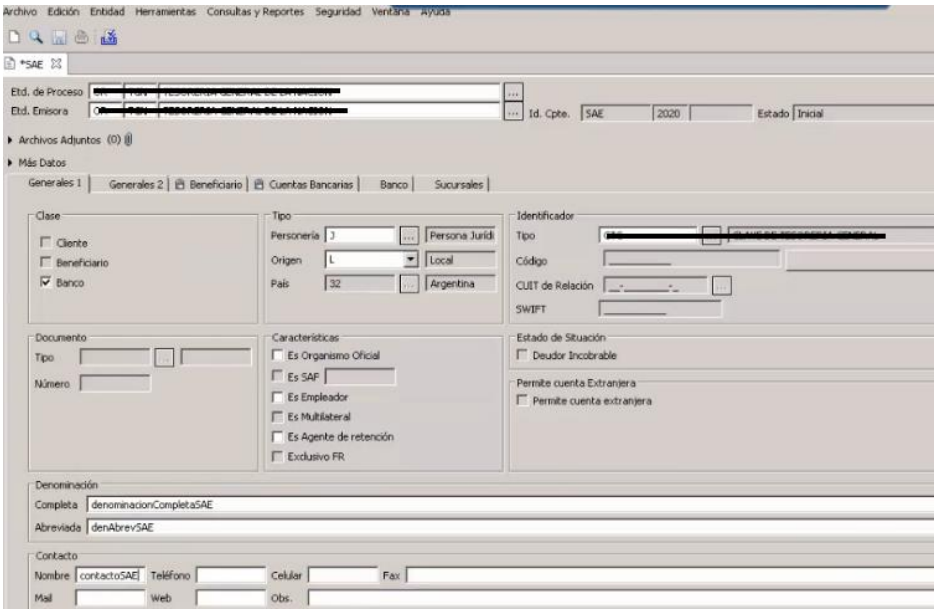

Figura 16. Comprobante Solicitud Alta Datos Cabecera

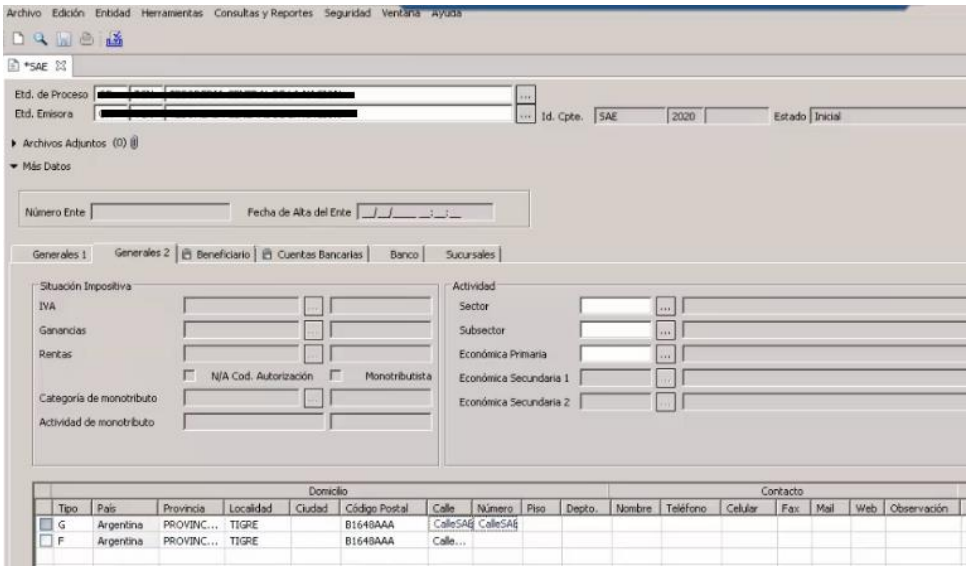

Figura 17. Comprobante Solicitud Alta Domicilios

Solicitud de Modificación

En la figura 18, 19 y 20 se puede apreciar un comprobante de Solicitud de Modificación que busca el comprobante Ingresado anteriormente y lo modifica.

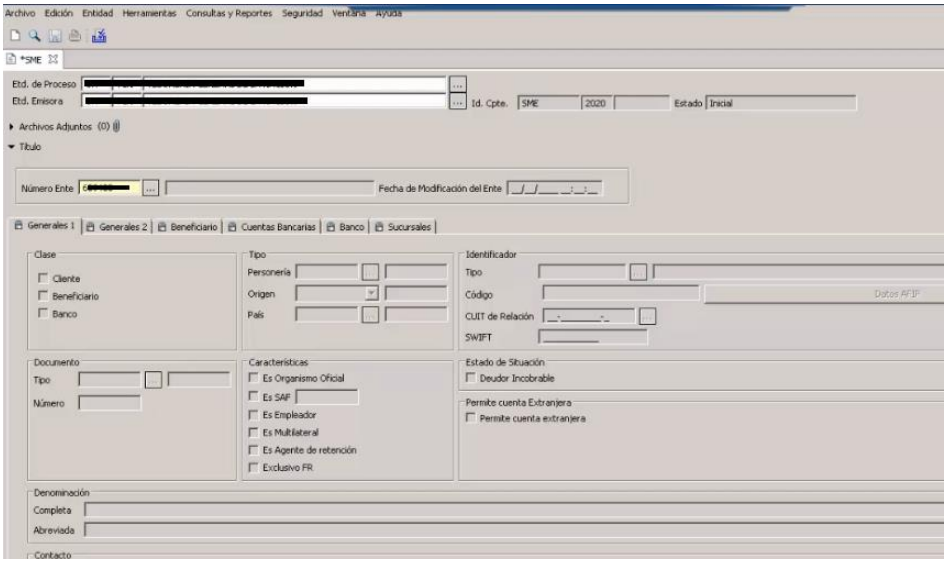

Figura 18. Comprobante Solicitud Modificación

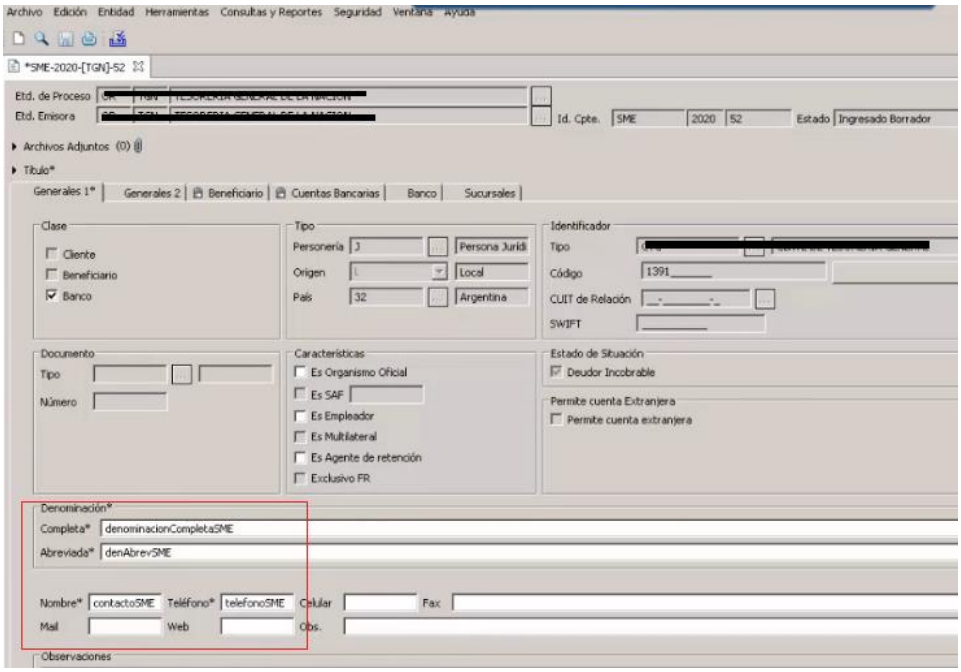

Figura 19. Comprobante de Solicitud Modificación. Se modifican datos generales

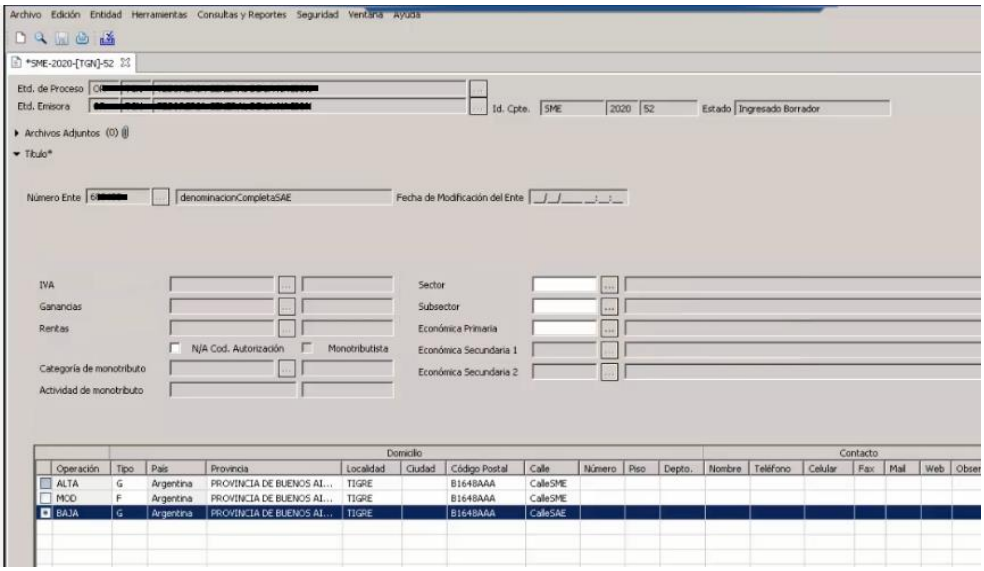

Figura 20. Comprobante Solicitud de Modificación. Se modifican domicilios

Solicitud de Baja

En la figura 21 se puede observar un comprobante de Solicitud de Baja en donde se le da de baja a un Banco.

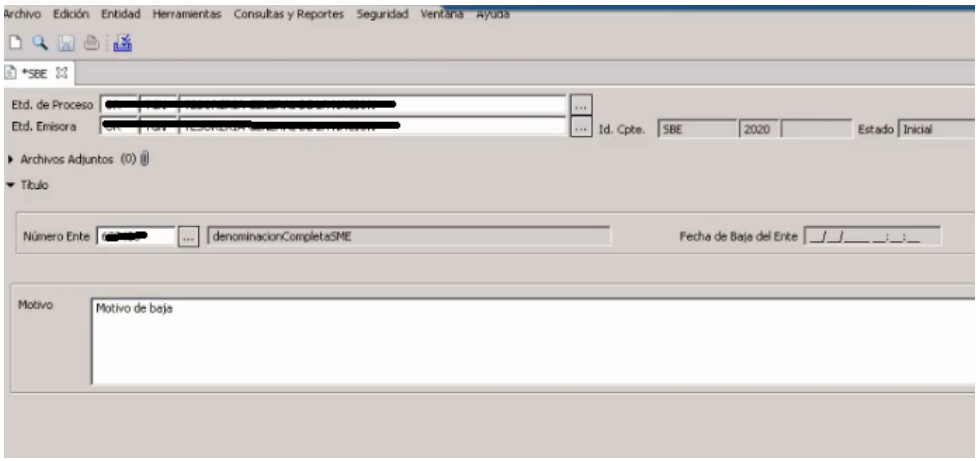

Figura 21. Comprobante Solicitud de Baja

Solicitud de Rehabilitación

En la figura 22 y 23 se puede apreciar un comprobante de Solicitud de Rehabilitación por medio del cual se rehabilita un Banco.

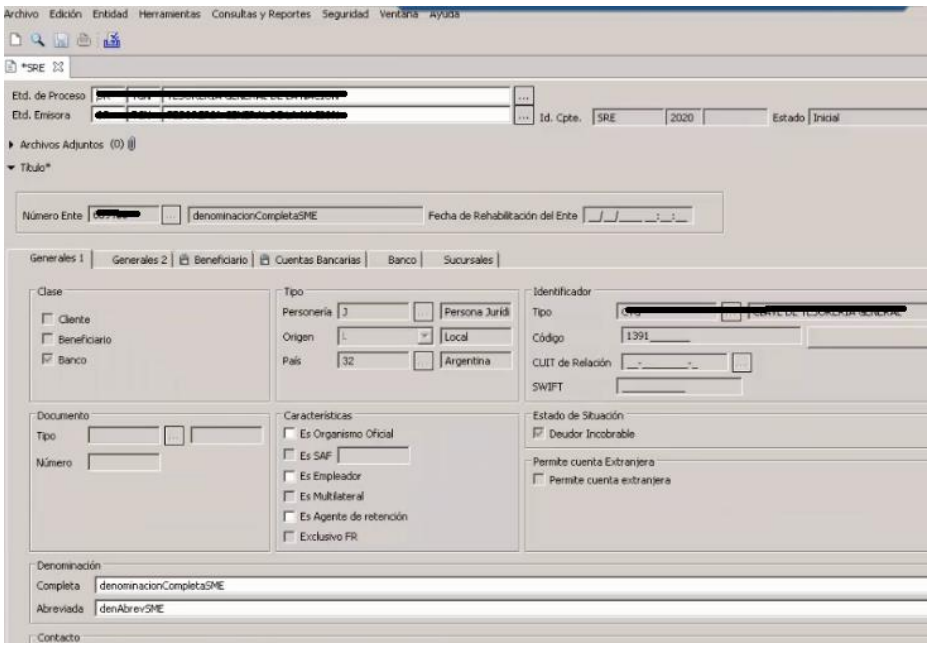

Figura 22. Comprobante Solicitud de Rehabilitación

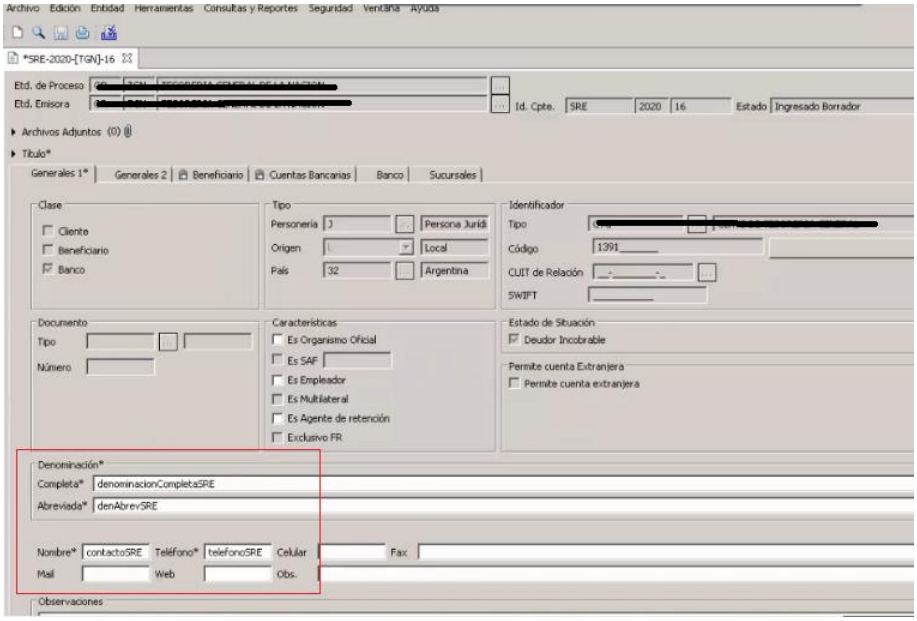

Figura 23. Comprobante Solicitud de Rehabilitación. Se modifican Datos generales

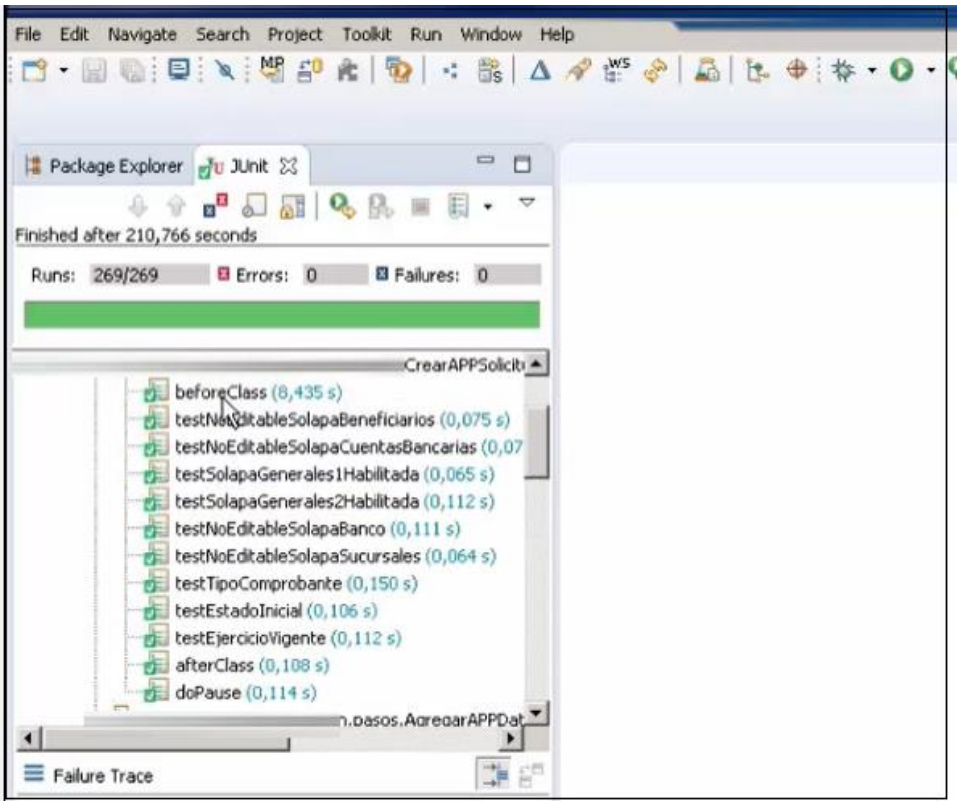

Figura 24. Verificaciones que funcionaron correctamente

3.5 Conclusiones

El proyecto de Testing_automático presentado en este capítulo, destaca las ventajas de incorporar procesos automatizados de control a una empresa dedicada a la gestión financiera los cuales eran realizados en sus inicios en forma manual.

Los test funcionales automatizados permitieron reducir grandes esfuerzos de recursos humanos y tiempos de despliegue, optimizar la calidad, y mitigar errores.

4. INVESTIGACIÓN DE DISTINTAS HERRAMIENTAS DE AUTOMATIZACIÓN

En el presente capítulo, se describen dos herramientas que son las más utilizadas en la automatización de casos de prueba para páginas web. Selenium es una herramienta que fue líder durante una década y Cypress es una herramienta surgida recientemente que está teniendo una gran aceptación en el desarrollo de pruebas automatizadas para aplicaciones web.

Abordamos el análisis de las mencionadas herramientas, ya que es muy posible que algunos módulos del sistema que se testea, migren a este tipo de tecnología o que en nuestra empresa surjan nuevos proyectos web que requieran ser testeados.

4.1 Selenium

Selenium es un conjunto de diferentes herramientas de software que permiten juntas la automatización de pruebas para aplicaciones basadas en la web. Las herramientas de Selenium realizan operaciones altamente flexibles, lo que habilita muchas opciones para localizar elementos de la interfaz de usuario en el navegador y comparar los resultados esperados de las pruebas respecto del comportamiento real de la aplicación. Una de las ventajas que presenta Selenium es la versatilidad del soporte en diferentes entornos, permitiendo así la ejecución de pruebas en múltiples plataformas de navegador. [Selenium.org, 2017]

Originalmente, Selenium fue desarrollado por Jason Huggins en 2004 para un proyecto privado aquí él desarrolla una librería en JavaScript que se convirtió en el core de Selenium, misma que subyace a todas las funciones de Selenium Remote Control (RC) y Selenium IDE. Posteriormente Huggins se une con Simon Stewart de Google que había desarrollado un WebDriver también para automatización de pruebas. En 2008 ambos proyectos emergen como uno brindando una herramienta más potente. [Selenium.org, 2017]

Selenium es un software de código abierto que se distribuye bajo la licencia apache 2.0 por lo que puede ser descargado directamente desde el sitio oficial y ser usado sin costo. Así mismo, la empresa se encuentra desarrollando otros

proyectos como Selenium Grid, que permite hacer pruebas de concurrencia múltiple o Flash Selenium para experimentar programas escritos en Adobe Flex o Selenium Silverlight. [Selenium.org, 2017]

Componentes de la suite Selenium

Selenium no es una única herramienta, sino que comprende una suite de 4 herramientas de software, donde cada una realiza un rol específico; estas son: Selenium IDE, Selenium RC, Selenium WebDriver, Selenium Grid.

Selenium WebDriver

Es la nueva herramienta de automatización de Selenium que incluye nuevas características tales como una API más cohesiva y orientada a objetos. Es compatible con WebDriver y ejecuta la interfaz Selenium RC para compatibilidad con versiones anteriores. Selenium WebDriver acepta comandos que se pueden enviar con Selenese (lenguaje específico de dominio para escribir pruebas que se pueden ejecutar en un amplio número de lenguajes de programación) o con el API de cliente hacia el navegador de preferencia. [Selenium.org, 2017]

Selenium WebDriver funciona realizando llamadas directas al navegador haciendo uso del soporte nativo de cada navegador para la automatización. Dado la variedad de navegadores web a disposición y varios lenguajes de programación populares, para desarrollar y correr las pruebas se necesita una especificación común, la misma que es proporcionada por la API de WebDriver. Cada navegador tiene que implementar esta API llamada Remote WebDriver o Remote WebDriver Server. En un nivel superior, la arquitectura Selenium WebDriver se parece a lo mostrado en la figura 24. [Selenium.org, 2017]

Figura 24. Arquitectura de Selenium WebDriver.

El proceso de pruebas con SeleniumWebDriver explicado resumidamente es el siguiente: el enlace de idioma envía los comandos a través del controlador común que proporciona la API, en el otro extremo existe también un controlador que escuchará estos comandos y los ejecutará en el navegador seleccionado utilizando el WebDriver remoto; finalmente este devolverá el resultado a través de API en el código de enlace. Hay que tomar en cuenta que cualesquiera sean los comandos emitidos en el código, estos se interpretarán en métodos de servicio web, con el protocolo JSON y luego el controlador remoto recibirá la solicitud HTTP, los ejecutará en el navegador y luego devolverá la respuesta. [Selenium.org, 2017]

Para Selenium WebDriver todos los elementos de una página web como los campos de texto, botones, links, imágenes son WebElements. Existen distintos tipos de localizadores para ubicar un elemento y realizar alguna acción sobre él. De esta misma forma puede también acceder a atributos visibles como el texto (mediante el método getText()) y a atributos no visibles desde la interfaz de usuario a través de comandos como getAttribute(“nombre del atributo”). Para comunicarle a Selenium cómo encontrar un elemento utilizamos el comando: driver. findElement() que es el método encargado de devolver el WebElement y que recibe como parámetros un localizador (By).

En la figura 25 se muestran cuáles son los distintos tipos de localizadores usados en Selenium WebDriver.

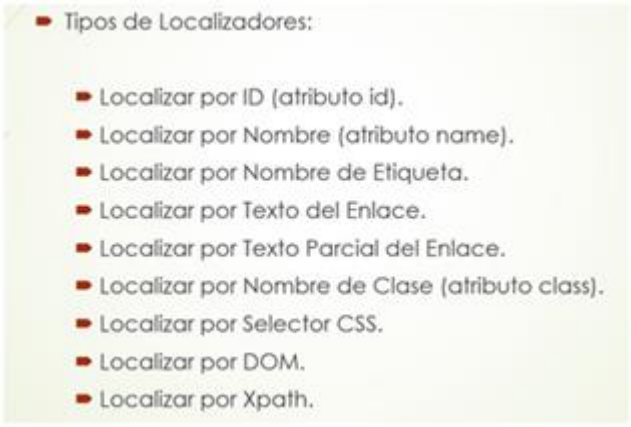

Figura 25. Localizadores de Selenium WebDriver

Un aspecto importante durante la creación de los tests automáticos con Selenium WebDriver es el tiempo de carga de la página o de los WebElements. Si no se tienen en cuenta estos tiempos, y no se ponen las protecciones adecuadas frente a ellos, es probable que nuestros tests fallen debido a un error en la programación de la prueba.

Los métodos para la localización de WebElements lanzan una excepción cuando no encuentran ninguno que coincida con el patrón o mecanismo de búsqueda. Para evitar que esta situación se produzca por el hecho de que no se haya cargado el WebElement en el momento en el que el script intenta localizarlo, Selenium WebDriver ofrece unos mecanismos de espera. Timeouts.

Se adjunta en Anexo I una descripción detallada de la herramienta Selenium.

4.2 Cypress

Cypress es una herramienta de testeo front end de código abierto de última generación creada para la web moderna. Este framework todo en uno, incluye librerías de aserciones, mocks y pruebas End To End (de principio a fin) automáticas sin utilizar Selenium.

Cypress cuenta con una arquitectura construida desde cero, que ejecuta los comandos en el mismo ciclo de ejecución. [Cypress doc]

Figura 26. Cypress herramienta todo en uno

Cypress es una herramienta utilizada especialmente para manejar frameworks de javaScript modernos como por ejemplo React, Angular, Vue, etc. Pero funciona perfectamente en páginas o aplicaciones renderizadas en servidor.

Cypress ejecuta pruebas en un ejecutor o corredor de pruebas, el cual permite ver los comandos que se ejecutan mientras se muestra la aplicación que se está probando. [Cypress doc]

El registro de comandos (Command Log) del Corredor de pruebas, es una representación visual del conjunto de pruebas.

Cada comando o aserción, cuando se coloca sobre él, restaura la aplicación bajo prueba al estado en que se encontraba cuando se ejecutó dicho comando o aserción, esto permite viajar a estados anteriores de la aplicación bajo prueba.

Comandos en Cypress

Una cadena de comandos en Cypress comienza con cy.[command] donde command puede ser concatenado por otros comandos.

cy.visit()

El comando cy.visit() se utiliza para dirigirnos a una URL.

Cuando Cypress consulta un elemento en el DOM, reintenta automáticamente la consulta hasta que el elemento se encuentra o se alcanza un tiempo de espera establecido esto hace que sea muy robusto y no lo afecten muchos problemas que tienen otras herramientas de pruebas. Estos problemas pueden producirse por ejemplo cuando el DOM no se ha cargado completamente, no se ha completado una animación, etc. Antes había que escribir código personalizado contra cada uno de estos problemas como ser esperas, reintentos condicionales y comprobaciones nulas que ensucian las pruebas. Cypress evita estos problemas. [Cypress doc]

cy.get()

Este comando se utiliza para obtener elementos del DOM.

.click()

Se utiliza para hacer click en un elemento del DOM

.type()

Se utiliza para escribir un texto en un elemento del DOM

Cypress se basa en muchas de las mejores bibliotecas de pruebas de código abierto. Cypress ha adoptado la sintaxis de Mocha, que encaja perfectamente

tanto en la integración como en las pruebas unitarias. Todas las pruebas que se escriben se basan en el arnés fundamental que proporciona Mocha. [Mocha]

Así como Mocha proporciona un marco para las estructuras de pruebas, Chain brinda a Cypress la capacidad de escribir Aserciones (afirmaciones) fácilmente. Cypress extiende esto, corrige varios errores comunes y envuelve el DSL de Chain usando sujetos y el comando .should() . [Chain]

Se adjunta en Anexo II una descripción detallada de la herramienta Cypress.

4.3 Conclusiones del análisis de las herramientas

En el proyecto Testing_automático se utilizó la herramienta WindowTester ya que es una herramienta que permite crear test automáticos para aplicaciones gráficas construidas con swing o con SWT (Standard Widget Toolkit). Dado que la aplicación principal de la empresa testeada utiliza como framework para su diseño y desarrollo a la interfaz gráfica SWT, conjunto de componentes para construir interfaces gráficas en Java, WindowTester nos pareció en su momento la herramienta más apropiada. En cambio, las herramientas investigadas Selenium y Cypress, son utilizadas para testear aplicaciones web.

La investigación de dichas herramientas se hizo con el propósito de ir analizando y evaluando las mismas para una posible migración de ciertos módulos de la aplicación testeada a aplicaciones web, como así también por la posibilidad de que se desarrollen aplicaciones web que requieran ser testeados.

Cypress es una herramienta basada puramente en javaScript opera directamente sobre el navegador y posee una técnica única de manipulación del DOM, Cypress difiere fundamental y arquitectónicamente en comparación con Selenium.

En un principio Cypress solo admitía pruebas en el navegador Chrome, pero las actualizaciones posteriores brindan soporte para los navegadores Firefox y Edge.

Cypress proporciona un corredor de prueba en el que ejecuta todos los comandos. Captura instantánea en el momento de la ejecución de la prueba. Esto permite que al pasar el cursor sobre un comando en el Registro de comandos se puede ver lo que sucedió en ese paso en particular. No es necesario agregar comandos de espera explícitos o implícitos en los scripts de prueba, a diferencia de Selenium, Cypress espera automáticamente los comandos y las afirmaciones. Cypress ejecuta en tiempo real, los comandos que escriben los desarrolladores, proporcionando información visual mientras se ejecutan. Cypress tiene una excelente documentación.

Como desventajas se puede decir que Cypress solo admite JavaScript para crear casos de prueba, no puede manejar dos navegadores al mismo tiempo, no proporciona soporte para pestañas múltiples y posee limitaciones en el servicio gratuito.

Selenium ha sido líder en la automatización web durante una década. Permite automatizar casos de prueba para el navegador deseado mediante el uso de la biblioteca Selenium WebDriver. Es necesario descargar el controlador específico del navegador para poder automatizar los casos de prueba. A diferencia de Cypress admite flexibilidad en el uso del lenguaje para los desarrolladores tales como Ruby, Python, Java, etc. Permite utilizar varios navegadores al mismo tiempo. Como desventaja podemos decir que Selenium no posee comando incorporado para la generación automática de resultados de prueba, soporte limitado para imágenes. Construir los casos de pruebas en Selenium demanda mucho tiempo, configurar el entorno es difícil en comparación a Cypress.

5. Conclusiones

5.1 Conclusiones

En el presente trabajo, hemos investigado sobre la automatización de los test funcionales que realiza el sector de control de calidad para las pruebas de regresión.

Indagamos el potencial de las pruebas como herramienta eficaz en los procesos de testing.

Implementamos Testing Automático, un proceso de testing enfocado en la automatización de las pruebas de regresión. Las mismas, tienen una gran importancia en el desarrollo de software, sirven para determinar qué nuevas funcionalidades de una aplicación no afecten a las ya existentes, para detectar fallas en forma temprana y para reducir tiempo y esfuerzo. Aseguran calidad del producto y disminuyen los tiempos de despliegue.

El tiempo que implica el desarrollo de las pruebas automáticas se ve recompensado por la cantidad de veces que se ejecutan las mismas.

El proyecto Testing Automático que desarrollamos fue creciendo y escalando en el tiempo e incorporando la automatización de nuevos negocios. Sigue teniendo vigencia y sigue siendo esencial para el despliegue de cada versión de la aplicación testeada, dado que la misma es muy extensa y está compuesta por una gran cantidad de negocios.

El proyecto brinda beneficios, ahorra recursos humanos, tiempo de despliegue, ejecuta una batería de circuitos que abarcan una gran cobertura en las pruebas de regresión en casi todos los negocios de la aplicación. Y las mismas son repetibles y reducen las fallas humanas.

Por otro lado, no podemos dejar de mencionar que la herramienta utilizada WindowTester, actualmente no posee soporte y nos limita a usar determinadas versiones de otras herramientas como ser el Eclipse y el JUnit utilizadas en el proyecto. Por esta razón y a la espera de que la aplicación testeada vaya de

a poco migrando a nuevas tecnologías web, es que se han investigado como alternativas las herramientas Selenium y Cypress, descritas en el presente trabajo.

5.2 Trabajos futuros

Ante la posibilidad de que algunos módulos de la aplicación testeada en un futuro cercano vayan migrando a tecnologías web, o se desarrollen en la empresa nuevos aplicaciones de estas características, se contempla continuar profundizando la investigación de herramientas para el testeo de las mismas.

6. Bibliografía

[Ceballos Guerrero, 2011] Rafael Ceballos Guerrero, “Técnicas automáticas para la diagnosis de errores en software diseñado por contrato” [en línea]. [Consulta: 02 de mayo 2020]. Disponible en: <https://documat.unirioja.es/servlet/tesis?codigo=24794>.

[Chain] DocumentaciónOficial [en línea][Consulta: 06 de Octubre 2020]. Disponible en:< https://www.chaijs.com/>

[CMMI02]. “Capability Maturity Model Integration (CMMI)”, Version 1.1 CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Suplier Sourcing (CMMI-SE/SW/IPPD/SS, V1.1), Continuous Representations CMU/SEI-2002-TR-011 ESC-TR 2002-011, 2002.

[Cypress doc] DocumentaciónOficial [en línea][Consulta: 26 de Agosto 2020]. Disponible en:<https://docs.cypress.io/es/guides/overview/why-cypress.html>

[Dijkstra, 1970] E. W. Dijkstra, “Notes on Structures Programming”, Technical University Eindhoven, The Netherlands, departamento de Matemáticas, Technical, pp. 15-64 [en línea]. [Consulta: 28 de marzo 2020] Disponible en: <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF>.

[Hetzel, 1988]. Hetzel B. “The Complete Guide to Software Testing, 2nd edition, ISBN 0-89435-242-3, QED, Information Sciences INC, 1988.

[Humphrey, 2001] W. S. Humphrey, Introducción al Proceso Software Personal (psp). Madrid, España: Addison Wesley, 2001, Pág. 247 [en línea]. [Consulta: 16.marzo 2020] Disponible en: <https://docplayer.es/4660334-Introducci-n-al-proceso-software-personal-s-m.ht>.

[Kaner - Bach, 2001] Kaner C., Bach J., Pretichord B., “Lessons Learned in Software Testing”, ISBN 0471081124, Wiley, 2001.

[Kaner, 2008] Cem Kaner, “Software Testing as a Social Science”, Presentation at STEP 2008 Memphis, May 2008 [en línea]. [Consulta: 28 de marzo 2020] Disponible en: <http://www.kaner.com/pdfs/KanerSocialScienceSTEP.pdf>.

[Maida - Pacienzia, 2015]. Maida, Esteban Gabriel, Pacienzia, Julián, Metodologías de Desarrollo de Software. [en línea][Consulta: 12 de mayo 2020]. Disponible en:<https://repositorio.uca.edu.ar/bitstream/123456789/522/1/metodologias-desarrollo-software.pdf >.

[Mera Paz, 2016] Mera Paz, Julián Andrés, Análisis del proceso de pruebas de calidad de software. [en línea], [Consulta: 01 de marzo 2020] <https://www.stickymindwww.researchgate.net/publication/313267627_Analisis_

del_proceso_de_pruebas_de_calidad_de_software/fulltext/5894a6aa45851563f82bc266/Analisis-del-proceso-de-pruebas-de-calidad-de-software.pdf>.

[MIT 2016] Massachusetts Institute of Technology, “Programa en Ciencia, Tecnología y Sociedad", junio 2016, Estados Unidos [en línea]. [Consulta: 16. marzo 2020] Disponible en: <http://web. mit.edu/sts/>.

[Mocha] DocumentaciónOficial [en línea][Consulta: 05 de Octubre 2020]. Disponible en:< http://mochajs.org/>

[Myers, 2004]. Myers, G. “The art of software testing, 2nd edition”, ISBN 0-471-46912-2, John Wiley & Sons Inc, 2004.

[Scalone, 2006] Scalone, Fernanda. Estudio comparativo de los modelos y estándares de calidad del software. Universidad Tecnológica Nacional, Facultad Regional de Buenos Aires, Tesis de Magister en Ingeniería de Calidad [en línea], [Consulta: 10 de marzo 2020] Disponible en: <http://laboratorios.fi.uba.ar/lsi/scalone-tesis-maestria-ingenieria-en-calidad.pdf>

[Selenium.org, 2017] DocumentaciónOficial [en línea][Consulta: 20 de Octubre 2020]. Disponible en:< https://www.selenium.dev/>

[Sommerville, 2011] Ian Sommerville, “Ingeniería de Software”. 9ª ed. México: Pearson Educación 2011. 792 p. ISBN: 978-607-32-0603-7.

[Swebok, 2004] Guide to the software Engineering Body of Knowledge SWEBOK, 2004 version. IEEE Computer Society. [en línea][Consulta: 17 de mayo 2020]. Disponible en:<https://www.computer.org/education/bodies-of-knowledge/software-engineering>

[Whittaker, 2002]. Whitaker, J. “How to break Software, a practical Guide”. ISBN 0201796198, Addison Wesley, 2002.

Anexo I - Descripción detallada de la Selenium

Selenium es un conjunto de diferentes herramientas de software que permiten juntas la automatización de pruebas para aplicaciones basadas en la web. Las herramientas de Selenium realizan operaciones altamente flexibles, lo que habilita muchas opciones para localizar elementos de la interfaz de usuario en el navegador y comparar los resultados esperados de las pruebas respecto del comportamiento real de la aplicación. Una de las ventajas que presenta Selenium es la versatilidad del soporte en diferentes entornos, permitiendo así la ejecución de pruebas en múltiples plataformas de navegador. [Selenium.org, 2017]

Originalmente, Selenium fue desarrollado por Jason Huggins en 2004 para un proyecto privado; aquí él desarrolla una librería en JavaScript que se convirtió en el core de Selenium, misma que subyace a todas las funciones de Selenium Remote Control (RC) y Selenium IDE. Posteriormente Huggins se une con Simon Stewart de Google que había desarrollado un WebDriver también para automatización de pruebas. En 2008 ambos proyectos emergen como uno brindando una herramienta más potente. [Selenium.org, 2017]

Selenium es un software de código abierto que se distribuye bajo la licencia apache 2.0 por lo que puede ser descargado directamente desde el sitio oficial y ser usado sin costo. Así mismo, la empresa se encuentra desarrollando otros proyectos como Selenium Grid, que permite hacer pruebas de concurrencia múltiple o Flash Selenium para experimentar programas escritos en Adobe Flex o Selenium Silverlight. [Selenium.org, 2017]

Componentes de la suite Selenium

Selenium no es una única herramienta, sino que comprende una suite de 4 herramientas de software, donde cada una realiza un rol específico; estas son: Selenium IDE, Selenium RC, Selenium WebDriver, Selenium Grid.

Selenium IDE

Es una herramienta de creación de prototipos para construir scripts de prueba. Esta viene como un plugin de Firefox que proporciona una interfaz para desarrollar pruebas automatizadas. Este IDE tiene la función de grabación y exportar, es decir registra las acciones del usuario en el navegador a medida que las realiza y luego las exporta como una secuencia de comandos, mismos que son reutilizables en varios lenguajes de programación para que posteriormente se puedan ejecutar. Actualmente Selenium provee un API para Java, C#, Ruby y Python. [Selenium.org, 2017]

Arquitectura física de Selenium IDE:

Se trata de una arquitectura monolítica como la mostrada en la Figura 1.

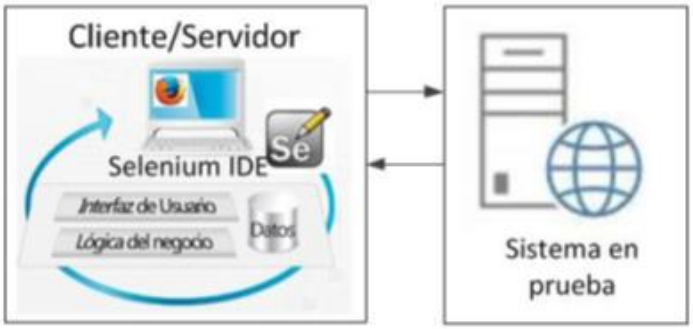

Figura 1. Diagrama de arquitectura de Selenium IDE

La arquitectura soporta múltiples usuarios ya que solo usa los recursos de la máquina cliente, permitiendo ejecutarse en varias instancias del navegador Firefox.

Arquitectura lógica de Selenium IDE:

Selenium IDE por ser una herramienta de entorno de desarrollo integrado y ejecutado en un 100% desde el cliente, contiene en un solo programa las 3 capas en el contexto de arquitectura lógica de sistemas multicapa: capa de presentación, capa de negocio y capa de datos. Como se visualiza en la imagen figura 2.

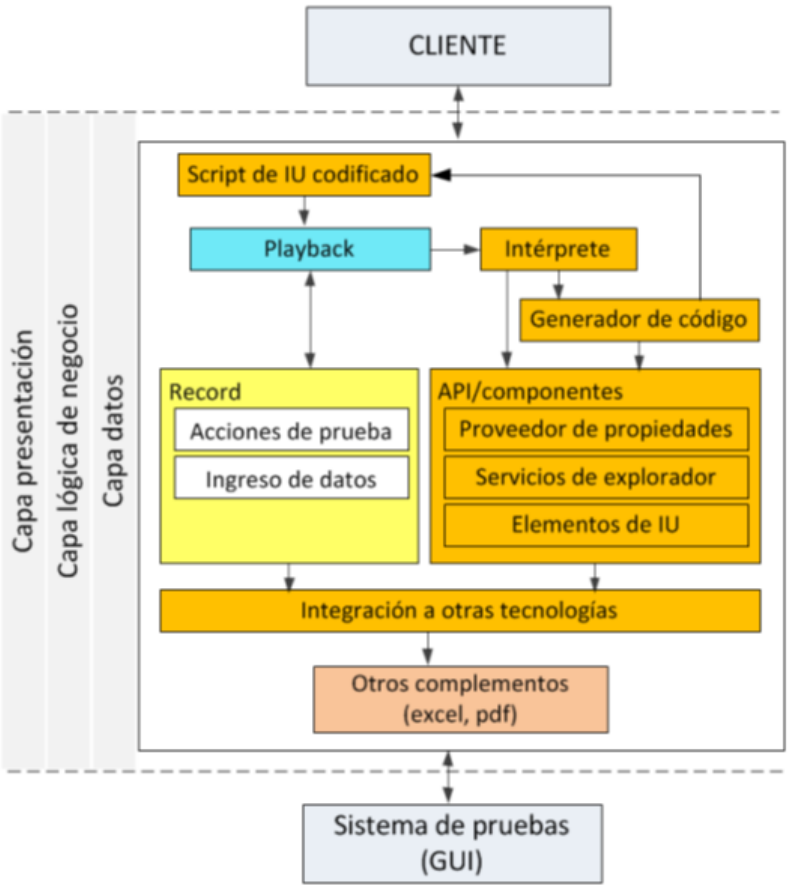

Figura 2. Diagrama de arquitectura lógica de la aplicación Selenium

Dentro de los beneficios de esta arquitectura se encuentra el bajo costo que significa su implementación; operan en una misma máquina tanto el cliente como servidor, sin embargo, una desventaja importante, no permite aislar la lógica de un script de prueba en componentes separados y hace al proceso de automatización de testing escasamente escalable, extensible y modificable. Es decir, no se podría separar la capa de presentación de una capa de lógica de negocio o de los datos.

En la figura 3 se pueden observar las partes más importantes de este plugin de Selenium.

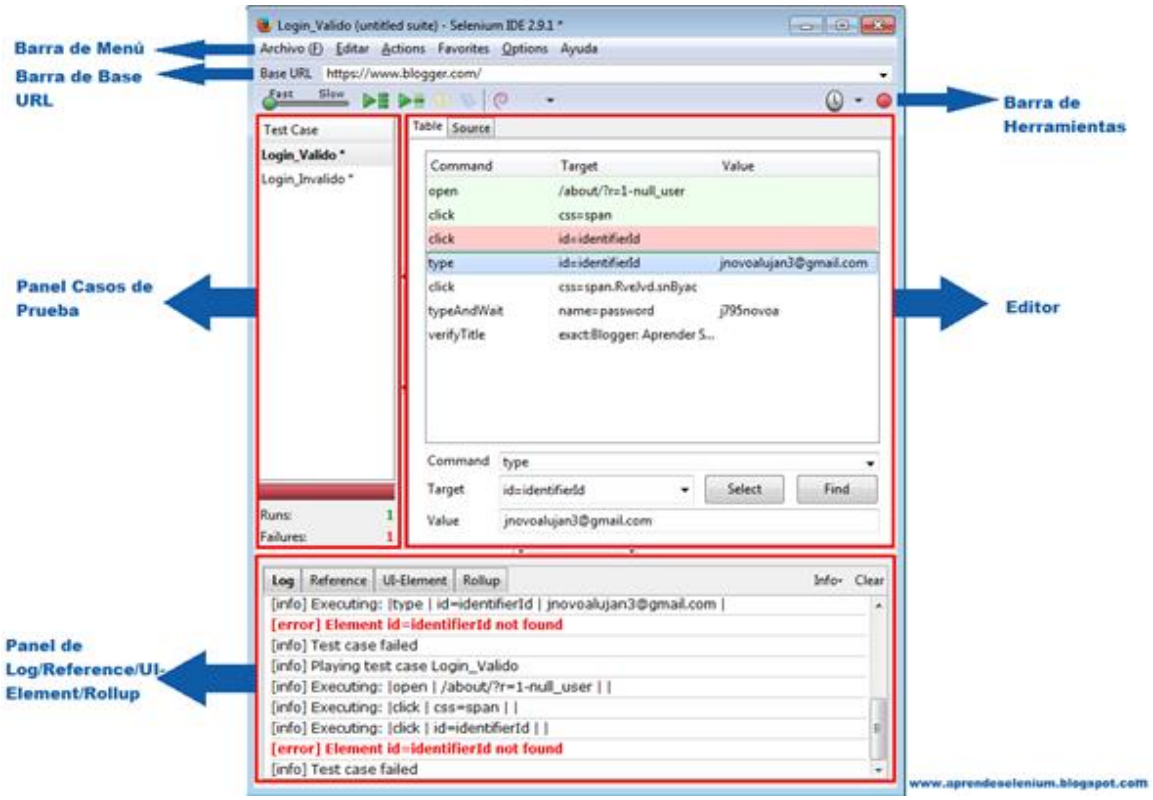

Figura 3. Plugin de Selenium

El panel izquierdo Test Case muestra todos los casos de prueba abiertos. URL Base indica la URL en donde se está realizando la prueba. El panel Table muestra una lista de acciones o líneas de comandos grabadas como parte de la prueba. En el control Fast – Slow se configura la velocidad de reproducción para la prueba grabada. El botón Play reproduce la prueba. El botón rojo es el interruptor de Record para grabar las interacciones de registro en cualquier sitio web. El panel inferior muestra un registro de acciones y una lista de referencias. Una vez llevada a cabo la prueba, se puede guardar el script grabado en el ordenador mediante Archivo > Save Test Case As.

Un Localizador (Locator) es un comando que le dice a Selenium IDE que elementos GUI (cuadro de texto, botones, casillas de verificación, etc) necesita para operar. La correcta identificación de elementos GUI es un requisito previo a la creación de una secuencia de comandos de automatización. Pero la identificación precisa de los elementos es más difícil de lo que parece. Por lo tanto,

Selenium IDE proporciona una serie de localizadores para localizar precisamente un elemento de la GUI.

Los diferentes tipos de localizadores en Selenium IDE:

- ID
- Name
- Link Text
- CSS Selector
  - Tag and ID
  - Tag and class
  - Tag and attribute
  - Tag, class, and attribute
  - Inner text
- DOM (Document Object Model)
  - getElementById
  - getElementsByName
  - dom:name
  - dom: index
- XPath

Hay comandos que no necesitan un localizador (por ejemplo, el comando «open»). Sin embargo, la mayoría de ellos necesita localizadores. Vemos un ejemplo utilizando el cuadro de texto de Correo Electrónico de la página principal de Facebook. Al inspeccionar el elemento en la página notamos que su ID = email, como nos indica la figura 4.

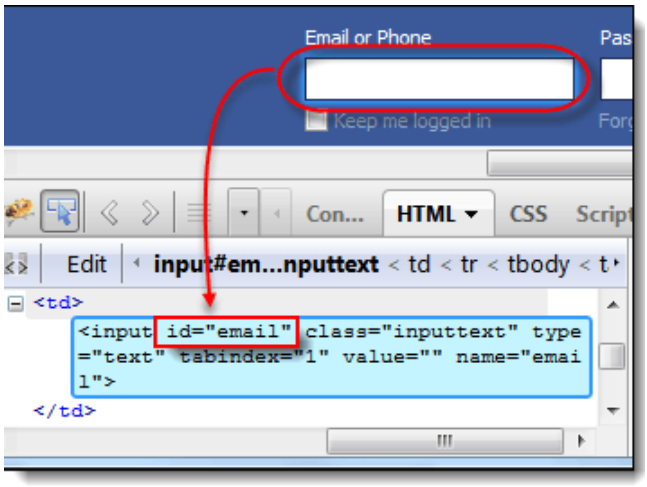

Figura 4. Ejemplo de ID

Por lo que en nuestro editor de Selenium IDE bastaría con escribir una línea nueva con el Target: id=”email”, para localizar el cuadro de texto correctamente, tal como lo muestra la figura 5.

Figura 5. Ejemplo de target ID

Comandos Selenese de Selenium IDE:

Estos comandos pueden tener hasta un máximo de dos parámetros: Target y Value; y estos parámetros no se requieren todo el tiempo, depende del número de comandos que se necesiten. Existen tres tipos de comandos: Actions, Accesors y Assertions.

Actions: Son comandos que interactúan directamente con los elementos de la página.

ejemplo: el comando «click» es una acción porque se interactúa directamente con el elemento que está haciendo clic, el comando «type» también es una acción porque está poniendo valores en un cuadro de texto y el cuadro de texto se los muestra a cambio. Hay una interacción de dos vías entre el usuario y el cuadro de texto.

Accessors: son comandos que permiten almacenar valores en una variable.

ejemplo: el comando «storeTitle» es un descriptor de acceso porque sólo «Lee» el título de la página y lo guarda en una variable. No interactúa con ningún elemento de la página.

Assertions: son comandos que verifican si se cumple una determinada condición.

3 Tipos de Assertions

- Assert. Cuando un comando «Assert» falla, la prueba se detiene inmediatamente

- Verify. Cuando falla un comando «Verify», Selenium IDE registra este error y continúa con la ejecución de la prueba.

- WaitFor. Antes de continuar con el siguiente comando, los comandos «Wait» esperarán que una cierta condición se haga realidad.

- Si la condición se hace realidad dentro del período de espera, el paso pasa.

- Si la condición no se hace realidad, el paso falla. El error se registra y el proceso de ejecución de prueba se realiza al siguiente comando.

- De forma predeterminada, el valor de tiempo de espera se establece en 30 segundos. Puede cambiar esto en el cuadro de diálogo Opciones de Selenium IDE en la ficha General.

En la figura 6 vemos un ejemplo de manejo de variables en Selenium IDE mediante el comando StoreElementPresent. Este comando almacena True o False dependiendo de la presencia del elemento especificado. El script booleano

a continuación almacena el valor True en la variable Var1 y False en Var2, mientras que para verificarlo usa el comando Echo para mostrar los valores de Var1 y Var2.

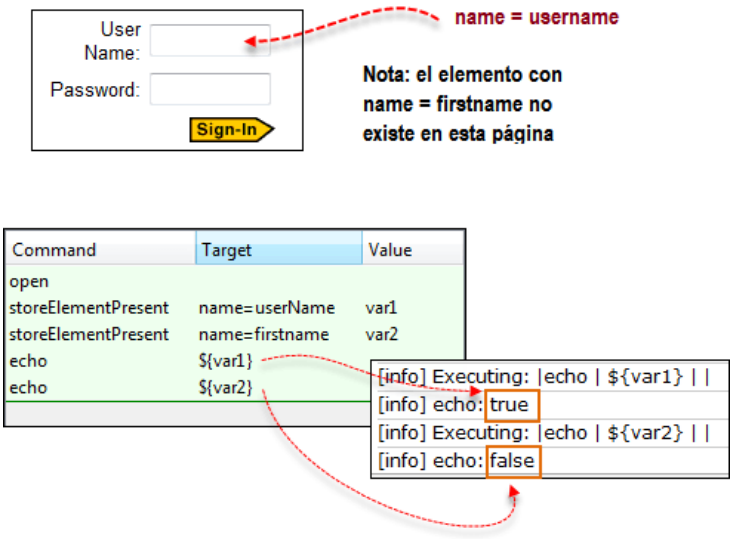

Figura 6. Manejo de variables

Principales características de Selenium IDE:

- Proporciona una plataforma simple para verificar la funcionalidad de la aplicación.

- Se ejecuta en muchos navegadores y sistemas operativos, y puede ser controlado por muchos lenguajes de programación y frameworks de pruebas.

- Con las pruebas, Selenium es capaz de probar las aplicaciones web desde la perspectiva del usuario, no desde el punto de vista de código y permitirá tener tantas pruebas diferentes como deseen para un mismo sistema y ejecutarlas una por vez automáticamente.

- Provee unos Apis en diferentes lenguajes (PHP, Ruby, JAVA, Javascript) que permiten indicar mediante comandos que pruebas debe hacer.

- Posee un IDE que automatiza aún más la tarea, es sencillo y ayuda a aprender los comandos más rápidamente.

Selenium Remote Control (RC)

También se lo conoce como Selenium 1, ya que fue reemplazado por Selenium WebDriver (Selenium 2). Este es un servidor desarrollado en Java que acepta comandos al navegador vía HTTP. Esta herramienta hace posible escribir pruebas automatizadas para aplicaciones web, en cualquier lenguaje de programación lo que resulta en una mejor integración a entornos de prueba ya existentes. La función principal de esta herramienta es la de ejecutar los test en diferentes navegadores y en diferentes plataformas.

Descripción del funcionamiento:

El cliente tendrá el test adaptado al lenguaje de programación con el que se realizará la prueba. Para ello, hará uso de las librerías proporcionadas para manejar los objetos necesarios durante la prueba. Estos objetos permiten manejar la aplicación.

Lo primero que hará el cliente es conectarse al servidor Selenium, el cual deberá de estar arrancado como un proceso independiente y activo durante toda la prueba. En caso de que por alguna necesidad se necesite interrumpir dicho servidor se realizará cualquier actividad de parada o arranque desde la línea de comandos. Después el servidor Selenium ejecutará el navegador seleccionado para el test, para ello abrirá dos ventanas:

- Ventana de Selenium Core: Visualiza los comandos que se están ejecutando.

- Ventana de página web: Visualiza las respuestas realizadas por los comandos anteriores. Dichas ventanas serán visibles desde el cliente que las ejecutó. Es muy importante decir que la ejecución del navegador se realiza mediante el uso de un servidor proxy, el cual se establece entre el navegador y el sitio web. De esta forma, Selenium permite habilitar un navegador para poder ejecutarse en sitios arbitrarios. El servidor Selenium no necesita correr en la misma máquina virtual (JVM) o en la misma máquina física. Una vez el servidor Selenium recibe la primera instrucción del cliente (vía HTTP proxy), realiza la acción recibida en la primera instrucción. Posteriormente el servidor web se sincronizará con la página, visualizando los cambios realizados (se podrán ver los resultados de las acciones realizadas en la ventana de página web).

Este funcionamiento se puede ver en la figura 7.

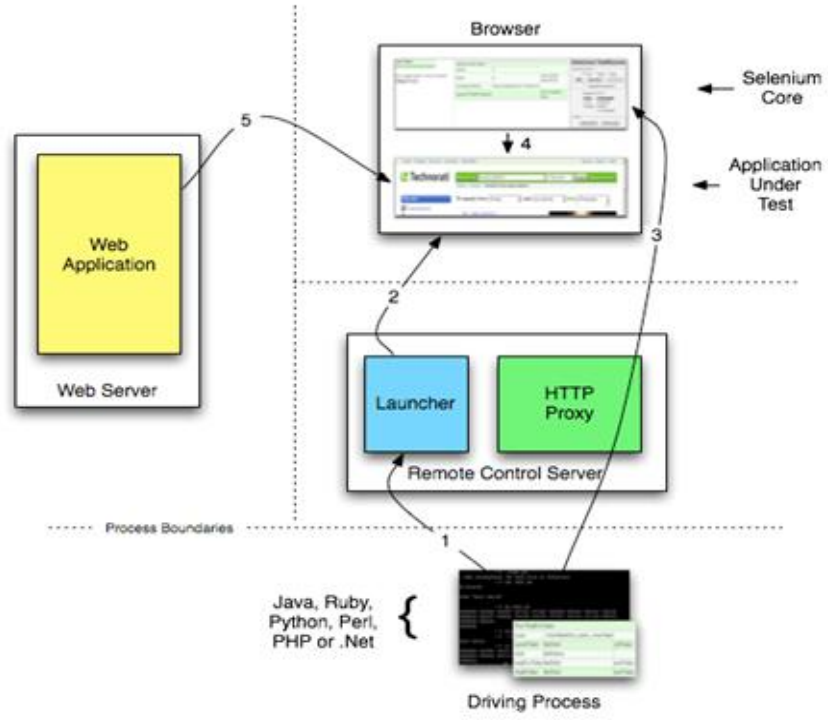

Figura 7. Funcionamiento de Selenium RC

Actualmente Selenium RC está oficialmente obsoleto y no cuenta con soporte. [Selenium.org, 2017]

Selenium WebDriver

Es la nueva herramienta de automatización de Selenium que incluye nuevas características tales como una API más cohesiva y orientada a objetos, así como también una respuesta a las limitaciones de Selenium 1. Es compatible con WebDriver y ejecuta la interfaz Selenium RC para compatibilidad con versiones anteriores. Selenium WebDriver acepta comandos que se pueden enviar con Selenese (lenguaje específico de dominio para escribir pruebas que se pueden ejecutar en un amplio número de lenguajes de programación) o con el API de cliente hacia el navegador de preferencia. [Selenium.org, 2017]

Selenium WebDriver funciona realizando llamadas directas al navegador haciendo uso del soporte nativo de cada navegador para la automatización. Dado la variedad de navegadores web a disposición y varios lenguajes de programación populares, para desarrollar y correr las pruebas se necesita una especificación

común, la misma que es proporcionada por la API de WebDriver. Cada navegador tiene que implementar esta API llamada Remote WebDriver o Remote WebDriver Server. En un nivel superior, la arquitectura Selenium WebDriver se parece a lo mostrado en la figura 8. [Selenium.org, 2017]

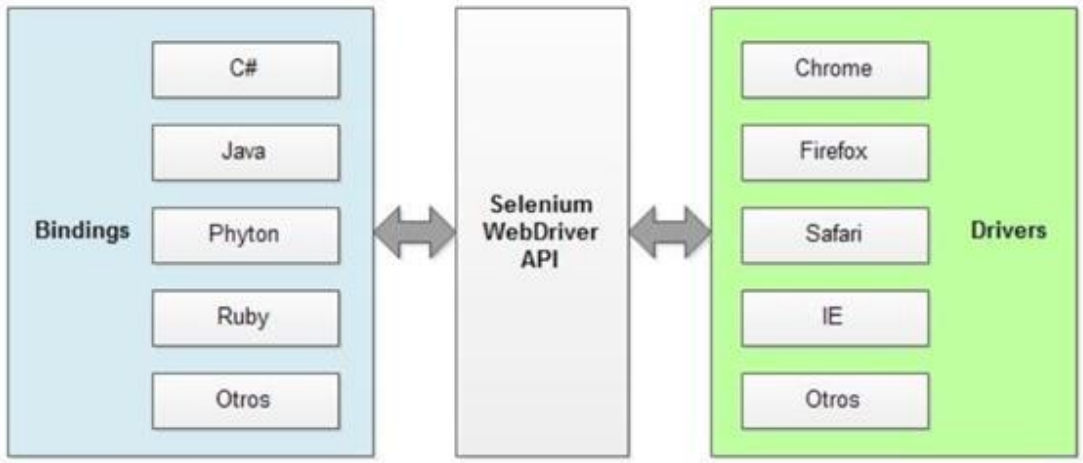

Figura 8. Arquitectura de Selenium WebDriver.

El proceso de pruebas con Selenium WebDriver explicado resumidamente es el siguiente: el enlace de idioma envía los comandos a través del controlador común que proporciona la API, en el otro extremo existe también un controlador que escuchará estos comandos y los ejecutará en el navegador seleccionado utilizando el WebDriver remoto; finalmente este devolverá el resultado a través de API en el código de enlace. Hay que tomar en cuenta que cualesquiera sean los comandos emitidos en el código, estos se interpretarán en métodos de servicio web, con el protocolo JSON y luego el controlador remoto recibirá la solicitud HTTP, los ejecutará en el navegador y luego devolverá la respuesta. Para Selenium WebDriver todos los elementos de una página web como los campos de texto, botones, links, imágenes son WebElements. Existen distintos tipos de localizadores para ubicar un elemento y realizar alguna acción sobre él. De esta misma forma puede también acceder a atributos visibles como el texto (mediante el método getText()) y a atributos no visibles desde la interfaz de usuario a través de comandos como getAttribute(“nombre del atributo”). Para comunicarle a Selenium cómo encontrar un elemento utilizamos el comando: driver.

findElement() que es el método encargado de devolver el WebElement y que recibe como parámetros un localizador (By). [Selenium.org, 2017]

En la figura 9 se muestran cuáles son los distintos tipos de localizadores usados en Selenium WebDriver.

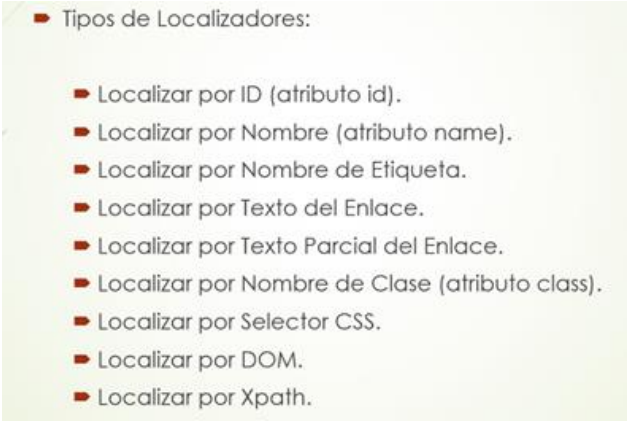

Figura 9. Localizadores de Selenium WebDriver

La localización por ID es una forma rápida de encontrar un elemento, cuando el mismo es único. Tal como muestra la figura 10.

Figura 10. Ejemplo de localización por el atributo Id

Validaciones:

-Sentencias Condicionales

- Assert: Son validaciones o puntos de control para una aplicación. Se puede decir que las afirmaciones en Selenium se utilizan para validar los casos de prueba. Ayudan a los evaluadores a comprender si las pruebas han pasado o no.

- JUnit: Es la herramienta utilizada para hacer pruebas unitarias en Java

En la figura 11 se muestra un ejemplo de uso de la sentencia condicional IF en Selenium WebDriver.

```
package ar.gov.mecon.dgsiaf.selenium.curso;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class Prueba99_PruebaIF {
    private static final String TITLE_EXPECTED = "Welcome: Mercury Tours";
    private static String TITLE_ACTUAL = "";
    public static void main(String[] args) {
        System.setProperty("webdriver.gecko.driver","exe/geckodriver.exe");
        WebDriver driver = new FirefoxDriver();
        String baseUrl = "http://newtours.demoaut.com/";
        driver.get(baseUrl);
        TITLE_ACTUAL = driver.getTitle();
        if (TITLE_ACTUAL.contentEquals(TITLE_EXPECTED)) {
            System.out.println("Test OK!");
        } else {
            System.out.println("Test Falló!");
        }
        driver.close();
    }
}
```

Figura 11. Ejemplo uso sentencia IF

Un aspecto importante durante la creación de los tests automáticos con Selenium WebDriver es el tiempo de carga de la página o de los WebElements. Si no se tienen en cuenta estos tiempos, y no se ponen las protecciones adecuadas frente a ellos, es probable que nuestros tests fallen debido a un error en la programación de la prueba.

Los métodos para la localización de WebElements lanzan una excepción cuando no encuentran ninguno que coincida con el patrón o mecanismo de búsqueda. Para evitar que esta situación se produzca por el hecho de que no se haya cargado el WebElement en el momento en el que el script intenta localizarlo, Selenium WebDriver ofrece unos mecanismos de espera. Timeouts.

- ImplicitlyWait(): Define el tiempo máximo de espera para la carga de cualquier WebElement. Si vence el timeout se lanza la excepción.

- PageLoadTimeout(): Define el tiempo máximo de espera para la carga completa de la página web. Si vence el timeout se lanza la excepción.

- SetScriptTimeout(): Define el tiempo máximo de espera para que la ejecución asíncrona de un programa javascript finalice. Si vence el timeout se lanza la excepción.

A través del método Manage().Timeouts() accedemos a los métodos antes descritos. En la Figura 12 podemos ver un ejemplo del manejo de Timeouts.

Ejemplo cap9.Timeouts.java

```
/*
 * EJEMPLO: Interfaz WebDriver.Timeouts
 * ACCIONES
 * Al no poder cargar la pagina en 1 milisegundo, el metodo pageLoadTimeout
   lanza la excepcion TimeoutException
*/

WebDriver driver = new FirefoxDriver();
try{
      driver.manage().timeouts().implicitlyWait(1, TimeUnit.SECONDS);
      driver.manage().timeouts().pageLoadTimeout(1, TimeUnit.MILLISECONDS);
      …

      Actions clickEncursor1 = new Actions(driver);
      clickEncursor1
            .moveToElement(checkbox1)
            .click();
      clickEncursor1.perform();

      Thread.sleep(2000);
      Actions clickEnElemento = new Actions(driver);
      clickEnElemento
            .click(checkbox2);
      clickEnElemento.perform();

      Thread.sleep(2000);

}catch(org.openqa.selenium.TimeoutException e){
      System.out.println(e);
}finally{
      driver.close();
}
```

Figura 12. Ejemplo de Interfaz Timeouts

Ventajas de Selenium WebDriver:

·WebDriver está diseñado en una interfaz de programación más simple y concisa junto con algunas limitaciones en la API de Selenium-RC.

·Comparada con Selenium 1.0, WebDriver es una API compacta orientada a objetos.

·Impulsa el navegador de manera mucho más efectiva y supera las limitaciones de Selenium 1.x, que afectó la cobertura de prueba funcional, como la carga o descarga de archivos, ventanas emergentes y barreras de diálogo.

- WebDriver supera la limitación de la política de origen de host único de Selenium RC.

Comparación entre Selenium 1 (Remote Control) y Selenium 2 (WebDriver):

- Una gran diferencia y ventaja que posee Selenium WebDriver respecto de su antecesor es que en Selenium 1 el servidor Selenium RC era indispensable; en esta nueva versión la herramienta inicia una instancia del navegador y la controla para la ejecución de las pruebas sin la necesidad de un servidor externo.

- Otra diferencia interesante es que Selenium 1 apuntaba a suministrar una interfaz diversa, con varias opciones de operaciones, mientras que Selenium WebDriver busca proveer bloques de construcción básicos, o mejor dicho prototipos, sobre los cuales los desarrolladores puedan escribir en su propio lenguaje específico de dominio las pruebas.

Desde el 2012, Simon Stewart del equipo de Selenium y David Burns de la Fundación Mozilla realizan negociaciones con el World Wide Web Consortium (W3C) para que Selenium WebDriver se convierta en un estándar de Internet, apuntando de esta forma a que la herramienta sea la implementación de referencia del estándar WebDriver en varios lenguajes de programación.

Anexo II - Descripción detallada Cypress

Cypress es una herramienta de testeo front end de código abierto de última generación creada para la web moderna. Este framework todo en uno, incluye librerías de aserciones, mocks y pruebas EndToEnd (de principio a fin) automáticas sin utilizar Selenium.

Cypress cuenta con una arquitectura construida desde cero, que ejecuta los comandos en el mismo ciclo de ejecución. [Cypress doc]

Figura 1. Cypress herramienta todo en uno

Cypress es una herramienta utilizada especialmente para manejar frameworks de javaScript modernos como por ejemplo React, Angular, Vue, etc. Pero funciona perfectamente en páginas o aplicaciones renderizadas en servidor.

*"Cypress prueba todo lo que se ejecuta en un navegador" [*Cypress doc]

Se utiliza para hacer principalmente test EndToEnd (tests que simulan el comportamiento de un usuario real. Prueban toda la aplicación de principio a fin).

Escribir pruebas EndToEnd requiere del uso de muchas herramientas, con Cypress se obtienen muchas herramientas en una, no es necesario instalar dichas herramientas, ni librerías independientes para configurar un conjunto de

pruebas. Se utiliza para que las pruebas y los desarrollos puedan ocurrir simultáneamente. [Cypress doc]

Cypress test Runner (ejecutor de pruebas)

Cypress ejecuta pruebas en un ejecutor o corredor de pruebas, el cual permite ver los comandos que se ejecutan mientras se muestra la aplicación que se está probando. En la figura 2 se puede ver el ejecutor de pruebas. [Cypress doc]

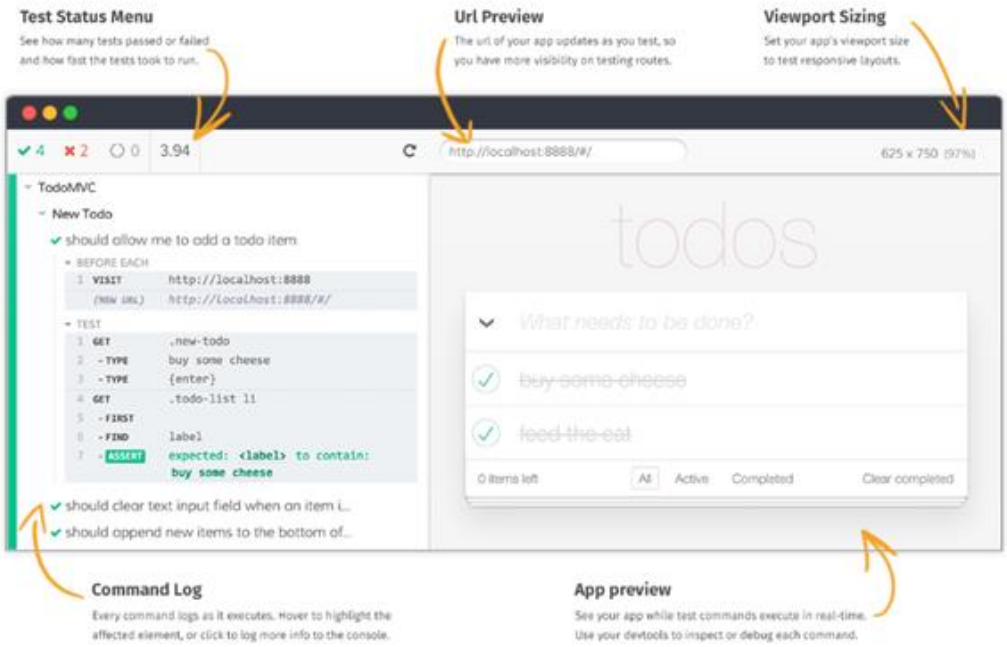

Figura 2. Descripción de Cypress Test Runner

El registro de comandos (Command Log) del Corredor de pruebas, es una representación visual del conjunto de pruebas. Cada bloque de prueba está anidado y cada prueba al clickear sobre esta, muestra todos los comandos y aserciones de Cypress dentro del bloque de prueba, así como también comando y aserciones ejecutados dentro de los comandos `before`, `beforeEach`, `afterEach`, y `after`. [Cypress doc]

Figura 3. Command log de Cypress Test Runner

Cada comando o aserción, cuando se coloca sobre él, restaura la aplicación bajo prueba al estado en que se encontraba cuando se ejecutó dicho comando o aserción, esto permite viajar a estados anteriores de la aplicación bajo prueba.

Errores

Cypress imprime varios datos cuando se producen errores durante la ejecución de pruebas tal como lo muestra la Figura 4.

Figura 4. Descripción de Errores

1. Nombre del Error: es el tipo de error por ejemplo (AssertionError, CypressError).

2. Mensaje de Error: es una descripción de cuál fue la causa del error, a veces puede indicar cómo corregir dicho error.

3. Más información: a veces contienen un enlace a la documentación que dará más información.

4. Código en archivo: al hacer click en este enlace se muestra el archivo y la línea de de código que fallo.

5. Muestra un fragmento del código donde ocurrió el error, con la línea resaltada.

6. Al hacer click se puede hacer un seguimiento de la pila.

7. Al hacer click se imprime el error en consola.

Comandos en Cypress

Una cadena de comandos en Cypress comienza con cy.[command] donde command puede ser concatenado por otros comandos.

cy.visit()

El comando cy.visit() se utiliza para dirigirnos a una URL. Cuando se hace click en nuestra prueba el navegador se dirige a la página indicada en el código, como se puede observar en la figura 5. En el ejecutor de pruebas del lado derecho se puede observar la aplicación que está bajo prueba. [Cypress doc]

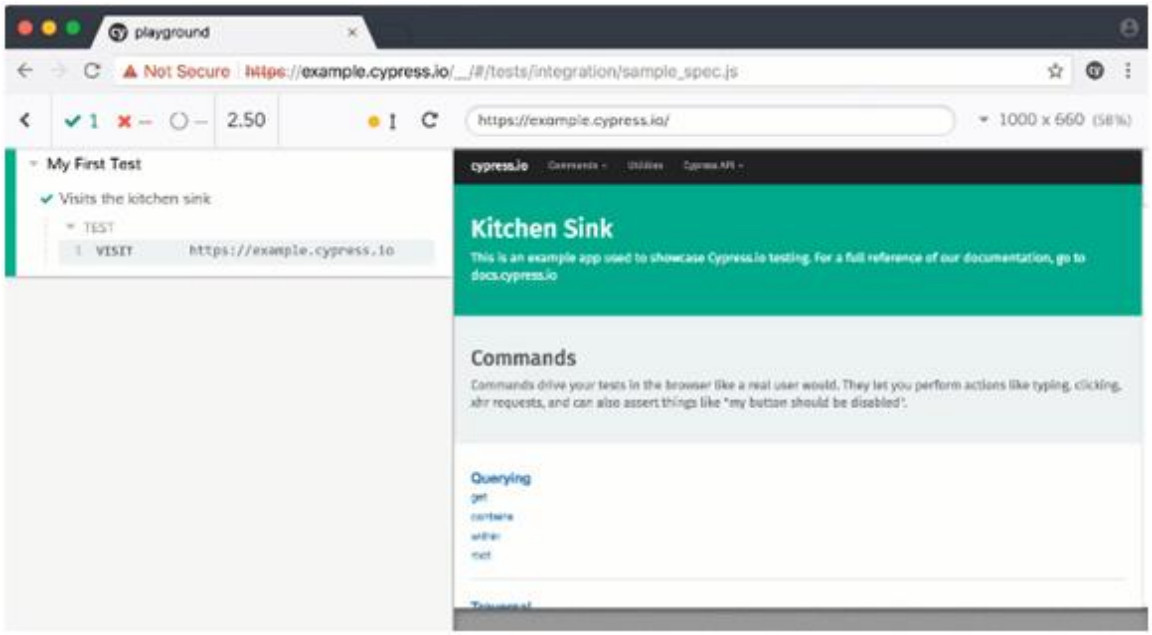

Figura 5. Visita de página mediante el comando cy.visit()

Cuando Cypress consulta un elemento en el DOM, reintenta automáticamente la consulta hasta que el elemento se encuentra o se alcanza un tiempo de espera establecido esto hace que sea muy robusto y no lo afecten muchos problemas que tienen otras herramientas de pruebas. Estos problemas pueden producirse por ejemplo cuando el DOM no se ha cargado completamente, no se ha completado una animación, etc. Antes había que escribir código personalizado contra cada uno de estos problemas como ser esperas, reintentos condicionales y comprobaciones nulas que ensucian las pruebas. Cypress evita estos problemas. [Cypress doc]

cy.get()

Este comando se utiliza para obtener elementos del DOM.

En la Figura 6 se puede observar algunos ejemplos del uso del comando cy.get()

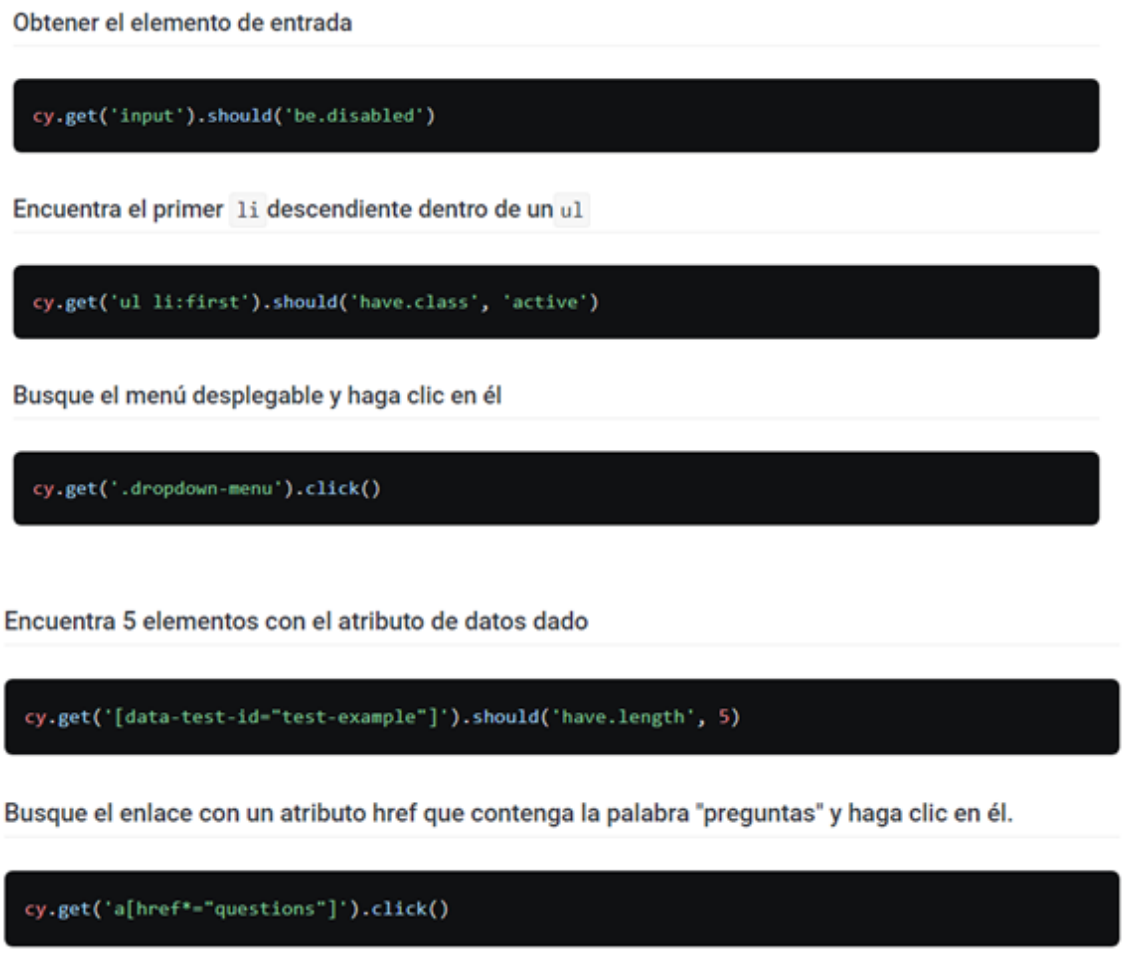

Figura 6. Ejemplos de la utilización del comando cy.get()

cy.contains()

Obtiene el elemento del DOM que contiene el texto deseado.

En la Figura 7, se puede ver un ejemplo de .contains()

Figura 7. Ejemplos de la utilización de cy.contains()

.click()

Se utiliza para hacer click en un elemento del DOM

En la Figura 8 se pueden observar algunos ejemplos del uso del .click():

Figura 8. Ejemplos del comando .click()

.type()

Se utiliza para escribir un texto en un elemento del DOM. En la figura 9 se puede observar un ejemplo del comando type.

```
cy.get('textarea').type('Hello world') // yields <textarea>
```

Figura 9. Ejemplos del comando type()

Cypress se basa en muchas de las mejores bibliotecas de pruebas de código abierto. Utiliza funciones provenientes de herramientas integradas describe() y it() provienen de Mocha. Cypress ha adoptado la sintaxis de Mocha, que encaja perfectamente tanto en la integración como en las pruebas unitarias. Todas las pruebas que se escriben se basan en el arnés fundamental que proporciona Mocha. En la figura 10 pueden verse algunas funciones utilizadas. [Mocha]

- `describe()`
- `context()`
- `it()`
- `before()`
- `beforeEach()`
- `afterEach()`
- `after()`
- `.only()`
- `.skip()`

Figura 10. Funciones provenientes de Mocha

Así como Mocha proporciona un marco para estructuras de pruebas, Chain brinda a Cypress la capacidad para escribir Aserciones (afirmaciones) fácilmente. Brinda aserciones legibles con excelentes mensajes de error. Cypress extiende esto, corrige varios errores comunes y envuelve el DSL de Chain usando sujetos y el comando. should() . [Chain]

En la Figura 11, se pueden ver algunos ejemplos de encadenadores para aserciones en Chain, estos encadenadores pueden utilizarse con (expect / should).

Chainer	Example
not	`expect(name).to.not.equal('Jane')`
deep	`expect(obj).to.deep.equal({ name: 'Jane' })`
nested	`expect({a: {b: ['x', 'y']}}).to.have.nested.property('a.b[1]')` `expect({a: {b: ['x', 'y']}}).to.nested.include({'a.b[1]': 'y'})`
ordered	`expect([1, 2]).to.have.ordered.members([1, 2]).but.not.have.ordered.members([2, 1])`
any	`expect(arr).to.have.any.keys('age')`
all	`expect(arr).to.have.all.keys('name', 'age')`
a(*type*) Aliases: an	`expect('test').to.be.a('string')`
include(*value*) Aliases: contain, includes, contains	`expect([1,2,3]).to.include(2)`
ok	`expect(undefined).to.not.be.ok`
true	`expect(true).to.be.true`
false	`expect(false).to.be.false`
null	`expect(null).to.be.null`
undefined	`expect(undefined).to.be.undefined`
exist	`expect(myVar).to.exist`
empty	`expect([]).to.be.empty`
arguments Aliases: Arguments	`expect(arguments).to.be.arguments`
equal(*value*) Aliases: equals, eq	`expect(42).to.equal(42)`
deep.equal(*value*)	`expect({ name: 'Jane' }).to.deep.equal({ name: 'Jane' })`

Figura 11. Algunos encadenadores Chain

En la Figura 12, se puede observar algunos captadores disponibles que sirven para escribir oraciones más claras.

`to` , `be` , `been` , `is` , `that` , `which` , `and` , `has` , `have` , `with` , `at` , `of` , `same`

Figura 12. Captadores encadenables

En la Figura 13 se puede ver algunas aserciones disponibles de Chain

Afirmación	Ejemplo
.isOk (*objeto* , *[mensaje]*)	`assert.isOk('everything', 'everything is ok')`
.isNotOk (*objeto* , *[mensaje]*)	`assert.isNotOk(false, 'this will pass')`
.equal (*real* , *esperado* , *[mensaje]*)	`assert.equal(3, 3, 'vals equal')`
.notEqual (*real* , *esperado* , *[mensaje]*)	`assert.notEqual(3, 4, 'vals not equal')`
.strictEqual (*real* , *esperado* , *[mensaje]*)	`assert.strictEqual(true, true, 'bools strict eq')`
.notStrictEqual (*real* , *esperado* , *[mensaje]*)	`assert.notStrictEqual(5, '5', 'not strict eq')`
.deepEqual (*real* , *esperado* , *[mensaje]*)	`assert.deepEqual({ id: '1' }, { id: '1' })`
.notDeepEqual (*real* , *esperado* , *[mensaje]*)	`assert.notDeepEqual({ id: '1' }, { id: '2' })`
.isAbove (*valueToCheck* , *valueToBeAbove* , *[mensaje]*)	`assert.isAbove(6, 1, '6 greater than 1')`
.isAtLeast (*valueToCheck* , *valueToBeAtLeast* , *[mensaje]*)	`assert.isAtLeast(5, 2, '5 gt or eq to 2')`
.isBelow (*valueToCheck* , *valueToBeBelow* , *[mensaje]*)	`assert.isBelow(3, 6, '3 strict lt 6')`
.isAtMost (*valueToCheck* , *valueToBeAtMost* , *[mensaje]*)	`assert.isAtMost(4, 4, '4 lt or eq to 4')`
.isTrue (*valor* , *[mensaje]*)	`assert.isTrue(true, 'this val is true')`
.isNotTrue (*valor* , *[mensaje]*)	`assert.isNotTrue('tests are no fun', 'val not true')`
.isFalse (*valor* , *[mensaje]*)	`assert.isFalse(false, 'val is false')`

Figura 13. Aserciones de chain

En la Figura 14, se describe cómo se utilizan desde Cypress algunas aserciones de elementos comunes, algunas enumeradas anteriormente, utilizando el comando .should().

Longitud

```
// retry until we find 3 matching <li.selected>
cy.get('li.selected').should('have.length', 3)
```

Clase

```
// retry until this input does not have class disabled
cy.get('form').find('input').should('not.have.class', 'disabled')
```

Valor

```
// retry until this textarea has the correct value
cy.get('textarea').should('have.value', 'foo bar baz')
```

Contenido del texto

```
// retry until this span does not contain 'click me'
cy.get('a').parent('span.help').should('not.contain', 'click me')
```

Visibilidad

```
// retry until this button is visible
cy.get('button').should('be.visible')
```

Existencia

```
// retry until loading spinner no longer exists
cy.get('#loading').should('not.exist')
```

Figura 14. Aserciones de elementos comunes