A4Q
Programa de Estudios de Probador de Selenium Básico

Versión de la
Entrega
2018

Versión 2018

Historial de la Revisión

Versión	**Fecha**	**Observaciones**
Versión 2018	5 de agosto de 2018	1. Versión de la entrega
Versión 2018	3 de diciembre 2018	pequeñas correcciones

Tabla de Contenidos

0 Introducción

0.1 Propósito de este Programa de Estudios

Este programa de Estudios presenta los resultados comerciales, objetivos de aprendizaje, y los conceptos esenciales de la capacitación y certificación de Probador Básico de Selenium.

0.2 Objetivos de aprendizaje examinables y niveles de conocimiento cognitivos

Los objetivos de aprendizaje apoyan los resultados comerciales y se utilizan para crear el examen para lograr la Certificación de Probador Básico de Selenium.

En general, todos los contenidos de este programa de estudios son examinables en un nivel K1, a excepción de la Introducción y los Apéndices. Es decir, se le puede pedir al candidato que reconozca, recuerde o memorice una palabra clave o concepto mencionado en cualquiera de los cuatro capítulos. Los niveles de conocimiento de los objetivos de aprendizaje específicos se muestran al comienzo de cada capítulo y se clasifican de la siguiente manera:

- K1: recordar
- K2: comprender
- K3: aplicar
- K4: analizar

Las definiciones de todos los términos enumerados como palabras clave justo debajo de los títulos de los capítulos se recordarán (K1), incluso si no se mencionan explícitamente en los objetivos de aprendizaje.

0.3 El Examen de Probador de Selenium Básico

El Examen de Probador de Selenium Básico se basará en este programa de estudios y en el curso acreditado de capacitación A4Q de Probador de Selenium Básico. Las respuestas a las preguntas del examen pueden requerir el uso de material basado en más de una sección de este programa de estudios y/o el curso de capacitación de Probador de Selenium Básico. Se pueden examinar todas las secciones del plan de estudios y el curso de capacitación de Probador de Selenium Básico, a excepción de la Introducción y los Apéndices.
Los estándares, los libros y los programas de la ISTQB® pueden incluirse como referencias, pero su contenido no es examinable, más allá de lo que se resume en este plan de estudios de dichos estándares, libros y programas de la ISTQB®.

El examen estará compuesto por 40 preguntas de selección múltiple. Cada respuesta correcta tiene un valor de un punto. Se requiere una calificación de al menos 65% (es decir, 26 o más preguntas respondidas correctamente) para aprobar el examen. El tiempo permitido para tomar el examen es de 60 minutos. Si el idioma nativo del candidato no es el idioma del examen, se le puede otorgar un 25% (15 minutos) de tiempo extra.

Los exámenes solo se pueden tomar después de tomar la capacitación A4Q de Probador de Selenium Básico, ya que la evaluación del instructor sobre la competencia del candidato en los ejercicios es parte de la obtención de la certificación.

0.4 Acreditación

La Capacitación A4Q de Probador de Selenium Básico es el único curso de capacitación acreditado.

0.5 Nivel de detalle

El nivel de detalle en este programa de estudios permite exámenes coherentes internacionalmente. Para lograr este objetivo, el programa de estudios consiste en:

- Objetivos generales de instrucción que describen la intención del Nivel Básico
- Una lista de términos que los estudiantes deben poder recordar
- Objetivos de aprendizaje para cada área de conocimiento, describiendo el resultado del aprendizaje cognitivo que se logrará
- Una descripción de los conceptos clave, incluidas referencias a fuentes tales como literatura o normas aceptadas

El contenido del programa de estudios no es una descripción de toda el área de conocimiento de las pruebas automatizadas de Selenium; refleja el nivel de detalle que se cubrirá en los cursos de capacitación de nivel básico. Se centra en conceptos y técnicas de prueba que pueden aplicarse a todos los proyectos de software, incluidos los proyectos Ágiles. Este programa de estudios no contiene objetivo de aprendizaje específico alguno relacionado con un ciclo de vida o método de desarrollo de software en particular, pero sí analiza cómo se pueden aplicar estos conceptos en varios ciclos de vida de desarrollo de software.

0.6 Cómo está organizado este programa de estudios

Hay cuatro capítulos con contenido examinable. El encabezado de nivel superior para cada capítulo especifica el tiempo para el capítulo; el tiempo no se proporciona debajo del nivel del capítulo. Para el curso de Capacitación A4Q de Probador de Selenium Básico, el programa de estudios requiere un mínimo de 16.75 horas de instrucción, distribuidas en los cuatro capítulos de la siguiente manera:

Capítulo 1: Conceptos Básicos de Automatización de Prueba 105 minutos
Capítulo 2: Tecnologías de Internet para la Automatización de Pruebas para Aplicaciones basadas en la Web 195 minutos Capítulo 3: Utilización de Selenium WebDriver 495 minutos
Capítulo 4: Preparación de Guiones de Prueba Mantenibles 225 minutos

0.7 Resultados Comerciales

SF-BO-1 Aplicar correctamente los principios de automatización de la prueba para construir una solución de automatización de prueba mantenible
SF-BO-2 Ser capaz de seleccionar e implementar correctamente herramientas de automatización de la prueba
SF-BO-3 Ser capaz de implementar guiones de Selenium WebDriver que ejecuten pruebas de aplicaciones basadas en la Web funcionales
SF-BO-4Ser capaz de implementar guiones mantenibles

0.8 Acrónimos

ALIAS: También conocido como
IPA: Interfaz de Programación de Aplicaciones
CERN: Consejo Europeo de Investigación Nuclear (Del francés - Conseil Europeen pour la Recherche Nucleaire)
IC: Integración Continua
CSS: Hojas de Estilo en Cascada
DOM: Modelo de Objeto de Documento GUI: Interfaz Gráfica de Usuario
HTTP: Protocolo de Transferencia de Hipertexto
ISTQB®:Junta Internacional de Calificaciones de Pruebas de Software
KDT: Prueba Guiada por Palabras Clave
REST: Transferencia de Estado Representativo
ROI: Rendimiento de la Inversión
SDLC: Ciclo de Vida de Desarrollo de Sistemas o Ciclo de Vida de Desarrollo de Software
SOAP: Simple Object Access Protocol/Protocolo de Acceso de Objeto Simple
SUT: Sistema Sujeto a Prueba
TAA: Arquitectura de Automatizacion de Prueba
TAE: Ingeniero de Automatización de la Prueba
TAS: Solución de Automatización de la Prueba
TCP: Protocolo de Control de Transporte
IU: Interfaz de Usuario
W3C: Consorcio World Wide Web

Capítulo 1 - Conceptos Básicos de Automatización de la Prueba

Palabras Clave

arquitectura, captura/repetición, comparador, prueba exploratoria, ataque de faltas, marco, anzuelo, paradoja del pesticida, deuda técnica, capacidad de ser probado, arnés de prueba, oráculo de prueba, productos de prueba

Objetivos de Aprendizaje para los Conceptos Básicos de Automatización de la Prueba

STF-1.1 (K2) Explicar los objetivos, ventajas, desventajas, y limitaciones de la automatización de la prueba
STF-1.2 (K2) Comprender la relación entre las pruebas manuales y las automatizadas
STF-1.3 (K2) Identificar los factores de éxito técnico de un proyecto de automatización de prueba
STF-1.4 (K2) Comprender los riesgos y beneficios de utilizar Selenium WebDriver
STF-1.5 (K2) Explicar el lugar de Selenium WebDriver en la TAA
STF-1.6 (K2) Explicar las razones y propósitos para la recopilación de métricas en la automatización
STF-1.7 (K2) Comprender y poder comparar los objetivos de utilizar el conjunto de herramientas de Selenium (WebDriver, Selenium Server, Selenium Grid)

1.1 Descripción General de la Automatización de la Prueba

La automatización de pruebas comprende muchas cosas. Limitaremos nuestra discusión sobre la automatización en este programa de estudios para definir la automatización de pruebas como la ejecución automática de pruebas funcionales, diseñada al menos de alguna manera para simular que un ser humano está ejecutando pruebas manuales. Hay muchas definiciones diferentes (Vea el Programa de Estudios de Probador Certificado para Ingeniero de Automatización de Prueba de Nivel Avanzado de la ISTQB®); este es el más adecuado para este programa de estudios.

Mientras que la ejecución de la prueba es en gran medida automática, el análisis de prueba, el diseño de la prueba y la implementación de la prueba generalmente se realizan de forma manual. La creación y el despliegue de los datos utilizados durante las pruebas pueden estar parcialmente automatizados, pero a menudo se realizan manualmente. La evaluación del estado de paso/fallo para una prueba puede ser parte de la automatización (a través de un comparador integrado en la automatización), pero no siempre.

La automatización requiere el diseño, creación y mantenimiento de diferentes niveles de software de prueba, incluido el entorno en el que se ejecutarán las pruebas, las herramientas utilizadas, las librerías de códigos que proporcionan funcionalidad, los guiones de prueba y los arneses de prueba y las estructuras de registro e información para evaluar los resultados de la prueba. Dependiendo de las

herramientas utilizadas, la supervisión y el control de la ejecución de las pruebas pueden ser una combinación de procesos manuales y automatizados.

Pueden haber muchos objetivos que tratemos de lograr con la automatización de pruebas funcionales. Algunos de estos incluyen:

- Mejorar la eficiencia de las pruebas reduciendo el costo de cada realización de una prueba
- Probar más y otras cosas más que las que seríamos capaces de probar manualmente
- Reducir la cantidad de tiempo necesario para ejecutar pruebas
- Ser capaz de impulsar más las pruebas tempranas en el SDLC para reducir los defectos de tiempo en el código (es decir, mayúscula izquierda)
- Aumentar la frecuencia con la que se pueden ejecutar las pruebas

Como todas las tecnologías, existen ventajas y desventajas en el uso de la automatización de las pruebas.

No todas las ventajas se pueden obtener en todos los proyectos, ni todas las desventajas se producen en todos los proyectos. Mediante la atención estricta a los detalles y el uso de buenos procesos de ingeniería, es posible aumentar el bien y disminuir los malos resultados que pueden derivarse de un proyecto de automatización. Una cosa es cierta: una organización nunca ha construido ***accidentalmente*** un proyecto de automatización exitoso.

Las ventajas de la automatización pueden incluir:

- La ejecución de pruebas automatizadas puede ser más eficiente que ejecutarlas manualmente
- Realizar algunas pruebas que no se pueden realizar en absoluto (o fácilmente) de forma manual (como las pruebas de fiabilidad o eficiencia).
- Reducir el tiempo necesario para la ejecución de la prueba, lo que nos permite ejecutar más pruebas por compilación
- Aumentar la frecuencia con la que se pueden ejecutar las pruebas
- Liberar a los probadores manuales para que realicen pruebas manuales más interesantes y complejas (p. ej., pruebas exploratorias)
- Menos errores cometidos por probadores manuales aburridos o distraídos, especialmente cuando se repiten pruebas de regresión
- Las pruebas se pueden mover a un tiempo antes en el proceso (p. ej., ubicadas en la máquina de compilación continua para ejecutar las pruebas unitarias, de componentes y de integración automáticamente), proporcionando una retroalimentación más rápida sobre la calidad del sistema y eliminando los defectos del software antes
- Ejecución de pruebas fuera del horario de trabajo normal
- Mayor confianza en la compilación Las desventajas pueden incluir:

- Aumentos en los costos (incluidos los altos costos de inicio)
- Retrasos, costos y errores asociados con los probadores a medida que aprenden nuevas tecnologías
- En el peor de los casos, la complejidad puede ser abrumadora

- Crecimiento inaceptable en el tamaño de las pruebas automatizadas, posiblemente excediendo el tamaño del sistema a prueba (SUT)
- Habilidades de desarrollo de software necesarias para el equipo de prueba o para proporcionar un servicio al equipo de prueba
- Se requiere un mantenimiento considerable de las herramientas, entornos y recursos de prueba
- La deuda técnica es fácil de agregar, especialmente cuando se agrega programación adicional para mejorar el contexto y la racionalidad de las pruebas automatizadas, pero es difícil de reducir (como con todo el software)

- La automatización requiere todos los procesos y disciplinas del desarrollo de software
- Concentrarse en la automatización puede hacer que los probadores se olviden de la gestión de riesgos para el proyecto
- La paradoja del pesticida aumenta cuando se usa la automatización, ya que exactamente las mismas pruebas se ejecutan cada vez
- Los falsos positivos ocurren cuando los fallos de automatización a menudo no son fallos del SUT; sino que se deben a defectos en la propia automatización
- Sin una programación inteligente en las pruebas, las herramientas son de mentalidad literal y estúpidas; los probadores no lo son
- Las herramientas tienden a ser de un subproceso único, es decir, solo buscan un resultado para un evento, los seres humanos pueden descubrir lo que sucedió y determinar sobre la marcha si era correcto

Hay muchas limitaciones en un proyecto de automatización; algunas de estas pueden mitigarse mediante una programación inteligente y buenas prácticas de ingeniería, otras no. Algunas de estas limitaciones son técnicas, algunas son gerenciales. Estas limitaciones incluyen:

- Expectativas poco realistas de la gerencia que a menudo conducen la automatización en la dirección incorrecta
- Pensamiento a corto plazo que puede destruir un proyecto de automatización; solo pensando a largo plazo el programa de automatización puede tener éxito
- Se requiere madurez organizacional para tener éxito; la automatización basada en procesos de prueba deficientes solo ofrece malas pruebas más rápidas (a veces incluso más lentas)
- Algunos probadores están absolutamente contentos con las pruebas manuales y no desean automatizarlas
- Los oráculos de prueba de automatización pueden ser diferentes de los oráculos de prueba manual, lo que requiere que sean identificados
- No todas las pruebas pueden o deben ser automatizadas
- Se requerirán pruebas manuales (pruebas exploratorias, ataques de faltas, etc.)
- El análisis, el diseño y la implementación manual probablemente aún son manuales
- Los seres humanos encuentran la mayoría de los bugs; la automatización solo puede encontrar lo que está programado a encontrar y está limitada por la paradoja del pesticida
- Falso sentido de seguridad debido a la gran cantidad de pruebas automáticas que se ejecutan sin encontrar muchos bugs
- Problemas técnicos para el proyecto cuando se utilizan herramientas o técnicas de última generación
- Requiere cooperación con el desarrollo, lo que puede crear problemas organizativos

La automatización puede y a menudo tiene éxito. Pero ese éxito es solo el resultado de la atención al detalle, buenas prácticas de ingeniería y un trabajo muy difícil a largo plazo.

1.2 Pruebas Manuales vs. Automatizadas

Las primeras versiones de herramientas de automatización de la prueba fueron un fracaso abismal. Se vendieron muy bien, pero rara vez funcionaron como lo anunciado en la publicidad. Parte de la razón es que no modelaron bien las pruebas de software, ni comprendieron el papel del probador en el proceso de prueba.

La herramienta básica de grabación/reproducción (también conocida como captura/repetición) se vendió con alguna versión de las siguientes instrucciones:

> ***Conecte la herramienta a su sistema que va a probar. Encienda el interruptor para comenzar a grabar. Haga que el probador realice la prueba. Después de terminar, apague la herramienta. Generará un guión (en un lenguaje de programación) que hará exactamente lo que hizo el probador. Puede ejecutar ese guión cada vez que quiera realizar la prueba.***

A menudo, estos guiones no funcionaban incluso la primera vez que se ejecutaban. Cualquier cambio en el contexto de la pantalla, el tiempo, las propiedades de la GUI u otros cientos de elementos harían que el guión grabado fallara.

Para comprender la automatización de pruebas, debe comprender qué es un guión de prueba manual y cómo se usa el mismo. En su mínima expresión, un guión de prueba manual tiende a tener información en tres columnas.

La primera columna contendrá una tarea abstracta a realizar. Resumen, por lo que no es necesario cambiarlo cuando cambia el software real. Por ejemplo, la tarea “Agregar registro a la base de datos” es una idea abstracta. No importa en qué versión de qué base de datos se puede realizar esta tarea abstracta, suponiendo que un tester manual tenga el conocimiento del dominio para traducirlo en acciones concretas

La segunda columna le dice al probador qué datos usar para realizar la tarea. La tercera columna le dice al probador qué comportamiento esperar.

Tabla 1: Fragmento de Pruebas Manuales

1	Agregar un registro a la base de datos.	Nombre de pila: Gerry Apellido: Franklin Número de Seguro Social: 234-34-5678	Registro creado, Registro # devuelto
2	Buscar el nombre	Franklin, Gerry	Espere encontrarlo
3	Editar el registro	Ocupación: Abogado Ingresos: $ 125,000	Espere el diálogo de verificación del cambio
4	Compruebe el pedido de registro	Registro del paso 1	Espere un pedido válido
5	Etc.		

Las pruebas manuales han funcionado muy bien durante muchos años porque hemos podido crear estos guiones de prueba manuales. Sin embargo, el guión por sí solo no es lo que permite que se realicen las pruebas. Solo cuando el guión está en manos de un probador manual conocedor, podemos extraer el valor.

¿Qué le agrega el probador al guión que le permita que ejecute la prueba correctamente? **Contexto** y **Racionalidad.** Cada tarea se filtra a través del conocimiento del probador: "¿Qué base de datos? ¿Qué mesa? ¿Cómo realizo esta tarea con esta versión de base de datos?" El contexto conduce a algunas de las respuestas, la racionalidad hace que otras permitan que el probador tenga éxito en las tareas en el guión de prueba.

Una prueba automatizada ejecutada a través de una herramienta automatizada tiene un contexto limitado y racionalidad. Para realizar una tarea en particular (p. ej., "Agregar registro"), la herramienta debe estar colocada en la ubicación exacta para realizar la tarea. La suposición es que el último paso del caso de prueba automatizada los coloca en la ubicación exacta donde pueden realizar "Agregar registro". ¿Y si no lo hizo? ¡Qué lástima! No solo fallará la automatización, sino que el mensaje de error real probablemente no tendrá sentido, ya que el fallo real probablemente fue durante el paso anterior. Un probador que ejecute la prueba nunca tendrá este problema.

Del mismo modo, cuando se ejecuta una prueba manual, el resultado de una acción es verificado por el probador manual. Por ejemplo, si la prueba requirió que el probador abra un archivo (es decir, tarea = "Abrir archivo"), hay una variedad de cosas que pueden suceder. El archivo puede abrirse. Puede aparecer un mensaje de error, un mensaje de advertencia, un mensaje informativo, un mensaje de sincronización y una docena de cosas más. En cada caso, el probador manual simplemente lee el mensaje, aplica racionalidad y contexto al problema, y hace lo correcto.

Ese es el quid de la diferencia entre un guión automatizado y un guión manual. El guión de prueba manual solo agrega valor en las manos de un probador manual. Podríamos agregar líneas en el guión manual para ayudar al probador a entender cómo lidiar con un problema, pero la mayoría no lo hacemos porque los probadores manuales tienen un cerebro humano, buena inteligencia y experiencia, lo que les permite resolver el problema por sí mismos. Y, en el peor de los casos, pueden levantar su teléfono y preguntarle a un experto.

Considere algunas de las preguntas que el probador manual puede responder:

- ¿Cuánto tiempo debo esperar para que algo suceda?
- ¿Qué sucede si obtengo de vuelta un resultado que se puede arreglar?
- ¿Qué sucede si recibo una advertencia?
- ¿Qué sucede si se supone que debo esperar 5 segundos, pero realmente me tomó 5,1 segundos?

Ninguno de esos resultados puede manejarse de forma nativa con una herramienta de automatización por sí sóla. Una herramienta de automatización no tiene inteligencia; es solo una herramienta. Un guión automatizado, si se registra, tiene una comprensión limitada de lo que puede hacer si ocurre algo distinto de lo esperado. Un tamaño le sirve a todos: lanza un error.

Sin embargo, un guión automatizado puede tener inteligencia incorporada por el automatizador a través de la programación. Una vez que los autómatizadores entendieron que se podía agregar inteligencia al guión de automatización, pudimos comenzar a mejorar el trabajo de automatización. Mediante la programación, podemos capturar los procesos de pensamiento que tiene un probador manual, el contexto y la racionalidad, y comenzar a hacer que la automatización agregue más valor al completar la prueba con más frecuencia en lugar de fallarla anticipadamente. Sin embargo, tenga en cuenta que no existe tal cosa como un almuerzo gratis. La programación adicional también puede causar que más mantenimiento sea necesario más adelante.

No todas las pruebas deben ser automatizadas. Debido a que un guión automatizado requiere más análisis, más diseño, más ingeniería y más mantenimiento que un guión manual, debemos calcular el costo de crearlo. Y, una vez que hayamos creado el guión automatizado, tendremos que mantenerlo para siempre, y eso también debe tenerse en cuenta. Cuando el SUT cambia, a menudo los guiones automatizados deberán cambiarse simultáneamente.

Inevitablemente, encontraremos nuevas formas en las que un guión automatizado podría fallar: simplemente obtener un valor de retorno que nunca vimos hará que la automatización falle. Cuando eso suceda, necesitaremos cambiar, volver a probar y volver a implementar el guión. Estos generalmente no son problemas presentes con las pruebas manuales.

Un estudio de viabilidad comercial necesita hacerse antes, y mientras, automatizamos nuestra prueba. ¿Se devolverá el costo total de la automatización de esta prueba si se puede ejecutar N veces durante los próximos M meses? Muchas veces, podemos determinar que la respuesta es no. Algunas pruebas manuales se mantendrán porque no hay retorno de la inversión (ROI) positivo para automatizarlas.

Algunas pruebas manuales simplemente no se pueden automatizar porque el proceso de reflexión del probador manual es esencial para el éxito de la prueba. Aún se necesitan pruebas exploratorias, ataques de fallas y algunos otros tipos de pruebas manuales para el éxito de un esfuerzo de prueba.

1.3 Factores de Éxito

La automatización no tendrá éxito por casualidad. El éxito generalmente requiere:

- Un plan a largo plazo que se alinea con las necesidades comerciales.
- Gestión sólida e inteligente.
- Atención extrema al detalle.
- Madurez del proceso.
- Una arquitectura y marco que se formalizan.
- Capacitación apropiada.
- Niveles maduros de documentación.

La capacidad de automatizar a menudo se basa en la capacidad de prueba del sistema a prueba (SUT). Hay muchas ocasiones en las que las interfaces del SUT simplemente no son adecuadas para las pruebas. Obtener interfaces especiales o privadas (a menudo denominadas anzuelos) de los desarrolladores del sistema a menudo puede significar la diferencia entre tener éxito o fallar en la automatización.

Hay varios niveles diferentes de interfaz que podemos automatizar un el SUT a:

- El nivel de la GUI (a menudo frágil y propenso a fallos)
- El nivel de la IPA (utilizando interfaces de programación de aplicaciones que los desarrolladores ponen a disposición para uso público o protegido)
- Anzuelos privados (IPA que son específicamente para pruebas)
- Nivel de protocolo (HTTP, TCP, etc.)
- Nivel de servicio (SOAP, REST, etc.)

Tenga en cuenta que Selenium funciona a nivel de IU. Este plan de estudios contendrá sugerencias y técnicas para reducir la fragilidad y mejorar la usabilidad al realizar pruebas a este nivel.

Los siguientes son todos factores de éxito para la automatización:

- Administración que ha sido educada para comprender lo que es y lo que no es posible (las expectativas poco realistas de la administración son una de las principales razones por las que fallan los proyectos de automatización de pruebas de software)
- Un equipo de desarrollo para los SUTs que entiende la automatización y está dispuesto a trabajar con los probadores cuando sea necesario
- SUTs que están diseñados para la capacidad de prueba
- Desarrollar un estudio de viabilidad comercial a corto, mediano y largo plazo
- Tener las herramientas adecuadas para trabajar en el entorno y con los SUTs
- La capacitación correcta, que incluya disciplinas de prueba y desarrollo
- Tener una arquitectura y un marco bien diseñados y bien documentados para apoyar los guiones individuales (el Programa de Estudios de Ingeniero de Automatización de la Prueba Avanzada lo llama TAS (solución de automatización de pruebas))
- Tener una estrategia de automatización de pruebas bien documentada, financiada y aceptada por la administración
- Un plan formal y bien documentado para el mantenimiento de la automatización
- Automatización en el nivel de interfaz correcto para el contexto de las pruebas necesarias

1.4 Riesgos y beneficios de Selenium WebDriver

WebDriver es una interfaz de programación para desarrollar guiones avanzados de Selenium utilizando los siguientes lenguajes de programación:

- C#
- Haskell
- Java
- JavaScript
- Objetivo C
- Perl
- PHP
- Python
- R
- Ruby

Muchos de estos idiomas también tienen marcos de prueba de código abierto disponibles. Los navegadores compatibles con Selenium (y el componente necesario para realizar la prueba) incluyen:

- Chrome (chromedriver.exe)
- Internet Explorer (IEDriverServer.exe)
- Edge (MicrosoftWebDriver.msi)
- Firefox (geckodriver.exe)
- Safari (safaridriver)
- HtmlUnit (HtmlUnit driver)

Selenium WebDriver trabaja usando sus IPAs (Interfaz de Programación de Aplicaciones) para hacer llamadas directas a un navegador utilizando el soporte nativo diferente de cada navegador para la automatización. Cada navegador compatible trabaja un poco diferente.

Como con cualquier herramienta, existen beneficios y riesgos al usar Selenium WebDriver.
Muchos de los beneficios de los que una organización puede sacar provecho son los mismos que cualquier otra herramienta de automatización, que incluyen:

- La ejecución de la prueba puede ser consistente y repetible
- Es muy apropiado para pruebas de regresión
- Debido a que prueba en el nivel de IU, puede detectar defectos omitidos al realizar pruebas a nivel de IPA.
- Menor inversión inicial porque es de código abierto
- Funciona con diferentes navegadores, por lo que es posible realizar pruebas de compatibilidad
- Admite diferentes lenguajes de programación para que pueda ser utilizado por más personas
- Porque requiere una comprensión más profunda del código, es útil en equipos Ágiles.

Con las ventajas, van los riesgos. Los siguientes son riesgos que se presentan con el uso de Selenium

WebDriver.

- Las organizaciones a menudo se involucran tanto en las pruebas de la GUI que olvidan que la pirámide de prueba sugiere que se realicen más pruebas unitarias/de componentes.
- Cuando se prueba para un flujo de trabajo de IC (integración continua), esta automatización puede hacer que la compilación tarde más en completarse de lo deseable.
- Los cambios en la interfaz de usuario causan más daño a las pruebas de nivel del navegador que a las pruebas unitarias o a nivel de IPA
- Los probadores manuales son más eficientes en la búsqueda de errores que la automatización
- Las pruebas que son difíciles de automatizar pueden omitirse
- La automatización debe ejecutarse con frecuencia para obtener un ROI positivo (retorno de la inversión). Si la aplicación Web es bastante estable, es posible que la automatización no se pague por sí sóla.

1.5 Selenium WebDriver en la Arquitectura de Automatizacion de Prueba

La TAA (Arquitectura de Automatizacion de Prueba) según la define la ISTQB® en su Programa de Estudios de Ingeniero de Automatización de la Prueba Avanzada (TAE) es un conjunto de capas, servicios e interfaces de una TAS (Solución de Automatización de la Prueba).

LEl TAA consta de cuatro capas (Vea la ilustración a continuación):

- Capa de generación de prueba: admite el diseño manual y/o automático de casos de prueba
- Capa de definición de prueba: admite la definición e implementación de casos y/o series de prueba
- Capa de ejecución de prueba: admite tanto la ejecución de automatización como el registro/información de los resultados
- Capa de adaptación de prueba: proporciona objetos y códigos para interactuar con el SUT en varios niveles

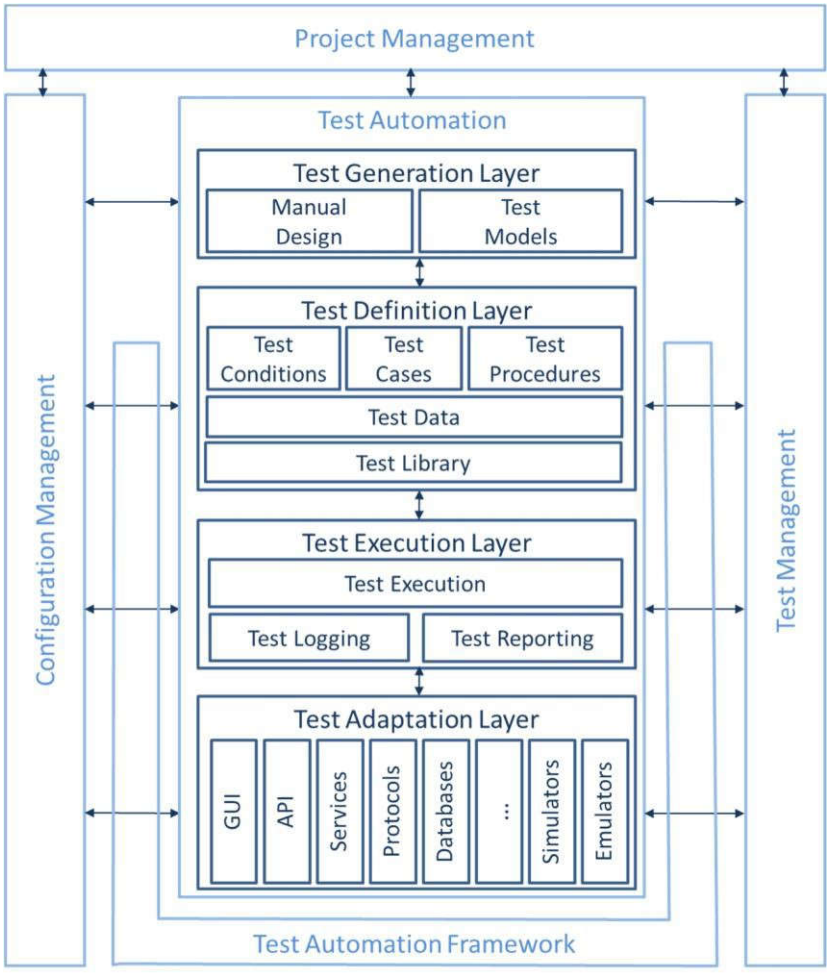

Figura 1: Una TAA genérica (del Programa de Estudios de TAE)

Selenium WebDriver se adapta a la capa de adaptación de prueba, proporcionando una forma programática para acceder al SUT a través de la interfaz del navegador.

La capa de adaptación de prueba facilita la separación del SUT (y sus interfaces) de las pruebas y series de prueba que queremos ejecutar contra el SUT.

Idealmente, esta separación permite que los casos de prueba sean más abstractos y separados del sistema que se está probando. Cuando el sistema sometido a prueba cambia, es posible que los casos de prueba no necesiten cambiar, suponiendo que se proporciona la misma funcionalidad, solo de una manera ligeramente diferente.

Cuando el SUT cambia, la capa de adaptación de prueba, en este caso, WebDriver, permite la modificación de la prueba automatizada, lo que le permite ejecutar el caso concreto de prueba concreta contra la interfaz modificada del SUT.

El guión automatizado que usa WebDriver utiliza eficazmente la IPA para comunicarse entre la prueba y el SUT de la siguiente manera:

- El caso de prueba hace una llamada para que se realice una tarea
- El guión llama a una IPA en WebDriver

- Esa IPA se conecta al objeto apropiado en el SUT
- El SUT responde (con suerte) según lo solicitado
- El éxito (o fallo) se comunica de vuelta al guión
- Si tiene éxito, se llama al siguiente paso en el caso de prueba en el guión

1.6 Propósitos para la Recopilación de Métricas en la Automatización

Hay un viejo dicho en la gestión empresarial, “Lo que se mide se hace”. Las mediciones son una de esas cosas que a muchos autómatizadores les gusta ignorar u ofuscar, porque a menudo son difíciles de cuantificar y comprobar.

El problema es que la automatización tiende a ser bastante costosa, tanto en recursos humanos como en recursos de herramientas. Hay poco o ningún retorno positivo de la inversión en el corto plazo; el valor llega a largo plazo. Años en lugar de meses. Pedirle a la administración una gran inversión en tiempo y recursos, y no hacer un estudio de viabilidad comprobable para ello (incluida la recopilación de métricas para proporcionar pruebas) es pedirles que crean en nosotros, que tengan fe. Desafortunadamente, para la alta gerencia, esas frases podrían ser obscenidades.

Deberíamos recopilar mediciones significativas donde podamos mostrar el valor de la automatización si queremos que el proyecto de automatización sobreviva a los gerentes obsesionados por los números. Programa de Estudios de Ingeniero de Automatización de la Prueba Avanzada (TAE) menciona varias áreas en las que las métricas serían útiles, que incluyen:

- Usabilidad
- Mantenibilidad
- Rendimiento
- Fiabilidad

s probable que muchas de las métricas que se mencionan en el programa de TAE sean más útiles para los equipos de automatización a gran escala en lugar de los pequeños proyectos de automatización que podrían usar Selenium WebDriver. Nos concentrarnos en proyectos más pequeños en esta sección. Si está trabajando en un proyecto de automatización grande y complejo, consulte el programa de TAE.

Recopilar métricas inapropiadas es probable que sea una distracción para un pequeño equipo que solo intenta que un proyecto salga adelante por primera vez. Un problema para identificar las métricas significativas que se recopilarán es que, aunque el tiempo de ejecución de la automatización de pruebas suele ser menor que el tiempo de prueba manual equivalente, el análisis, diseño, desarrollo, resolución de problemas y mantenimiento de pruebas automatizadas suele llevar mucho más tiempo que el mismo trabajo para pruebas manuales.

Intentar capturar métricas significativas para determinar el estudio de viabilidad (ROI) a menudo se convierte en un caso de comparación de manzanas con naranjas. Por ejemplo, puede tomar 1 hora para crear un caso de prueba manual específico. Treinta minutos para ejecutarla. Supongamos que planeamos ejecutar la prueba tres veces para esta versión. Necesitamos

2.5 2,5 horas para esa prueba. Mediciones simples para pruebas manuales.

Sin embargo, ¿cuánto costará automatizar esa misma prueba? Lo primero es determinar si hay algún valor en automatizar esa prueba en absoluto. Luego, suponiendo que decidamos que debe ser automatizada, averiguar cómo automatizarla, escribir el código necesario, depurar el código y garantizar que está haciendo lo que esperamos puede llevar varias horas. Cada prueba puede tener diferentes desafíos; ¿Cómo estimamos eso?

La prueba manual, al ser abstracta, probablemente no necesita ser cambiada cuando cambia la GUI del SUT. Es probable que el guión automatizado deba modificarse incluso para cambios menores en la GUI.
Podemos minimizarlo con una buena arquitectura y diseños de marcos, pero los cambios aún tomarán tiempo. Entonces, ¿Cómo estimamos cuántos serán los cambios esperados?

Cuando hay un fallo en una prueba manual, la solución de problemas suele ser sencilla. El analista de pruebas generalmente puede analizar lo que falló fácilmente para que puedan escribir el informe del incidente.

Sin embargo, un fallo en la automatización podría tomar más tiempo para solucionarlo (consulte la sección de registro a continuación).

¿Cómo se calcula cuántas veces puede fallar la prueba, especialmente cuando esos fallos a menudo se deben a la automatización misma, no a un fallo del SUT? Solo para dificultar el cálculo del rendimiento de la inversión, la automatización requiere una gran inversión inicial que no es necesaria para las pruebas manuales. Esta inversión incluye la compra de una o más herramientas, entornos adecuados para ejecutar las herramientas, la creación de la arquitectura y el marco para facilitar la automatización, la capacitación (o contratación) y otros costos fijos.

Esa inversión debe agregarse a cualquier cálculo de ROI que podamos hacer. Debido a que esos son costos fijos, lograr un ROI positivo probablemente signifique que necesitamos automatizar y ejecutar un gran número de pruebas. Eso requiere un nivel más elevado de escalabilidad que a su vez hará que la automatización cueste más.

Otro problema con muchas métricas de automatización es que a menudo deben estimarse en lugar de medirse directamente; esto a menudo significa que se distorsionen en un esfuerzo por demostrar que la automatización vale la pena.

Para un pequeño proyecto de inicio, aquí hay algunas métricas que pueden ser útiles:

- Costos fijos para hacer que la automatización esté lista y funcionando
- Esfuerzo de prueba de regresión que ha sido guardado por la automatización
- Esfuerzo empleado por el equipo de automatización que mantiene, respalda y extiende la automatización
- Cobertura
    - A nivel de prueba unitaria, mediante la cobertura de sentencias/decisiones
    - A nivel de cobertura de prueba de integración, interfaz o flujo de datos
    - A nivel de cobertura de prueba del sistema, requerimientos, prestaciones o riesgos identificados

    - Configuraciones probadas
    - Historias de usuario cubiertas (en Ágil)
    - Casos de uso cubiertos
- Configuraciones probadas cubiertas
- Número de ejecuciones exitosas entre los fallos
- Patrones de fallos de automatización (buscando la similitud de los problemas mediante el seguimiento de las causas raíz de los fallos)
- Número de fallos de automatización encontrados en comparación con las fallos del SUT encontrados por la automatización

Aquí está el punto final sobre las métricas. Trate de encontrar métricas sobre su proyecto que ayuden a explicar por qué el proyecto es importante y válido Negocie con la administración, asegurándose de que entiendan que probar el valor de la automatización es difícil desde el principio y, a menudo, lleva mucho tiempo realizar un progreso razonable hacia métricas significativas. Es tentador usar escenarios optimistas e imaginación creativa para probar el valor de la automatización. Quedarse atrapado en ello por la administración, sin embargo, podría ser el golpe de gracia al proyecto.

Cuando se realiza correctamente, un proyecto de automatización tiene buenas posibilidades de agregar valor a la organización, incluso si es difícil de probar.

1.7 El conjunto de herramientas de Selenium

Selenium WebDriver no es la única herramienta del Selenium estable. El ecosistema de fuente abierta de Selenium consta de cuatro herramientas, cada una de las cuales desempeña diferentes roles en la automatización de pruebas:

- Selenium IDE
- Selenium WebDriver
- Selenium Grid
- Servidor Autónomo de Selenium

Durante la larga historia de las herramientas de Selenium, existió la herramienta llamada Selenium RC, que implementó Selenium versión 1. Esta herramienta ya no se usa.

Selenium IDE es un complemento para los navegadores web Chrome y Firefox. Selenium IDE no funciona como una aplicación independiente. Su función principal es grabar y reproducir acciones de usuario en páginas Web.

Selenium IDE también permite que un automatizador inserte puntos de verificación durante la grabación. Los guiones grabados se pueden guardar en el disco como tablas HTML o exportarse a varios lenguajes de programación diferentes.

Las principales ventajas de Selenium IDE son su simplicidad y los localizadores de elementos razonablemente buenos. Su principal desventaja es la falta de variables, procedimientos e instrucciones de flujo de control; como tal, realmente no es útil para crear pruebas automatizadas

robustas. Selenium IDE se usa principalmente para registrar guiones provisionales (p. ej., para fines de depuración) o simplemente con el fin de encontrar localizadores.

Selenium WebDriver es principalmente un marco que permite que los guiones de prueba controlen los navegadores Web. Está basado en HTTP y ha sido estandarizado por W3C. ¡Este programa de estudios muestra las características básicas de este protocolo en caso de **Cápitulo 3 - Uso de Selenium WebDriver.** Selenium WebDriver tiene enlaces para muchos lenguajes de programación diferentes, p. ej. Java, Python, Ruby, C #. A lo largo de este programa de estudios, Python se usa para mostrar ejemplos de cómo usar diferentes objetos WebDriver y sus métodos.

El uso de librerías que implementan la IPA de WebDriver para diversos lenguajes de programación permite que los automatizadores de prueba combinen la capacidad de WebDriver para controlar los navegadores Web con la potencia de los lenguajes de programación generales. Esto permite que los automatizadores utilicen las bibliotecas de otros idiomas para crear marcos complejos de automatización de pruebas con registro, manejo de aserciones, subprocesos múltiples y mucho más.

Para lograr el éxito a largo plazo, la construcción de marcos de automatización de pruebas debe hacerse con cuidado y con buenos principios de diseño, tal como se describe en la sección:

1.5 Selenium WebDriver en la Arquitectura de Automatizacion de Prueba.

Algunos navegadores Web necesitan procesos WebDriver adicionales para iniciar nuevas instancias de prueba y controlarlas. El guión llama a los comandos de la librería, y la librería, a través de WebDriver, envía los comandos al navegador Web. Esto será discutido en el capítulo tres.

Otra herramienta que puede ser útil en un entorno de prueba es Selenium Grid. Permite ejecutar guiones de prueba en muchas máquinas con diferentes configuraciones. Permite la ejecución distribuida y simultánea de casos de prueba. La arquitectura de Selenium Grid es muy flexible. Se puede configurar para usar muchas máquinas físicas o virtuales con diferentes combinaciones de sistemas operativos y versiones de navegadores Web.

En el centro de Selenium Grid, hay un concentrador que controla otros nodos y actúa como un único punto de contacto para los guiones de prueba. Los guiones de prueba que ejecutan comandos de WebDriver no necesitan (en la mayoría de los casos) cambiarse para funcionar en diferentes sistemas operativos o navegadores Web.

La última herramienta del ecosistema de Selenium que mencionamos en este programa de estudios es el Servidor Autónomo de Selenium. Esta herramienta está escrita en Java y se entrega como un archivo jar que implementa las funciones de ejes y nodos para Selenium Grid. Esta herramienta debe iniciarse por separado (fuera de los guiones de prueba) y configurarse correctamente para desempeñar su función en el entorno de prueba.

Una descripción más detallada de Selenium Grid y del Servidor Autónomo de Selenium está más allá del alcance de este programa de estudios.

Capítulo 2 - Tecnologías de Internet para la Automatización de Pruebas para Aplicaciones Basadas en la Web

Palabras Clave

Selector de CSS, HTML, etiqueta, XML, XPath

Objetivos de Aprendizaje para Tecnologías de Internet para la Automatización de Pruebas para Aplicaciones Basadas en la Web

STF-2.1 (K3) Comprender y poder escribir documentos de HTML y XML
STF-2.2 (K3) Aplicar XPath para buscar documentos XML
STF-2.3 (K3) Aplicar localizadores de CSS para encontrar elementos de documentos HTML

2.1 Comprensión sobre HTML y XML

2.1.1 Comprensión sobre HTML

No sería exagerado decir que HTML (Lenguaje de Marcas de Hipertexto) abrió la red informática mundial. Un documento HTML es un archivo de texto plano que contiene elementos que especifican ciertos significados contextuales cuando se analiza el documento. Los elementos trabajan en conjunto para identificar cómo un navegador debe representar esas partes del documento. Esencialmente, HTML describe la estructura de una página Web semánticamente.

Quizás el beneficio más importante de usar HTML para especificar páginas Web es la aplicabilidad universal del lenguaje. Cuando se escribe adecuadamente, cualquier navegador en cualquier sistema de cómputo puede procesar la página correctamente.

El aspecto de la página puede cambiar de alguna manera, según el tipo de computadora (p. ej., PC contra un teléfono inteligente), el monitor, la velocidad de conexión a Internet y el navegador.

El crédito por inventar la World Wide Web se le otorga a Timothy Berners Lee. Mientras trabajaba para el CERN (Consejo Europeo de Investigación Nuclear) en 1980, propuso usar el hipertexto para permitir compartir y actualizar la información entre los investigadores. Luego, en 1989, implementó la primera

comunicación efectiva entre un cliente y servidor HTTP (Protocolo de transferencia de hipertexto), introduciendo la World Wide Web (red informática mundial) y cambiando el mundo tal como lo conocemos.

Los elementos HTML se introducen y a menudo están rodeados por etiquetas que están definidas por corchetes angulares como se ve a continuación:

```
<!DOCTYPE html>
<html>
   <head>
 <title> Page Title</title>
   </head>
   <body>
      <h1>This is a Heading</h1>
      <p>This is a paragraph</p>
   </body>
</html>
```

Figura 2: Ejemplo de HTML

Algunas etiquetas introducen directamente el contenido en la página que se está procesando (p. ej., **** dará como resultado que se coloque una imagen en la página). Otras etiquetas rodean y brindan información semántica sobre cómo se va a renderizar el elemento (como se ve arriba) para el elemento de encabezado **<h1>….</h1>**). Analizaremos las etiquetas a continuación.

Durante la mayor parte de la historia de la red informática mundial, la versión de HTML utilizada fue estable en la 4.01. Sin embargo, en el 2014, el organismo rector que controla HTML publicó la versión actual, HTML 5.

HTML es algo flexible, lo que permite algunas variaciones en la forma en que se usan las etiquetas (p. ej., algunas etiquetas pueden no tener una etiqueta de cierre). Esto ha ocasionado que algunos navegadores muestren páginas erráticamente. XML, que se analizará a continuación, es un lenguaje que es más restrictivo que HTML, requiriendo que cada página esté "bien formada" con cada etiqueta de apertura balanceada con una etiqueta de cierre. Cuando se escribe de una forma bien formada (es decir, todas las etiquetas de apertura se combinan con etiquetas de cierre), HTML es un subconjunto de XML.

La automatización con Selenium requiere una comprensión de las etiquetas HTML. Para automatizar cualquier GUI, el automatizador debe ser capaz de identificar cada control único en la pantalla que se está manipulando. La búsqueda a través de la página HTML permite que el automatizador detecte distintivamente los controles que se colocarán en la página renderizada por el navegador. Para encontrar los controles, el automatizador debe comprender el diseño y la lógica de la página HTML; eso se hace comprendiendo y analizando las etiquetas.

Los elementos HTML generalmente tienen una etiqueta de inicio y una de finalización. La etiqueta de finalización es la misma que la etiqueta de inicio, excepto que la etiqueta de finalización está precedida

por una barra, como se muestra a continuación:

** **Texto de párrafo** **

Algunos elementos se pueden cerrar en la etiqueta abierta; p. ej., el elemento de salto de línea vacío de la siguiente manera:

**
**

Una implementación menos estricta del salto de línea, que podría causar problemas para algunos navegadores sería:

**
**

Las siguientes son etiquetas que todo automatizador de Selenium debe entender.

Tabla 2: Etiquetas básicas de HTML

Etiqueta	Se usa para
<html> … </html>	Significa raíz del documento HTML
<!DOCTYPE>	Define el tipo de documento (no es necesaria en HTML 5)
<head> … </head>	Definición y metadatos para el documento
<body> … </body>	Define el contenido principal para el documento
…	Define un párrafo
 	Inserta un solo salto de línea
…	Define una sección en el documento
<-- … -->	Define un comentario (puede ser multilínea)

Las etiquetas de encabezado definen diferentes niveles de encabezados. El formato real del texto (tamaño, negrita, fuente) se puede especificar en hojas de estilo CSS.

Tabla 3: Etiquetas de encabezado

Etiquetas	Representa:
Heading 1	**Encabezado 1**
Heading 2	Encabezado 2
Heading 3	Encabezado 3
Heading 4	Encabezado 4
Heading 5	Encabezado 5
Heading 6	Encabezado 6

Los enlaces y las imágenes son fundamentales para las páginas de navegador bien diseñadas y son fáciles de crear con HTML.

*****link text*****

Esta combinación de símbolos comienza con una etiqueta de anclaje **<a …>** y termina con ****. Estos definen un hipervínculo en el que se puede hacer clic. El **href=”***URL***”** es un atributo que indica el destino del enlace. El texto del *enlace* entre las etiquetas representa el texto que aparecerá en el enlace al que se hará clic para llevar al usuario a la URL objetivo.

****

La etiqueta básica aquí, **<img…/>**, define una imagen que se colocará en este punto del documento. El atributo **src=”***pulpitrock.jpg***”** es la dirección de enlace de la imagen real que se mostrará. El otro atributo, **alt=“***Mountain view***”**, representa el texto que se mostrará si la imagen no se puede encontrar o mostrar.

Tabla 4: Listas y Tablas de Etiquetas

Etiqueta	Se usa para
** … **	**Define una lista desordenada (con viñetas)**
** … **	**Define una lista ordenada (enumerada)**
** … **	**Define un elemento de lista (para o)**
<table> … </table>	**Define una tabla HTML**
<tr> … </tr>	**Define una fila de tabla**
<th> … </th>	**Define el encabezado de columna para una tabla**
<td> … </td>	**Define una celda de datos de tabla**
<tbody> … </tbody>	**Agrupa el contenido del cuerpo en una tabla HTML**
<thead> … </thead>	**Define el encabezado de una tabla HTML**
<tfoot> … </tfoot>	**Define el pie de una tabla HTML**
<colgroup> … </colgroup>	**Agrupa columnas de tabla para formatear**

Las listas son simples de definir y representar. El siguiente código HTML representará la lista que le sigue.

```
<!DOCTYPE html>
<html>
<head>

<body>

<h4>An Unordered List containing an Ordered List</h4>
<ul>
      <li>Coffee</li>
    <li>Tea</li>
    <ol>
      <li>Oolong</li>
         <li>Black</li>
         <li>Earl Grey</li>
    </ol>
    <li>Milk</li>
</ul>
</body>
</html>
```

Figura 3: Ejemplo de Lista HTML

An Unordered List containing an Ordered List

- Coffee
- Tea
    1. Oolong
    2. Black
    3. Earl Grey
- Milk

Figura 4: Lista desordenada que contiene una lista ordenada

Las tablas también son sencillas de crear y renderizar. Los datos resultantes de la prueba a menudo se devuelven en tablas, por lo que los automatizadores a menudo trabajan con ellos. El siguiente código representará la tabla como se muestra debajo del código.

```
<!DOCTYPE html>
<html>
     <head>
          <style>
               table, th, td {
                   border: 1px solid black;
               }
          </style>
     </head>
     <body>

     <table>
          <tr>
               <th>Year</th>
               <th>Car Payment</th>
          </tr>
          <tr>
               <td>2017</td>
               <td>$3,780</td>
          </tr>
          <tr>
               <td>2018</td>
               <td>$2,905</td>
          </tr>
          <tr>
               <td>2019</td>
               <td>$4,812</td>
          </tr>
          <tr>
               <td>2020</td>
               <td>$1,790</td>
          </tr>
     </table>
     </body>
</htm
```

Figura 5: Código de generación

de tablas Tabla 5: Tabla

renderizada

Año	Pago del auto
2017	$ 3,700
2018	$ 2,905
2019	$ 4,812
2020	$ 1,790

Los formularios de HTML y los controles asociados se utilizan para recolectar las entradas de datos de los usuarios. A continuación se muestran las etiquetas necesarias para representar los formularios y los controles sobre ellos. Los usuarios de Selenium WebDriver a menudo tienen que interactuar con estos formularios y controles para automatizar sus pruebas.

Las siguientes etiquetas se utilizan para crear controles en la pantalla.

<form> … </form>

Define un formulario HTML para la entrada del usuario.

<input>

Defines un control de entrada. El tipo de control se define por el tipo de atributo **type=**. Los tipos posibles incluyen texto, radio, casilla de verificación, enviar, etc. Por ejemplo, las siguientes líneas se mostrarán de la manera siguiente:

```
<form action="/action_page.php">
First name: <input type="text" name="FirstName" value="Mortey"><br>
Last name: <input type="text" name="LastName" value="Moose"><br>
<input type="submit" value="Submit">
</form>
```

Figura 6: Listado de formularios

Nombre de pila:	Morty
Apellido:	Moose
Enviar	

Figura 7: Campos de Entrada del Listado de Formularios

<textarea> … </textarea>

Define un control de entrada de múltilínea. El área de texto puede contener un número ilimitado de caracteres. Por ejemplo, las siguientes líneas se mostrarán de la manera siguiente:

```
<textarea rows="4" cols="50">
Selenium allows you to automate browsers with
maximum return and minimum effort.
</textarea>
```

Figura 8: Código HTML para Control de Entrada de Múltilínea.

Selenium le permite automatizar navegadores con devolución máxima y esfuerzo mínimo

Figura 9: Control de Entrada de Múltilínea.

<button>

Define un botón cliqueable.

<select> … </select>

Define una lista desplegable. Cuando se utiliza con las etiquetas **<option>…</option>**, el autor puede definir una lista desplegable y completarla de la manera siguiente:

```
<select>
  <option value="volvo">Volvo</option>
  <option value="saab" >Saab</option>
  <option value="mercedes">Mercedes</option>
  <option value="audi">Audi</option>
</select>
```

Figura 10: Código para el Control de Selección

El código anterior mostrará el siguiente control desplegable:

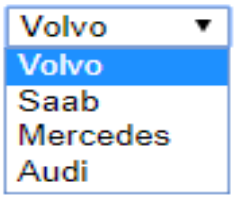

Figura 11: Una lista desplegable.

<fieldset> … </fieldset>

Estas etiquetas le permiten al autor agrupar elementos relacionados en un formulario. Cuando se usa con la etiqueta,
<legend>, dibuja un recuadro con nombre alrededor de los controles que se consideran relacionados como se muestra:

```
<fieldset>
   <legend>Child 1:</legend>
     Name: <input type="text"><br>
     Email: <input type="text"><br>
     Date of birth: <input type="text">
 </fieldset>
```

Figura 12: Código de conjunto de campos

El código anterior generará el siguiente conjunto de controles:

Figura 13: Elementos agrupados en un formulario

2.1.2 Comprensión sobre XML

XML (eXtensible Markup Language/*Lenguaje de Marcado Extensible*) es un lenguaje de marcado que se usa para definir reglas para formatear documentos de una manera legible por la máquina, pero que también es altamente legible para seres humanos. XML fue diseñado para acentuar la simplicidad y la usabilidad en la red informática mundial. Mientras se usa en documentos, XML permite la representación de estructuras de datos arbitrarias que pueden diseñarse sobre la marcha.

HTML fue diseñado para mostrar datos con un enfoque específico sobre cómo se ven los datos. XML fue diseñado para ser una herramienta independiente del software y del hardware que se puede utilizar para transportar y almacenar datos en un formato legible.

Las etiquetas XML no están predefinidas, como las etiquetas HTML. En cambio, las etiquetas son especificadas por el creador del documento XML para cualquier norma que quieran usar. El formato de las etiquetas es muy parecido al HTML. Por ejemplo, aquí hay un conjunto de campos en XML:

```
<?xml version="1.0" ?>
<note>
      <date>2018-06-12</date>
      <hour>10:30</hour>
       <to>Francis</to>
       <from>Morrow</from>
       <body>Please pick me up this weekend!</body>
</note>
```

Figura 14: Ejemplo de Código XML

Tenga en cuenta que cada etiqueta de apertura, **<from>**, tiene una etiqueta de cierre coincidente, **</from>**. La construcción total se llama elemento.

Las secciones se pueden incrustar en otras secciones como se muestra. Un documento XML siempre forma una estructura de árbol. El primer elemento en la figura anterior muestra que este es un documento XML.

Además de las etiquetas, XML admite atributos que brindan información adicional sobre el elemento al que están asociados. Un atributo consiste en un par de términos separados por un signo igual. Por ejemplo:

<person *gender=”female”*>

El atributo está dentro de los corchetes del elemento. En vez de usar un atributo, la misma información también se puede usar como un elemento. Por ejemplo, los siguientes dos ejemplos contienen exactamente la misma información.

```
<person gender="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>

<person>
  <gender>female</gender>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

Figura 15: Atributo vs. Elemento

Los atributos no son tan flexibles como los elementos. Por ejemplo, W3C destaca los siguientes puntos, el grupo que controla el estándar XML:

- los atributos no pueden contener valores múltiples (los elementos pueden)
- los atributos no pueden contener estructuras de árbol (los elementos pueden)
- los atributos no son fácilmente expandibles (para futuros cambios)

Las diferentes computadoras almacenan datos de diferentes maneras; la mayoría de esas formas son incompatibles. XML permite que estas diferentes computadoras compartan datos porque los datos XML se almacenan en formato de texto plano. No hay necesidad de complejos apretones de manos entre computadoras porque pueden comunicarse utilizando archivos de texto.

XML separa los datos de la forma en que se presentan. Por lo tanto, los mismos datos XML pueden presentarse de la forma que el autor desee.

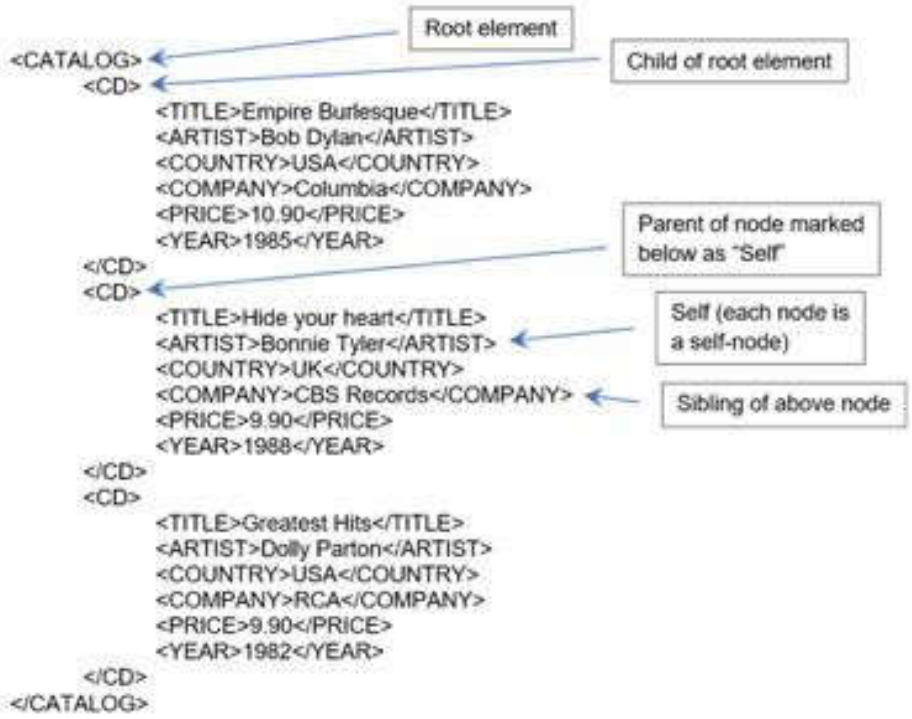

Figura 16: Ejemplo de Base de datos XML (Parcial)

Como XML siempre describe un formato de árbol, podemos definir relaciones específicas entre los elementos. Estas relaciones se pueden definir de la siguiente manera:

- El nodo **actual** es cualquier nodo arbitrario que elijamos. Todas las demás relaciones derivan del nodo actual. En la diapositiva siguiente, hemos elegido "Bonnie Tyler" como el nodo actual. Entonces, todas las relaciones se refieren a ese nodo.
- Cada nodo puede identificarse a sí mismo como **auto.**
- Un nodo **padre** es un nivel más alto en la jerarquía que el nodo actual. Cada nodo y atributo de elemento tiene exactamente un padre (a excepción del elemento raíz).
- Un nodo **hijo** siempre está un nivel por debajo de su padre en la jerarquía.
- Un nodo **hermano** está al mismo nivel que el nodo actual.
- Un nodo **ancestro** es uno en una ruta directa desde el nodo actual hasta su padre, abuelo, bisabuelo, etc.
- Un nodo **descendiente** es uno en una ruta directa hacia abajo desde el nodo actual en la jerarquía (es decir, un hijo, un hijo de un hijo, etc.)

2.2 XPath y Búsqueda de Documentos HTML

Como se mencionó anteriormente, para automatizar con Selenium, un automatizador debe ser capaz de localizar cualquier objeto o elemento dado en un documento XML. Una de las maneras más poderosas de localizar elementos específicos es utilizar XPath.

XPath utiliza expresiones de ruta para identificar y navegar nodos en un documento XML. Como HTML es un subconjunto de XML, XPath también se puede usar para buscar documentos HTML. Estas expresiones de ruta están

relacionadas con las expresiones de ruta que pueden usarse con un sistema de archivos de computadora tradicional que le permiten al usuario recorrer una jerarquía de carpetas.

Las siguientes expresiones seleccionarán nodos:

Tabla 6: Expresiones XPath

Expresión	Descripción
TheNode	Selecciona todos los elementos con el nombre “TheNode”
/	Selecciona desde el elemento raíz
//	Devuelve descendientes del elemento actual
.	Devuelve el elemento actual
..	Selecciona el padre del elemento actual
@	Selecciona un atributo del elemento actual

A continuación se muestra un ejemplo de documento XML y algunos ejemplos del uso de XPath.

```
<?xml version="1.0" encoding="UTF-8"?>
<Magazines>
    <Magazine>
        <title lang="en"">Time</title>
        <price>9.99</price>
     </Magazine>
     <Magazine>
        <title lang="en"">Newsweek</title>
        <price>8.95</price>
     </Magazine>
</Magazines>
```

Figura 17: Ejemplo de base de datos XML (Parcial)

A continuación se encuentran algunas expresiones de ruta y los resultados de ellas.

Tabla 7: Expresiones de Ruta

Expresión de Ruta	Resultado
Magazines	Selecciona todos los nodos con el nombre “Magazines”
/Magazines	Selecciona desde el elemento raíz “Magazines”
Magazines/Magazine	Selecciona todos los elementos Magazines de Magazines
//Magazines	Selecciona todos los elementos Magazines en el documento
Magazines/Magazine	Selecciona todos los elementos Magazine que son descendientes de Magazines
//@lang	Selecciona todos los atributos llamados lang

Los predicados se utilizan para encontrar un elemento específico o un elemento que contiene un valor específico. Los predicados están rodeados por corchetes [] y aparecen después del nombre del elemento.

Tabla 8: Expresiones de Ruta que Utilizan Predicados

Expresión de Ruta	Resultado
/Magazines/Magazine[1]	Selecciona el primer elemento Magazine
/Magazines/Magazine[last()]	Selecciona el último elemento hijo Magazine
/Magazines/Magazine[last()-1]	Selecciona el penúltimo elemento Magazine
//title[@lang]	Selecciona todos los elementos de título con el atributo “lang”
//title[@lang=’en’]	Selecciona todos los elementos de título con el atributo lang=en

XPath tiene comodines que se pueden usar para seleccionar nodos XML basados en criterios específicos.

Tabla 9: Comodines en XPath

Comodín	Descripción
*	Coincide con cualquier nodo de elemento
@*	Coincide con cualquier nodo de atributo
Node()	Coincide con cualquier nodo de cualquier tipo

Existe una gran diversidad de operadores que se pueden usar en expresiones XPath.

Tabla 10: Operadores en XPath

Operador	Descripción	Ejemplo
|	Selecciona múltiples rutas	//Magazine | //CD
+	Adición	2 + 2
-	Sustracción	5 – 3
*	Multiplicación	8 * 8
div	División	14 div 2
mod	Módulo (resto después de la división)	7 mod 3
=	Igual	precio=4.35
!=	No es igual	precio!=4.35
<	Menor que	precio<4.35
<=	Menor que o igual a	precio<=4.35
>	Mayor que	precio>4.35
>=	Mayor que o igual a	precio>=4.35
or	O	precio>3.00 o lang=”en”
and	Y	precio>3.00 y lang=”en”
not	Invertir	no lang=”en”
||	Concatenación de cadenas	“en” || “glish”

Del mismo modo, hay muchas funciones útiles de manipulación de cadenas disponibles en XPath. Las funciones se pueden llamar con el prefijo de espacio de nombre “fn:”. Sin embargo, dado que el prefijo predeterminado de espacio de nombre es “fn:”, las funciones de cadena normalmente no necesitan el prefijo.

Tabla 11: Funciones de Cadena en XPath

Nombre	Descripción
string(arg)	Devuelve el valor de la cadena del argumento
substring(str, start, len)	Devuelve una subcadena de una cadena de longitud "len" que comienza en "start"
string-length(str)	Devuelve la longitud de la cadena. Si no se ha pasado argumento, devuelve la longitud del nodo actual
compare(str1, str2)	Devuelve -1 si str1 < str2, 0 si las cadenas son iguales, +1 si str1 > Str2
concat(str1, str2, …)	Devuelve una concatenación de todas las cadenas ingresadas
upper-case(str)	Convierte el argumento de cadena a mayúscula
lower-case(str)	Convierte el argumento de cadena a minúscula
contains(str1, str2)	Devuelve VERDADERO si str1 contiene str2
starts-with(str1, str2)	Devuelve VERDADERO si str1 comienza con str2
ends-with(str1, str2)	Devuelve VERDADERO si str1 finaliza con str2

Un enlace útil para probar expresiones XPath se puede encontrar en:

https://www.freeformatter.com/xpath-tester.html

2.3 Localizadores de CSS

Hay muchas opiniones sobre si CSS es un lenguaje de programación o no. CSS significa hojas de estilo en cascada y se utiliza principalmente para prescribir cómo deberían representarse los diversos elementos HTML en un conjunto de documentos HTML, en la pantalla, en papel o en otros medios. Las hojas de estilo CSS externas se almacenan en archivos CSS.

HTML es un lenguaje de marcado y CSS es un lenguaje de hojas de estilo. Si bien HTML y CSS le dan al autor herramientas muy poderosas para mostrar materiales, la mayoría de los expertos no los consideran lenguajes de programación reales.

Sin embargo, cuando se aplica a las pruebas de Selenium, CSS es muy útil para encontrar elementos HTML, por lo que las pruebas del navegador se pueden automatizar.

CSS se puede utilizar en documentos HTML de tres maneras diferentes:

1. Una hoja de estilo externa: cada página HTML debe incluir una referencia al archivo de hoja de estilo externo dentro del elemento **<link>** que va dentro de la sección **<head>**
2. Una hoja de estilo interna: cuando una sola página HTML debe tener un estilo único; los estilos se definen dentro del elemento **<style>**, dentro de la sección **<head>** del documento
3. Un estilo en línea: se aplica a un elemento específico y se agrega directamente al elemento como un atributo

Cuando se definen múltiples estilos CSS para el mismo elemento, se usará el valor de la última hoja de estilo de lectura. Por lo tanto, el orden en que se utilizará un estilo se definirá (comenzando en la parte superior) de la siguiente manera:

1. Estilo en línea (como un atributo dentro de un elemento HTML)
2. Hojas de estilo externas e internas definidas en la sección cabecera
3. El valor predeterminado del navegador

Un conjunto de reglas para CSS consta de selectores y bloques de declaración como sigue:

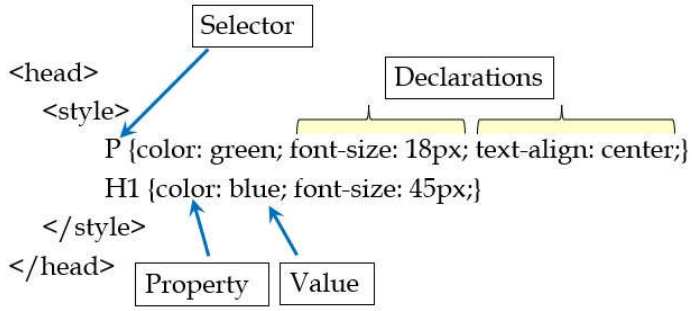

Figura 18: Conjunto de reglas para CSS

Cada selector apunta a un elemento HTML que usted desea estilizar. Un bloque de declaración contiene una o más declaraciones terminadas por punto y coma. Cada declaración incluye un nombre de propiedad CSS y un valor separado por dos puntos. Los bloques de declaración están rodeados de llaves.

Los selectores de CSS se pueden emplear para buscar elementos en un documento HTML en función del nombre del elemento, ID, clase, atributo u otros especificadores. En muchos casos, XPath se puede usar de la misma manera para encontrar ciertos elementos, como se muestra en la tabla siguiente:

Tabla 12: CSS vs. Selectores XPath

CSS	XPath	Resultado
div.even	//div[@class="even"]	Elementos *div* con un atributo *class="even"*
#login	//*[@id="login"]	Un elemento con *id="login"*
*	//*	Todos los elementos
input	//input	Todos los elementos de entrada
p,div		Todas las *p* y todos los elementos *div*
div input	//div//input	Todos los elementos *de entrada* dentro de todos los elementos *div*
div > input	//div/input	Todos los elementos *de entrada* que tienen el elemento *div* como el padre
br + p		Selecciona todos los elementos *p* que se colocan inmediatamente después del elemento *br*
p ~ br		Selecciona todos los elementos *p* que se colocaninmediatamente antes del elemento *br*

Como podría esperarse, los selectores de CSS también trabajan con atributos.

Tabla 13: Selectores de CSS para atributos

CSS	Resultado
[lang]	Todos los elementos con el atributo *lang*
[lang=en]	Todos los elementos con el atributo *lang* de exactamente *en*
[lang^=en]	Todos los elementos con el atributo *lang* que comienzan con la cadena *en*
[lang | =en]	Todos los elementos que tienen el atributo *lang* igual a *en* o comenzando con la cadena *en* seguida de un guión
[lang$=en]	Todos los elementos que tienen el atributo *lang* que termina con la cadena *en*
[lang~=en]	Todos los elementos que tienen el atributo *lang* cuyo valor es una lista de palabras separadas por espacios en blanco, una de las cuales es exactamente la cadena *en*
[lang*=en]	Todos los elementos que tienen el atributo *lang* que contiene una cadena *en*

Cuando se trata de elementos de formulario, hay una variedad de selectores de CSS disponibles:

Tabla 14: Selectores de CSS para elementos de formulario

CSS	Resultado
:checked	Selecciona todos los elementos marcados (para casillas de verificación, botones de radio, y opciones que se cambian a un estado *activado*
:default	Selecciona cualquier elemento de formulario que sea el predeterminado entre un grupo de elementos relacionados
:defined	Selecciona todos los elementos que se han definido
:disabled	Selecciona todos los elementos que se han desabilitado
:enabled	Selecciona todos los elementos que se han habilitado
:focus	Selecciona el elemento actualmente con foco
:invalid	Selecciona cualquier elemento de formulario que no se valide
:optional	Selecciona elementos de formulario que no tienen un conjunto de atributos *requerido*
:out-of-range	Selecciona cualquier elemento *de entrada* cuyo valor actual está fuera de los atributos *mín* y *máx*
:read-only	Selecciona elementos que no son editables por el usuario
:read-write	Selecciona elementos que son editables por el usuario
:required	Selecciona elementos de formulario que tienen un conjunto de atributos requerido
:valid	Selecciona elementos de formulario que se validan con éxito
:visited	Selecciona los enlaces que un usuario ya ha visitado

Finalmente, hay una variedad de selectores de CSS que pueden ser útiles para un automatizador, que incluyen:

Tabla 15: Selectores de CSS Útiles para la Automatización

CSS	Resultado
:not(<selector>)	Selecciona los elementos que no coinciden con el selector especificado
:first-child	Selecciona todos los elementos que son primeros hijos de sus elementos padres
:last-child	Selecciona todos los elementos que son últimos hijos de sus elementos padres
:nth-child(<n>)	Selecciona todos los elementos que son *enésimos* hijos de sus elementos padres
:nth-last-child(<n>)	Selecciona todos los elementos que son *enésimos* hijos de sus elementos padres contados a partir del último hijo
div:nth-of-type(<n>)	Selecciona todos los elementos que son *enésimos div* hijos de sus elementos padres

Comprender HTML, CSS, XML y XPath es esencial para tener éxito con Selenium. Lo que hemos presentado aquí es solo la punta del iceberg.

El W3C (World-Wide-Web Consortium) es una comunidad internacional que trabaja con muchas organizaciones y el público para definir y desarrollar estándares Web, incluidos XML y XPath.

Aunque no está conectado con W3C, hay un excelente conjunto de tutoriales sin costo en un sitio Web llamado W3Schools. Los tutoriales incluyen HTML, XML, XPath, CSS y cubren muchas otras tecnologías que son útiles al realizar las pruebas de Selenium.

W3Schools y todos sus tutoriales son propiedad de los derechos de autor de Refsnes Data. A4Q no tiene conexión alguna con W3Schools of Refsnes Data. El siguiente enlace se conectará a los tutoriales:

https://www.w3schools.com/default.asp

Cápitulo 3 - Uso de Selenium WebDriver

Palabras Clave

Objetivos de Aprendizaje por Utilizar Selenium WebDriver

STF-3.1 (K3) Utilizar los mecanismos adecuados de registro y gestión de información

STF-3.2 (K3) Navegar a diferentes URL usando los comandos de WebDriver

STF-3.3 (K3) Cambiar el contexto de la ventana en los navegadores web usando los comandos de WebDriver

STF-3.4 (K3) Recopilar Capturas de Pantalla de Páginas Web utilizando comandos de WebDriver

STF-3.5 (K4) Localizar Elementos de la GUI usando varias estrategias

STF-3.6 (K3) Obtener el estado de los elementos de la GUI utilizando los comandos de WebDriver

STF-3.7 (K3) Interactuar con elementos de la GUI utilizando comandos de WebDriver

STF-3.8 (K3) Interactuar con los Mensajes de Usuario en los Navegadores Web utilizando los comandos de WebDriver

3.1 Mecanismos de registro y gestión de información

Los guiones de prueba automatizados son programas de software que ejecutan comandos en el SUT. Al hacer esto, simulan que los humanos realizan pruebas usando el teclado y el mouse. Dentro del TAS debe existir un mecanismo que implemente la capa de ejecución de prueba. Una forma de implementar dicho mecanismo es escribir guiones de automatización de prueba desde cero. Tales guiones se pueden ejecutar como cualquier otro guión de Python.

En lugar de crear completamente dichos guiones, podemos aprovechar las librerías de pruebas unitarias existentes que pueden ejecutar pruebas e informar sus resultados, p. ej., unittest o pytest. En este programa de estudios, hemos elegido Pytest como nuestra librería de ejecución de prueba.

Pytest es un marco de prueba que hace que sea más fácil escribir pruebas para Python, y más para nuestro interés, para Selenium WebDriver usando Python. Pytest hace que sea fácil escribir pruebas pequeñas, pero se amplía para admitir la escritura de conjuntos de automatización complejos.

Cuando se invoca, Pytest ejecuta todas las pruebas en el directorio actual o sus subdirectorios. Busca específicamente todos los archivos que coincidan con los patrones: “test__*.py” o :*__test.py” y luego los ejecuta.

Cuando se ejecuta sin indicadores (**pytest**), pytest simplemente ejecuta todas las pruebas que se encuentran en los directorios debajo de él. Alternativamente, se pueden establecer indicadores para modificar su comportamiento de la siguiente manera:

- (**pytest -v**) Modo detallado que muestra los nombres completos de las pruebas en lugar de solo un período
- (**pytest -q**) Modo silencioso que muestra menos salida
- ***(pytest --html=report.html)***: Ejecuta prueba (s) con un informe en el archivo HTML

Las pruebas pueden marcarse para ser tratadas de una manera especial. Por ejemplo, cuándo coloca

(**@pytest.mark.skip)** antes de una definición de prueba, pytest no ejecutará la prueba. Si coloca (**@pytest.mark.xfail**) antes de una definición de prueba, informa al motor en tiempo de ejecución que se espera que falle la prueba. Tales pruebas, si fallan cuando se ejecutan, no se informarán de la misma manera que las pruebas que fallan cuando se espera que pasen.

Cuando un probador manual ejecuta una prueba con guiones y falla, el probador tiene una idea bastante buena de lo que sucedió. Saben exactamente qué pasos llevaron al fallo, exactamente qué sucedió durante el fallo, qué datos se usaron y cómo se vio el fallo. Cuando se ejecuta una prueba exploratoria, mientras que el probador no tiene un guión para los pasos exactos que se tomaron, tienen una teoría general de la ejecución y la capacidad limitada para retroceder y ver por qué ocurrió un fallo.

En automatización, casi nada de eso es verdad.

A menudo, en la automatización, el mensaje de error que registra la herramienta es totalmente insuficiente para que la persona que está revisando una ejecución fallida comprenda lo que sucedió. Con frecuencia, el mensaje de error no tiene contexto; el error registrado puede no tener nada que ver con el fallo real que detuvo la ejecución de la prueba.

Por ejemplo, supongamos que ocurre una falla en el paso ***N***. Dependiendo de la forma en que se produjo el fallo, el resultado puede significar que la interfaz del SUT no se queda en el estado correcto. A medida que el guión intenta ejecutar el paso ***N + 1***, el paso falla porque el SUT no está en el estado correcto para realizar el paso. El registro informa que fue el paso ***N + 1*** el que falló.

La solución de problemas luego comienza incorrectamente. Dado que el analista de prueba que ejecutó la prueba automatizada (generalmente en modo por lotes) puede no tener conocimiento directo de la prueba real que se ejecutó, intentar solucionarlo para determinar si es un fallo del SUT real o un fallo de automatización puede llevar una gran cantidad de hora.

No tiene que ser así en la automatización. Un buen registro puede ayudar a un analista de pruebas a determinar rápidamente si el fallo fue causado por el SUT o no. Es esencial que un automatizador preste atención al registro de la prueba. Un buen registro puede marcar la diferencia entre un proyecto de automatización fallido y uno que agrega mucho valor a la organización.

El registro es una forma de rastrear eventos que ocurren durante la ejecución. El automatizador puede agregar llamadas de registro a un guión para registrar información que facilite la comprensión de la ejecución. El registro puede realizarse antes de que se realice un paso y después de que se haya realizado un paso, y puede incluir los datos que se usaron en el paso y el comportamiento que ocurrió como resultado del paso. Cuantos más registros se hagan, mejor podrá comprender el resultado de la prueba.

En algunos casos, cuando está probando software crítico para la seguridad o de misión crítica, es posible que se requiera un registro detallado con fines de auditoría.

El registro puede ser condicional. Es decir, los datos pueden guardarse en una ubicación intermedia y solo en el registro formal si ocurre un fallo o si se necesita un registro completo de solución de problemas. El registro de cada paso puede no ser deseable cuando la prueba se ejecuta como se esperaba; sin embargo, en caso de fallo, esa información de registro puede ahorrarle horas de solución de problemas registrando, paso a paso, exactamente qué sucedió durante la ejecución y qué datos se usaron.

No todos los proyectos de automatización necesitan registros tan sólidos. A menudo, un proyecto de automatización comenzará con uno o dos automatizadores que crearán y ejecutarán sus

propios guiones. En tal caso, el registro descrito anteriormente se puede ver como exceso. Sin embargo, lo que debe tenerse en cuenta es que cuando los pequeños proyectos de automatización tienen éxito, se garantiza que la administración querrá más: mucha más automatización. Cuanto mayor sea el éxito, mayor será la demanda de más.

Esto significa que, con el tiempo, el proyecto alcanzará un tamaño tal que ***se requerirá*** el registro descrito. En ese punto, las pruebas existentes tendrían que ser adaptadas con un mejor registro. Es más efectivo y eficiente comenzar a iniciar sesión de manera robusta desde el principio.

Python tiene un conjunto muy sólido de recursos de registro que se pueden usar con WebDriver. En cualquier punto de un guión automatizado, el automatizador puede agregar llamadas de registro para informar cualquier información
deseada. Esto puede incluir información general para el seguimiento posterior de la prueba (p. ej., "Estoy a punto de hacer clic en el botón XYZ"), advertencias sobre algo que sucedió que no llega al nivel de un fallo (p. ej., "Abrir archivo <ABC> llevó más tiempo de lo esperado") o errores reales, generando una excepción que desencadena el final de los eventos de prueba, como limpiar el entorno y pasar a la siguiente prueba.

La librería de registro de Python tiene cinco niveles diferentes de mensajes que pueden guardarse. Desde el nivel más bajo al más alto:

- DEBUG: para diagnosticar problemas
- INFO: para la confirmación de que las cosas funcionaron
- WARNING: sucedió algo inesperado, se produjo un problema potencial
- ERROR: ocurrió un problema grave
- CRITICAL: ocurrió un problema crítico

Cuando se imprime un registro en la consola o en un archivo, el nivel del mensaje se puede establecer en una de esas cinco configuraciones (o se pueden crear configuraciones personalizadas), lo que permite al usuario ver solo los mensajes deseados. Por ejemplo, si la consola está configurada en nivel WARNING, el usuario no verá los mensajes DEBUG o INFO, pero verá los mensajes de registro WARNING, ERROR y CRITICAL. Por ejemplo, suponga que se ejecutó el siguiente código:

```
import logging
logging.basicConfig(level=logging.WARNING)
# default logging level is WARNING

logging.info("Hello world.")
logging.warning("Title: %d Dalmatians" % 101)
logging.debug("Title: %s" % "101 Dalmatians")
```

Figura 19: Ejemplo de Registro

Debido a que el nivel de registro predeterminado es WARNING, se imprimirá lo siguiente en la

consola: WARNING:root: Título: 101 dálmatas

Cuando se ejecuta un caso de prueba, es importante que algo realmente se pruebe. Cada caso de prueba necesita tener resultados y comportamientos esperados que podamos verificar. Python tiene un dispositivo integrado para verificar si se produjeron los datos o el comportamiento correctos: la aserción.

Una aserción es una afirmación de que se espera que sea verdadero en un punto particular en el guión. Por ejemplo, si prueba una calculadora, podríamos agregar dos más dos y luego afirmar que la respuesta debe ser igual a cuatro. Si por algún motivo el cálculo es incorrecto, el guión mostrará una excepción.

La sintaxis es la siguiente:

assert sumVariable==4, "sumVariable should equal 4."

Con Selenium WebDriver, un paso en un caso de prueba a menudo consiste en las siguientes acciones.

1. Localizar un elemento Web en una pantalla.
2. Actuar sobre el elemento Web.
3. Asegurarse de que sucedió lo correcto.

Para la acción (1), si el elemento no se encuentra o está en estado incorrecto, se mostrará una excepción.

Para la acción (2), si el intento de actuar sobre el elemento Web falla, se indica una excepción.

Para la acción (3), podemos emplear una aserción para verificar que el comportamiento esperado ocurrió o que se recibió el valor.

Esto, en esencia, sigue la forma en que un probador manual realizaría este paso y le permite a la persona que evalúa una prueba fallida comprender lo que ocurrió.

La gestión de información a menudo está asociada con el registro, pero es diferente. El registro proporciona información sobre la ejecución de la automatización al analista de pruebas que ejecutó el paquete, y al (a los) automatizador(es) responsable(s) de reparar las pruebas cuando sea necesario. La gestión de información está suministrando esa información, y probablemente otra información contextual a las distintas partes interesadas, hacia afuera y hacia arriba, que lo quieran o necesiten.

Es importante que los automatizadores determinen quién quiere o necesita gestión de información y qué información les interesa. Simplemente enviar los registros sin editar a las partes interesadas probablemente abrumaría a muchos de ellos sepultándolos en información que no necesitan ni desean.

Una forma de lidiar con la gestión de información es crearla a partir de los registros y otra información y ponerla a disposición de las partes interesadas, de modo que puedan descargarla cuando lo deseen. La otra forma es crear y enviar los informes tan pronto como estén listos para las partes interesadas que los quieran.

De cualquier manera, es una buena idea automatizar la creación y diseminación de informes, si es posible eliminar otra tarea manual que debe realizarse.

Los informes deben contener un resumen con una descripción general del sistema que se está probando, los entornos en los que se realizaron las pruebas y los resultados que se produjeron durante la ejecución. Como se mencionó, cada parte interesada puede querer una visión diferente de los resultados y esas necesidades deben ser proporcionadas por el equipo de automatización. A menudo, las partes interesadas pueden querer ver las tendencias de las pruebas, no solo un punto

en el tiempo. Como es probable que cada proyecto tenga diferentes partes interesadas con diferentes necesidades, cuanto más automatizadas sean las tareas de gestión de información, más fácil será.

3.2 Navegar a diferentes URL

3.2.1 Iniciar una sesión de automatización de prueba

Hay muchos navegadores diferentes que posiblemente que desee probar. Si bien la tarea básica de un navegador es permitirle ver e interactuar con varias páginas Web, es probable que cada navegador se comporte de manera diferente a otros navegadores.

Al principio de Internet, había algunas páginas que solo funcionaban en Netscape, otras que solo funcionaban correctamente en IE. Afortunadamente, la mayoría de esos problemas desaparecieron hace tiempo, aunque todavía puede haber algunas incompatibilidades entre los navegadores. Cuando una organización desea entregar un sitio Web, las pruebas deben realizarse en diferentes navegadores para garantizar la compatibilidad.

En el capítulo 1.4, cuando presentamos por primera vez Selenium WebDriver, mencionamos que diferentes navegadores requieren diferentes controladores para garantizar que los guiones automatizados que escribimos funcionen con los diferentes navegadores. La siguiente es la lista que primero mostramos allá:

- Chrome (chromedriver.exe)
- Internet Explorer (IEDriverServer.exe)
- Edge (MicrosoftWebDriver.msi)
- Firefox (geckodriver.exe)
- Safari (safaridriver)
- HtmlUnit (HtmlUnit driver)

Por ejemplo, si quisiéramos probar el navegador Chrome, primero tendríamos que descargar el controlador de Chrome (convenientemente llamado ***chromedriver.exe***) e instalar ese archivo en un lugar conocido de la estación de trabajo de prueba. Esto a menudo se hace editando la variable de entorno ***Path*** para que su sistema operativo la encuentre cuando sea necesario.

Por favor tenga en cuenta que, como gran parte de la tecnología, estos archivos de controladores y la información necesaria para usarlos cambian rápidamente. La información en este programa de estudios ha sido verificada y es válida en este momento de su redacción. Todo es posible, no hay nada imposible para mañana. Lo mejor que puede hacer un automatizador es familiarizarse con la documentación de ayuda en los diversos sitios Web de ayuda para varios navegadores y el sitio de ayuda de Selenium WebDriver.

Para trabajar con páginas Web, primero necesita abrir un navegador Web. Esto se puede lograr creando un ***objeto WebDriver***. La instancia del ***objeto WebDriver*** creará la interfaz de programación entre el guión y el navegador Web. También ejecutará un proceso WebDriver si se necesita uno para un navegador Web en particular. Por lo general, también ejecutará el navegador Web.

```
from selenium import webdriver
driver = webdriver.Chrome()
```

Figura 20: Código para crear el objeto WebDriver

Este código solo funcionará después de instalar ***chromedriver.exe*** como se describe arriba.

Al crear un objeto WebDriver se iniciará un navegador Web con una página vacía. Para navegar a la página del sitio deseado, se debe usar la función ***get()***. Tenga en cuenta que se accede a la funcionalidad del objeto del controlador mediante la notación de puntos, ya que Python es un lenguaje orientado a objetos.

```
driver.get('https://www.python.org')
```

Figura 21: Conducción hacia el URL

También es posible adjuntar un objeto WebDriver a un proceso WebDriver existente o crear un proceso WebDriver y hacer que se adjunte a un navegador ya en ejecución a través de Selenium RemoteWebDriver, pero estas operaciones están más allá del alcance de este programa de estudios.

Después de abrir una página o navegar a otra, es aconsejable verificar si se ha abierto la página correcta. El objeto WebDriver contiene dos atributos útiles para hacerlo: ***current_url*** y ***title***. Verificar los valores de estos campos permite que el guión realice un seguimiento de su página actual.

```
assert driver.current_url == 'https://www.python.org/', ErrMsg
assert driver.title == 'Welcome to Python.org', ErrMsg
```

Figura 22: Afirmar la Página Correcta

3.2.2 Navegación y actualización de páginas

Cuando necesite simular la navegación hacia adelante y hacia atrás en el navegador Web, debe usar los métodos ***back()*** y ***forward()*** del objeto WebDriver. Enviarán los comandos apropiados al WebDriver. Estos métodos no tienen argumentos.

```
driver.back()
driver.forward()
```

Figura 23: Navegación por el Historial del Navegador

También es posible hacer que el navegador Web actualice la página actual. Esto se puede hacer llamando al método
refresh() de WebDriver.

```
driver.refresh()
```

Figura 24: Actualizar el Navegador

3.2.3 Cerrar el Navegador

Al final de la prueba, debe cerrar el proceso del navegador Web y cualquier proceso de controlador adicional que se haya ejecutado. Si no cierra el navegador, permanecerá abierto incluso cuando la prueba haya finalizado. Es una buena idea tratar de establecer el entorno en el mismo estado en el que comenzó su(s) prueba(s). Eso permite que una cantidad indefinida de pruebas se ejecute desatendida.

Para cerrar el navegador controlado por WebDriver, llame al método ***quit()*** del objeto WebDriver.

```
driver.quit()
```

Figura 25: Cerrar todo el navegador

Debe colocar la función ***quit()*** en la parte del guión de prueba que se ejecuta independientemente del resultado de la prueba. Un error común es tratar de cerrar el navegador como uno de los pasos de prueba ordinarios. En tal caso, cuando la prueba falla, una librería del ejecutador de la prueba detiene la ejecución de los pasos de prueba y el paso que es responsable de cerrar el navegador nunca se ejecuta. Por lo general, las librerías de prueba tienen sus propios mecanismos para definir y ejecutar el código de desmontaje (p. ej., los métodos ***tearDown()*** del módulo de prueba ***unittest*** de Python). Vea la documentación de la librería que está utilizando para obtener más detalles sobre este tema.

Si tiene varias pestañas abiertas en el navegador a prueba, puede determinar cuál de las pestañas está abierta marcando el título de la ventana actual usando:

```
cur_win = driver.title
```

Figura 26: Obtener el Título de la Ventana Activa

Si no desea salir de todo el navegador y solo cierra la pestaña en el navegador, use la función anterior y avance por las pestañas hasta que llegue a la ventana que va a cerrar. Luego cierre la pestaña usando el método ***close()***. Cuando se cierra la última pestaña abierta, el proceso del navegador se cierra automáticamente.

```
driver.close()
```

Figura 27: Cerrar la Pestaña Activa

Este método no toma parámetro alguno y cierra la pestaña activa. Recuerde cambiar el contexto de la pestaña deseada para poder cerrar esa pestaña. Una vez que ocurre el cierre, llamar a cualquier otro comando de WebDriver que no sea ***driver.switch_to.window()*** generará una excepción ***NoSuchWindowException*** ya que la referencia del objeto aún apunta a la ventana que ya no existe. Debe cambiar de forma proactiva a otra ventana antes de llamar a cualquier método WebDriver. Después de cambiar las pestañas, puede verificar el título para determinar qué pestaña está activa ahora. Analizaremos el control de múltiples pestañas a continuación.

3.3 Cambiar el Contexto de la Ventana

Algunas veces, al probar aplicaciones o escenarios más complejos, deberá cambiar el contexto

actual de la GUI que está probando. Esto puede deberse a la necesidad de verificar el resultado de la prueba en un sistema diferente o ejecutar un paso de prueba en una aplicación diferente.

A veces, la GUI de la aplicación que está probando será tan compleja que necesitará cambiar entre marcos o ventanas.

El navegador Web no necesita tener enfoque para poder ejecutar los comandos de Selenium WebDriver, ya que el protocolo WebDriver se basa en la comunicación HTTP. Esto permite que WebDriver ejecute varias pruebas de forma simultánea o controle varios navegadores en un guión.

Cambiar el contexto del guión se puede hacer de dos maneras:

- cambiar los navegadores
- cambiar las ventanas/pestañas dentro de un navegador
- cambiar marcos dentro de una página

Para abrir dos navegadores, debe crear dos objetos WebDriver. Cada objeto WebDriver controla un navegador. Cada objeto webdriver se usa en el guión de prueba de acuerdo con los pasos del caso de prueba automatizado. Los objetos WebDriver pueden controlar el mismo tipo de navegadores web (p. ej., ambos Chrome, ambos Firefox) o diferentes tipos (p. ej., uno Chrome y el otro Firefox). También puede abrir más de dos navegadores si lo necesita. Recuerde que cada navegador está controlado por un objeto WebDriver. Y recuerde cerrar todos los navegadores al finalizar la prueba.

Abrir varias pestañas en un solo navegador puede ser complicado, ya que los diferentes navegadores y sistemas operativos tienen diferentes métodos para hacerlo. Una forma de que funcione para Windows en el navegador Chrome es ejecutar una llamada de función en código JavaScript de la siguiente manera:

```
driver.execute_script("$(window.open(''))")
```

Figura 28: Llamada de JavaScript para abrir una pestaña nueva

Más discusión sobre la apertura de varias pestañas en un navegador Web está fuera del alcance de este programa de estudios.

Para alternar entre pestañas abiertas en un navegador, primero debe obtener la lista de todas las pestañas abiertas (ventanas). Esta lista se puede encontrar en el atributo ***window_handles*** del objeto WebDriver. Tenga en cuenta que el orden de los identificadores en la matriz ***window_handles*** puede ser diferente al orden de las pestañas en el navegador.

Puede recorrer todas las pestañas de la ventana con el siguiente código:

```
for handle in driver.window_handles:
    driver.switch_to.window(handle)
```

Figura 29: Ciclaje a través de las ventanas abiertas

La forma más segura de determinar qué ventana es la que está abierta actualmente es usar el atributo ***driver.title***
mencionado anteriormente.

El siguiente código Python abre la página principal de Python, abre una nueva pestaña, luego abre la página principal de Perl en la segunda pestaña y luego vuelve a cambiar el navegador Web a la pestaña que contiene la página principal de Python:

```
From selenium import webdriver
driver = webdriver.Chrome()
driver.get('https://python.org')
driver.execute_script("$(window.open(''))")
driver.switch_to.window(driver.window_handles[1])
driver.get('https://perl.org')
driver.switch_to.window(driver.window_handles[0])
```

Figura 30: Abrir Dos Pestañas y Cambiar Entre Ellas

Tenga en cuenta que este código no es seguro para una prueba de producción real, ya que supone que los controladores de la pestaña estarán en orden. Es solo para fines de referencia. Para ver realmente este código ejecutarse, agregue algunas funciones ***sleep()*** para que no cambie demasiado rápido.

La tercera situación en la que puede querer cambiar el contexto en un navegador Web es cambiar los marcos. Esto es frecuentemente requerido; si no cambia el contexto a un marco en particular, no podrá encontrar elementos en ese marco y, en consecuencia, no podrá automatizar las pruebas de los elementos en ese marco.

Para cambiar el contexto a un marco específico, primero necesita encontrar ese marco. Si conoce la ***ID*** del marco, puede cambiar directamente a ella de la siguiente manera:

```
driver.switch_to.frame('foo')
```

Figura 31: Cambio de Marcos

donde ***foo*** es la ***ID*** del marco al que desea cambiar el contexto.

Si el marco no tiene ***ID*** o si desea usar una estrategia diferente para encontrarla, el cambio de contexto se puede hacer en dos pasos. El primer paso, como se mencionó anteriormente, es encontrar el marco como un elemento Web, y el segundo paso es cambiar al marco encontrado. El cambio en este caso se ejecuta por el mismo método que cambiar por ***ID***, pero el argumento dado a una función es el elemento Web encontrado.

A continuación hay una sección del código de Python que muestra un ejemplo de cambio de marco donde encontrar el marco es el primer paso:

```
frm_message = driver.find_element_by_name('message')
driver.switch_to_frame(frm_message)
```

Figura 32: Encontrar y luego cambiar de marco

Tenga en cuenta que durante la llamada a ***switch_to.frame()***, no usamos apóstrofes ni comillas, ya que pasamos la variable ***frm_message*** como un argumento de variable en lugar de una cadena que contiene la ***ID*** del marco.

Si desea volver al marco principal, use:

```
driver.switch_to.parent_frame()
```

Figura 33: Cambiar al Padre de un Marco

También puede volver a la página completa usando:

```
driver.switch_to.default_content()
```

Figura 34: Cambiar a la ventana principal

Además de cambiar un contexto de ventana o un contexto de marco del navegador Web, el marco Selenium WebDriver le permite manipular el tamaño de la ventana de un navegador Web. El término ventana aquí se usa como la noción de sistema operativo de toda la ventana, en lugar de solo una pestaña dentro de un navegador Web.

Selenium permite minimizar y maximizar el navegador Web y ponerlo en modo de pantalla completa. Los enlaces de Python para esta funcionalidad se muestran a continuación:

```
maximize: driver.maximize_window()
minimize:  driver.minimize_window()
fullscreen: driver.fulllscreen_window()
```

Figura 35: Dimensionar el Navegador

Estas funciones no toman argumento alguno, ya que operan en el navegador Web controlado por el objeto del ***controlador***.

3.4 Recopilar Capturas de Pantalla de Páginas Web

Cuando un probador manual está ejecutando un caso de prueba, interactúan visualmente con los objetos de la GUI en la pantalla. Si un caso de prueba falla, pueden saberlo porque lo que ven en la pantalla ya no es correcto.

Un probador que usa la automatización no se permite ese lujo.

Los guiones de automatización de pruebas no pueden verificar de forma fiable el diseño y la apariencia de las páginas Web. Hay muchas ocasiones en las que es útil tomar capturas de pantalla o un elemento particular de la pantalla y guardarlos en el registro o en una ubicación conocida, para que puedan verse en otro momento.

- Cuando la prueba automatizada detecta que ocurrió un fallo.
- Cuando la prueba es muy visual, y la determinación de paso/fallo puede realizarse solo observando la imagen de la pantalla.
- Cuando se trata de software de seguridad o de misión crítica que podría requerir una auditoría de la prueba.
- Al hacer pruebas de configuración en diferentes sistemas.

Para aprovechar la información que proporcionan las capturas de pantalla, deben tomarse en el momento adecuado. A menudo, las partes de los guiones de prueba que toman capturas de pantalla se colocan justo después de los pasos de prueba que controlan la interfaz de usuario o en las funciones de anulación. Sin embargo, dado que pueden constituir una herramienta invaluable para comprender las pruebas automatizadas, se pueden llevar a cualquier parte.

Una cuestión importante a tratar es nombrar los archivos con nombres y ubicaciones únicos para que su automatización no sobrescriba las capturas de pantalla tomadas tomadas antes en la ejecución. Hay una variedad de formas diferentes de hacer esto; sin embargo, están más allá del alcance de este programa de estudios que se va a elaborar.

Las capturas de pantalla se pueden tomar en dos ámbitos diferentes: la página completa del navegador o un solo elemento en la página del navegador[1]. Ambos usan la misma llamada a un método pero se llaman desde diferentes contextos.

El método de llamada a utilizar es ***screenshot()***. El siguiente ejemplo de Python muestra cómo tomar una captura de pantalla de una página completa y colocarla en una ubicación específica:

```
driver.get_screenshot_as_file('C:\\temp\\screenshot.png')
```

Figura 36: Guardar una captura de pantalla de toda la página

Este ejemplo muestra cómo tomar una captura de pantalla de un elemento y guardarlo en una ubicación específica:

```
ele = driver.find_element_by_id('btnLogin')
ele.screenshot('c:\\temp\\element_screenshot.png')
```

Figura 37: Guardar una captura de pantalla del elemento

Tomar una captura de pantalla en el navegador puede tomar algo de tiempo ya que la estación de trabajo debe realizar mucho procesamiento. Si el estado de la GUI cambia rápidamente (p. ej., al ejecutar simultáneamente librerías AJAX), es posible que la captura de pantalla no muestre el estado exacto de la página o del elemento esperado.
Una vez más, hay remedios para esta situación, pero están más allá del alcance de este programa de estudios.

Si desea tomar una captura de pantalla y tratarla como algo más que un archivo, existen alternativas en WebDriver. Por ejemplo, supongamos que en lugar de usar un archivo HTML para un registro,

desea iniciar sesión en una base de datos. Cuando toma una instantánea de la pantalla, o de un elemento, no le gustaría crear un archivo ***.png*** con él, es posible que desee transmitirlo directamente a un registro de la base de datos. Las siguientes llamadas crearían una imagen como una versión de cadena codificada en base64 de la instantánea. La versión codificada en base64 es mucho más segura de transmitir que un archivo binario, como un archivo ***.png***. La primera llamada capturará toda la ventana, el segundo un único elemento:

---

1 La documentación de la librería de WebDriver Python versión 3.13.0 afirma que la funcionalidad para tomar una captura de pantalla de un WebElement está disponible llamando a la función: WebElement.screenshot('Filename.png'). Los autores de este programa de estudios no han podido verificar que esta funcionalidad funcione en el navegador Chrome 67 con chromedriver 2.36, aunque está definida en la recomendación técnica de WebDriver W3C. Si necesita esta funcionalidad, existen soluciones alternativas documentadas en varios sitios Web. Pruebe una búsqueda en Google de "Captura de pantalla de WebElement de Python WebDriver". Esta función parece funcionar cuando se prueba el navegador Firefox.

```
img_b64 = driver.get_screenshot_as_base64()
img_b64 = element.screenshot_as_base64
```

Figura 38: Creación de una cadena de Base64 a partir de una imagen

Asimismo, si desea obtener una cadena binaria que represente una imagen, sin guardarla en un archivo, puede usar las siguientes llamadas para devolver la representación binaria de un archivo ***.png***. La primera llamada devolverá la imagen de una pantalla completa, la segunda devolverá una imagen de un elemento individual.

```
png_str = driver.get_screenshot_as_png()
png_str = element.screenshot_as_png()
```

Figura 39: Creación de una cadena binaria a partir de una imagen

3.5 Localizar Elementos de la GUI

3.5.1 Introducción

Para realizar la mayoría de las operaciones con WebDriver, deberá ubicar los elementos de la IU en los que desea operar en la pantalla actualmente activa. Esto se puede lograr utilizando los métodos ***find_element_*** o ***find_elements_*** dentro de WebDriver. Ambos métodos tienen versiones diferentes según lo que desee buscar. Por ejemplo, los siguientes están disponibles para la búsqueda:

- ***by_id (id_)***
- ***by_class_name (name)***
- ***by_tag_name (name)***
- ***by_xpath (xpath)***
- ***by_css_selector (css_selector)***

En cada caso, el argumento tomado por la función es una cadena que representa el(los) elemento(s) que estamos tratando de encontrar. Los valores de retorno son diferentes; p. ej., la versión singular (***find_element_***) devolverá un solo WebElement (si se encuentra uno) mientras que la versión múltiple (***find_elements_***) devolverá una lista de WebElements que coincidan con el argumento.

Para tener este debate, necesitamos introducir un concepto utilizado en el desarrollo Web y en las pruebas: el DOM (Document Object Model/*Modelo de Objetos del Documento*). Cuando se carga una página Web en el navegador, el navegador crea un DOM, modelando la página Web como un árbol de objetos. Este DOM define un estándar para acceder a la página Web. Según lo definido por el W3C:

> *"El Modelo de Objetos del Documento (DOM) del W3C es una plataforma y una interfaz de lenguaje neutral que permite que los programas y guiones accedan y actualicen dinámicamente el contenido, la estructura y el estilo de un documento".*

El DOM define:

- Todos los elementos HTML como objetos
- Las propiedades de todos los elementos HTML
- Los métodos que se pueden utilizar para acceder a todos los elementos HTML
- Los eventos que pueden afectar a todos los elementos HTML

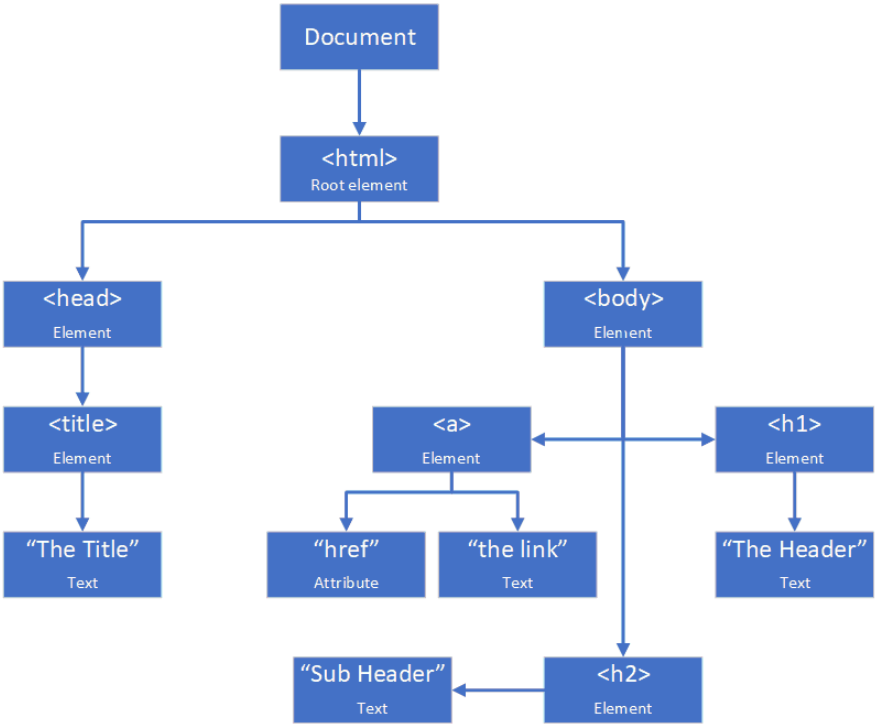

Figura 40: Árbol del Modelo de Objetos del Documento

3.5.2 Métodos HTML

Los siguientes métodos dependen de la búsqueda de elementos de pantalla buscando en el documento artefactos HTML. El primer método que discutiremos es usar el atributo de ***ID*** HTML para cada forma diferente de ubicar un WebElement, es posible que haya ventajas y desventajas.

```
<element id="unique_id">
```

Figura 41: Fragmento de HTML

Un ejemplo de localización de Python WebDriver:

```
element_found = driver.find_element_by_id('unique_id')
```

Figura 42: Localizar por ID

Ventajas:

- Eficiente
- Por definición, cada ID debe ser único en el documento HTML
- Un probador puede agregar IDS al SUT fácilmente (nota: estos cambios deben ser revisados por pares)
- Los ID se pueden generar automáticamente, lo que significa que pueden cambiar dinámicamente
- Los ID no son apropiados para el código que se usa en varios lugares, p. ej., una plantilla utilizada para generar encabezados y pies de página para diferentes diálogos modales
- A un probador no se le puede permitir modificar el SUT

La segunda forma de encontrar un WebElement es encontrarlo buscando su nombre de clase. Este se refiere al atributo de *clase* HTML en un elemento DOM, por ejemplo, en este fragmento de HTML:

```
<element class="class-name1">
```

Figura 43: Fragmento de HTML

Ejemplo de localización de Python WebDriver:

```
element_found = driver.find_element_by_class_name('class-name1')
```

Figura 44: Localizar por nombre de clase

Ventajas:

- Los nombres de clase se pueden usar en varios lugares en el DOM, pero puede limitar la ubicación a la página cargada (p. ej., en un cuadro de diálogo modal emergente)
- Un probador puede agregar nombres de clase al SUT fácilmente (nota: estos cambios deben ser revisados por pares)

Desventajas:

- Dado que los nombres de clase se pueden usar en varios lugares, se debe tener más cuidado de no ubicar el elemento incorrecto
- A un probador no se le puede permitir modificar el SUT

La tercera forma de analizar la búsqueda de un WebElement es usar el nombre de la etiqueta HTML del elemento. Esto se refiere al nombre de la etiqueta DOM de un elemento, por ejemplo, en este fragmento de HTML:

```
<h2>
```

Figura 45: Etiqueta HTML

Ejemplo de localización de Python WebDriver:

```
heading2_found = driver.find_element_by_tag_name('h2')
```

Figura 46: Localizar por medio de la etiqueta

Ventaja: Si un elemento es exclusivo de una página, puede limitar su búsqueda.

Desventajas: Si un elemento *no* es exclusivo de una página, puede localizar el elemento incorrecto.

Nuestra cuarta forma de identificar WebElement por el texto del enlace. Esto se refiere solo a una etiqueta de anclaje que contiene texto que se resaltará para que un usuario haga clic. En realidad, hay dos métodos similares: puede especificar la cadena de texto completa o una parte de la cadena. Por ejemplo, en el siguiente fragmento de HTML, el texto del enlace se ha escrito en negrita para identificarlo.

```
<html>
 <body>
  <p class="paragraph">XyzAbc.</p>
  <a href="next.html">Next Page</a>
 </body>
</html>
```

Figura 47: Fragmento de HTML

Ejemplos de localización de Python WebDriver:

```
element = driver.find_element_by_link_text('Next Page')
```

Figura 48: Localizar por medio del texto del enlace

```
element = driver.find_element_by_partial_link_text('Next Pa')
```

Figura 49: Localizar por medio del texto parcial del enlace

Ventajas:

- Si el texto del enlace es exclusivo de una página, puede encontrar el elemento.
- El texto del enlace es visible para el usuario (en la mayoría de los casos), por lo que es fácil saber qué es lo que busca el código de prueba.
- El texto parcial del enlace es ligeramente menos probable que el

texto completo del enlace a cambiar.

Desventajas:

- Más probabilidades de cambiar el texto del enlace que un ID o nombre de clase
- El uso de texto parcial puede dificultar la identificación exclusiva de un solo enlace

3.5.3 Métodos XPath

Como se discutió en la sección 2.2, XPath (XML Path Language) es un lenguaje que se puede usar para seleccionar nodos específicos usando una variedad de criterios en un documento XML. Por ejemplo, considere el siguiente fragmento HTML. Como HTML es un subconjunto de XML, se puede buscar utilizando XPath. Puede haber problemas si el HTML está mal formado (p. ej., no todas las etiquetas cercanas están incluidas).

```
<html>
 <body>
  <form id="sample_form1">
   <input name="text1" type="text" />
   <input name="text2" type="text" />
   <input name="submit_button" type="submit" value="Enter" />
  </form>
 </body>
</html>
```

Figura 50: Fragmento de HTML

WebDriver puede usar XPath para encontrar un nodo específico, y a partir de él, se puede localizar un elemento Podría especificar una ruta absoluta, pero esa es una mala idea porque cualquier cambio probablemente invalidaría su código. Una mejor manera es identificar una ruta relativa comenzando desde un nodo encontrado que coincida con un criterio. A continuación se muestra un ejemplo de la especificación de una ruta absoluta, y luego una versión más sólida para encontrar el elemento deseado a través de XPath. Ambos devolverán el segundo campo de entrada en el fragmento de HTML.

```
element = driver.find_element_by_xpath('/html/body/form[2]')
```

Figura 51: Uso de XPath con Ruta Absoluta

```
element = driver.find_element_by_xpath("//form[@id='sample_form1']/input[2]")
```

Figura 52: Uso de XPath con Ruta Relativa

El uso de la ruta relativa a menudo requerirá más pulsaciones de tecla para definir, pero no podrá encontrar el nodo deseado con mucha menos frecuencia. Ese es un buen equilibrio.

Dentro de una cadena XPath, puede buscar el ***ID***, ***nombre***, ***clase***, *nombre de etiqueta*, etc. Por lo tanto, es posible crear funciones de localización genéricas usando XPath pasando un tipo de atributo a la función, o una cadena de ruta que incorpore el atributo en ella. Por ejemplo:

```
def find_by_xpath(driver, path_string):
    element = driver.find_element_by_xpath('path_string')
    return element
```

Figura 53: Función de Búsqueda Genérica con XPath

La variable ***path_string*** se puede ensamblar según el atributo que estamos tratando de encontrar. A continuación se encuentran dos ejemplos, la primera búsqueda por ***ID***, la segunda búsqueda por ***clase***:

```
path_string = "//*[@id = '%s']" % 'id_to_find'
path_string = "//*[@class = '%s']" % 'class_to_find'
```

Figura 54: Construcción de una Cadena XPath

Llamar a la función genérica se realizaría de la siguiente manera:

```
element_found = find_by_xpath(driver, path_string)
```

Figura 55: Llamar a la Función Genérica de XPath

Esto es poco elegante, pero demuestra la flexibilidad de XPath para localizar

elementos. Ventajas:

- Puede encontrar elementos sin atributos únicos (ID, clase, nombre, etc.)
- Puede utilizar XPath en localizadores genéricos, usando las diferentes estrategias "By" (por ID, por clase, etc.) según sea necesario

Desventajas:

- El código XPath absoluto es "frágil" y puede romperse con los cambios más pequeños en la estructura HTML
- El código XPath relativo puede encontrar el nodo incorrecto si el atributo o elemento que busca no es único en la página
- XPath puede implementarse de manera diferente entre los diferentes navegadores, se puede necesitar un esfuerzo adicional para ejecutar pruebas de WebDriver en esos entornos

3.5.4 Métodos de Selectores de CSS

Como discutimos en la sección 2.3, también podemos encontrar elementos usando selectores de CSS. Por ejemplo, en este fragmento de HTML:

```
<html>
 <body>
  <p class="paragraph">Some Latin nonsense.</p>
 </body>
</html>
```

Figura 56: Fragmento de HTML

Al usar las reglas de los selectores de CSS, que se encuentran en la sección 2.3, el siguiente código debe identificar el nodo esperado:

```
element = driver.find_element_by_css_selector('p.paragraph')
```

Figura 57: Identificación de WebElement mediante Selectores de CSS

Ventaja: Si un elemento es exclusivo de una página, puede limitar su búsqueda.

Desventajas: Si un elemento *no* es exclusivo de una página, puede localizar el elemento incorrecto.

3.5.5 Localizar a través de las condiciones esperadas

Selenium con enlaces de Python tiene un módulo ***expected_conditions*** (condiciones esperadas) que se puede importar desde ***selenium.webdriver.support*** con varias condiciones convenientes predefinidas. Puede crear clases personalizadas de expected_condition, pero las clases predefinidas deberían satisfacer la mayoría de sus necesidades. Estas clases ofrecen una mayor especificidad que los localizadores mencionados anteriormente. Es decir, no solo determinan si existe un elemento, todos verifican un estado específico para ese elemento. Por ejemplo, ***element_to_belected()*** no solo determina que el elemento existe, sino que también verifica si se trata de un estado seleccionado.

Si bien esta lista no está completa, algunos ejemplos son:

- alert_is_present
- element_selection_state_to_be(element, is_selected)
- element_to_be_clickable(locator)
- element_to_be_selected(element)
- frame_to_be_available_and_switch_to_it(locator)
- invisibility_of_element_located(locator)
- presence_of_element_located(locator)
- text_to_be_present_in_element(locator, text_)
- title_is(title)
- visibility_of_element_located(locator)

Discutiremos más sobre esto en la sección 4.2, ya que muchos de estos también se utilizan como mecanismos de espera.

3.6 Obtener el estado de los elementos de la GUI

Simplemente obtener la ubicación de un WebElement específico a menudo no es suficiente. Para la automatización de pruebas, puede haber varias razones por las cuales también necesitamos acceder a cierta información sobre un elemento. La información puede incluir su visibilidad actual, si está habilitada o no, o si está seleccionada o no. Las razones pueden incluir:

- Asegúrese de que el estado sea el esperado en un punto determinado de la prueba
- Asegúrese de que un control esté en un estado que pueda ser manipulado según sea necesario en el caso de prueba (es decir, habilitado)
- Asegúrese de que después de manipular el control, ahora está en el estado esperado
- Asegúrese de que los resultados esperados sean correctos después de ejecutar una prueba

Los diferentes WebElements tienen diferentes formas de acceder a la información de ellos. Por ejemplo, muchos WebElements tienen una propiedad de texto que se puede recuperar utilizando el siguiente código:

```
element_text = Element.text
```

Figura 58: Recuperación de la propiedad de texto

No todos los WebElements tienen una propiedad de texto. Por ejemplo, si examinamos un nodo de encabezado (quizás usando un método ***find_by_css_selector***('h1')), esperaríamos que tubiera una propiedad de texto. Otros WebElements pueden no tener la misma propiedad. El contexto de WebElement y cómo se usa puede ayudarlo a comprender qué propiedades es probable que tenga.

Se debe acceder a algunas propiedades de WebElement utilizando un método WebElement. Por ejemplo, suponga que desea determinar si un WebElement específico está actualmente habilitado o deshabilitado. Puede llamar al siguiente método para obtener ese valor booleano:

```
cur_state = element.is_enabled()
```

Figura 59: Comprobar si WebElement está habilitado

La siguiente tabla no es detallada, pero enumera muchas de las propiedades comunes y los métodos de acceso que podría necesitar para la automatización.

Tabla 16: Propiedades y Métodos de Acceso Comunes

Propiedad/Método	Argumentos	Devuelve	Descripción
get_attribute()	propiedad a recuperar	propiedad, atributo o Ninguno	Obtiene propiedad. Si no existe propiedad con ese nombre, obtiene un atributo de ese nombre. Si no hay ninguno de los dos, devuelve Ninguno
get_property()	propiedad a recuperar	propiedad	Obtiene propiedad.
is_displayed()		Booleano	Devuelve verdadero si es visible para el usuario

is_enabled()	Booleano	devuelve verdadero si el elemento está habilitado
is_selected()	Booleano	Devuelve verdadero si se selecciona casilla de verificación o radio
location	Ubicación X, Y	Devuelve la ubicación X, Y en el lienzo renderizable
size	Altura, Ancho	Devuelve la altura y el ancho del elemento
tag_name	La propiedad tag_name	Devuelve el nombre de etiqueta del elemento
text	Texto para el elemento	Devuelve el texto asociado con el elemento

3.7 Interactuar con elementos de la GUI utilizando comandos de WebDriver

3.7.1 Introducción

Por lo tanto, ha localizado el WebElement con el que desea que interactúe su prueba automática (discutido en la sección 3.5). Se ha asegurado de que el elemento esté en el estado correcto, por lo que puede manipularlo según el caso de prueba (debatido en la sección 3.6). Ahora es el momento de hacer el cambio que realmente deseas.

Una de las principales razones por las que las interfaces gráficas de usuario se hicieron populares es que hay un conjunto limitado de controles que pueden manipularse; cada control es esencialmente una metáfora en pantalla que se puede manipular fácilmente con el teclado y/o el mouse. Cada control está diseñado para ser fácilmente comprendido, incluso por usuarios principiantes de computadoras. Entonces, puedo escribir en un campo de texto, hacer clic en un botón de radio, seleccionar un elemento de un cuadro de lista, etc.

Automatizar la manipulación de estos controles, sin embargo, puede ser más difícil de entender. Lo que hace un probador manualmente a menudo se hace inconscientemente. Como automatizadores, debemos asegurarnos de que entendemos todos los matices de hacer cambios a los controles en pantalla.

En las próximas secciones, discutiremos la manipulación de lo siguiente:

- Campos de texto
- WebElement cliqueable (campos en los que puede hacer clic)
- Casillas de verificación
- Listas desplegables

Para cualquier WebElement que planee manipular, es posible que le interesen varias cosas:

- El WebElement existe
- Se muestra WebElement
- El WebElement está habilitado

Dependiendo del sitio Web, la forma en que se escribió el HTML, si se está utilizando Ajax, y cualquier cantidad de otras condiciones, puede haber una discrepancia sobre si el WebElement con el que desea tratar realmente necesita mostrarse o no para que lo manipule. Por ejemplo, si WebElement está en la página activa y está habilitado, pero se ha desplazado de la vista, tratar de manipularlo puede no funcionar. Las pruebas ejecutadas en Chrome han demostrado que, en algunas páginas, al trabajar con WebElement no está visible en la actualidad funciona y en otros sitios indica una excepción. Nuestro mejor consejo es intentar que todos los WebElements con los que trabaja sean visibles (visualizados) en la pantalla.

Dado que una pantalla del navegador puede modificarse dinámicamente (a través de Ajax, por ejemplo) o actualizarse en base a acciones previas, estas verificaciones probablemente se realicen utilizando las clases ***expected_condition*** que nos permiten sincronizar el tiempo; discutiremos los del capítulo 4.

3.7.2 Manipulación de campos de texto

Al escribir en un campo de texto editable, generalmente deseará primero borrar el elemento, luego escriba la cadena deseada en el control. Supondremos que ya ha verificado que el elemento existe, se muestra y está habilitado. Si no asegura el estado del control, y uno o más de ellos es falso, su intento de manipulación del elemento causará una excepción.

Supondremos que usted ya ha localizado el control de entrada, y está en la variable nombrada *element*.

```
# First clear the control
element.clear()

# Now type into the control
string_to_type = 'XYZ'
element.send_keys(string_to_type)
```

Figura 60: Escribir en un Campo de Edición

3.7.3 Hacer clic en WebElements

Al hacer clic en un elemento se simula un clic del mouse. Esto se puede hacer con un botón de radio, enlace o imagen; esencialmente cualquier lugar en el que pueda hacer clic con el mouse manualmente. Aquí no incluimos las casillas de verificación; las analizaremos en la próxima sección. Una vez más, podría ser importante verificar para asegurarse de que WebElement es cliqueable actualmente, puede usar el método, ***element_to_be_clickable***, de la clase ***expected_condition*** para esperar a que se pueda hacer clic en él.

También, suponemos que WebElement ha sido localizado. Es posible que tengamos que esperar para asegurarnos de que esté listo para hacer clic, utilizando el método de ***expected_condition***:

```
Driver.support.expected_conditions.element_to_be_clickable(locator)
```

Figura 61: Método de Sincronización

La sincronización se tratará en el capítulo 4. Suponiendo que WebElement ha sido referenciado por el elemento de la variable y se puede hacer clic en él, entonces llamamos a:

```
element.click()
```

Figura 62: Hacer clic en WebElement

Si WebElement fuera un enlace o un botón, esperaríamos que ocurriera un cambio de contexto que podría verificarse en la pantalla. Si se trata de una casilla de verificación, el resultado puede ser seleccionado o no seleccionado: esto se analizará en la siguiente sección. Sin embargo, si se hizo clic en este botón de radio, puede verificar que el botón se haya seleccionado llamando a:

```
element.is_selected()
```

Figura 63: Comprobación del estado de WebElement

3.7.4 Manipulación de casillas de verificación

Al hacer clic en las casillas de verificación, deben tratarse de manera diferente a otros controles cliqueables. Si hace clic en un botón de opción varias veces, permanece *seleccionado*. No obstante, cada vez que hace clic en una casilla de verificación, alterna el estado seleccionado. Haga clic una vez, ahora *seleccionado*. Haga clic nuevamente, ahora *no seleccionado*. Haga clic de nuevo y vuelva a *seleccionado*.

Por lo tanto, es importante comprender el estado que queremos lograr al manipular una casilla de verificación. Podemos escribir una función que tomará la casilla de verificación y el estado deseado y realizará la acción correcta sin importar el estado actual en que se encuentre la casilla de verificación.

Suponga que la casilla de verificación se ha cargado en la casilla de verificación de la variable. Asuma una variable Booleana.
want_checked pasada para determinar el estado final que queremos.

```
# if we want it selected and it is not, then click on it
# if we want it deselected and it is selected, click on it
def set_check_box(element, want_checked):
    if want_checked and not element.is_selected():
            element.click()
    elif element.is_selected() and not want_checked:
            element.click()
```

Figura 64: Función para establecer la casilla de verificación

```
set_checkbox (checkbox, True)
```

Figura 65: Llamar a la función de casilla de verificación para marcar la casilla

3.7.5 Manipulación de controles desplegables

Los cuadros de lista desplegable (controles de selección) son utilizados por muchos sitios Web para permitirle a los usuarios seleccionar una de muchas opciones. Algunos de los cuadros desplegables permiten seleccionar múltiples elementos de la lista simultáneamente.

Hay muchas formas de trabajar con la lista de un control de selección. Estas incluyen opciones para seleccionar elementos individuales o múltiples. También hay diferentes formas de anular la selección de los elementos.

Las opciones de selección incluyen:

- Busque el HTML para encontrar el elemento deseado y luego haga click en él
- Seleccionar por un valor de elemento (***select_by_value***(*valor*))
- Seleccionar todos los elementos que muestran texto coincidente

(***select_by_visible_text***(*texto*))

- Seleccionar un elemento por índice

(***select_by_index***(índice)) Las opciones para anular

selección incluyen:

- Deseleccionar todos los elementos (***deselect_all***())
- Deseleccionar un elemento por índice (***deselect_by_index***(índice))
- Deseleccionar por valor (***deselect_by_value***(*valor*))
- Deseleccionar por texto visible (***deselect_by_visible_text***(*texto*))

Una vez que haya terminado de seleccionar/deseleccionar los elementos en la lista desplegable, hay varias opciones para que pueda ver lo que se ha seleccionado:

- Devuelve una lista de todos los elementos seleccionados (***all_selected_options***)
- Devuelve el primero seleccionado (o solo si es un solo control de selección)

(***first_selected_option***)

- Ver una lista de todas las opciones en la lista (***options***)

Puesto que hacer clic en los elementos de la lista es una tarea común que tenemos que hacer, podemos crear una función como la que hicimos con las casillas de verificación. En este caso, primero debemos hacer clic en el botón desplegable para mostrar la lista completa. Luego, una vez que se abra la lista, encuentre la opción deseada y haga clic en ella. Aquí mostramos una función que haría eso.

```
def click_dropdown_option_by_id_and_id(driver, dropdown_id, option_id):
    dropdown_element = driver.find_element_by_id('dropdown_id')
    dropdown_element.click()
    option_element = driver.find_element_by_id('option_id')
    option_element.click()
```

Figura 66: Función para hacer clic en el elemento de la lista desplegable

3.7.6 Trabajo con Dialogos Modales

Un diálogo modal es un cuadro que aparece sobre una ventana del navegador y no permite el acceso a la ventana subyacente hasta que se haya manejado. Estos son similares a los cuadros de diálogo de solicitudes del usuario, pero lo suficientemente diferentes como para analizarlos por separado. Analizaremos los avisos de usuario, como las alertas, en la siguiente sección.

En general, los diálogos modales se invocan cuando el autor del sitio Web desea obtener información específica del usuario (p. ej., un nombre de usuario/contraseña emergente) o tareas específicas realizadas por el usuario. Por ejemplo, en un sitio de comercio electrónico, agregar un artículo a un carro puede hacer que aparezca una ventana informativa como se muestra a continuación:

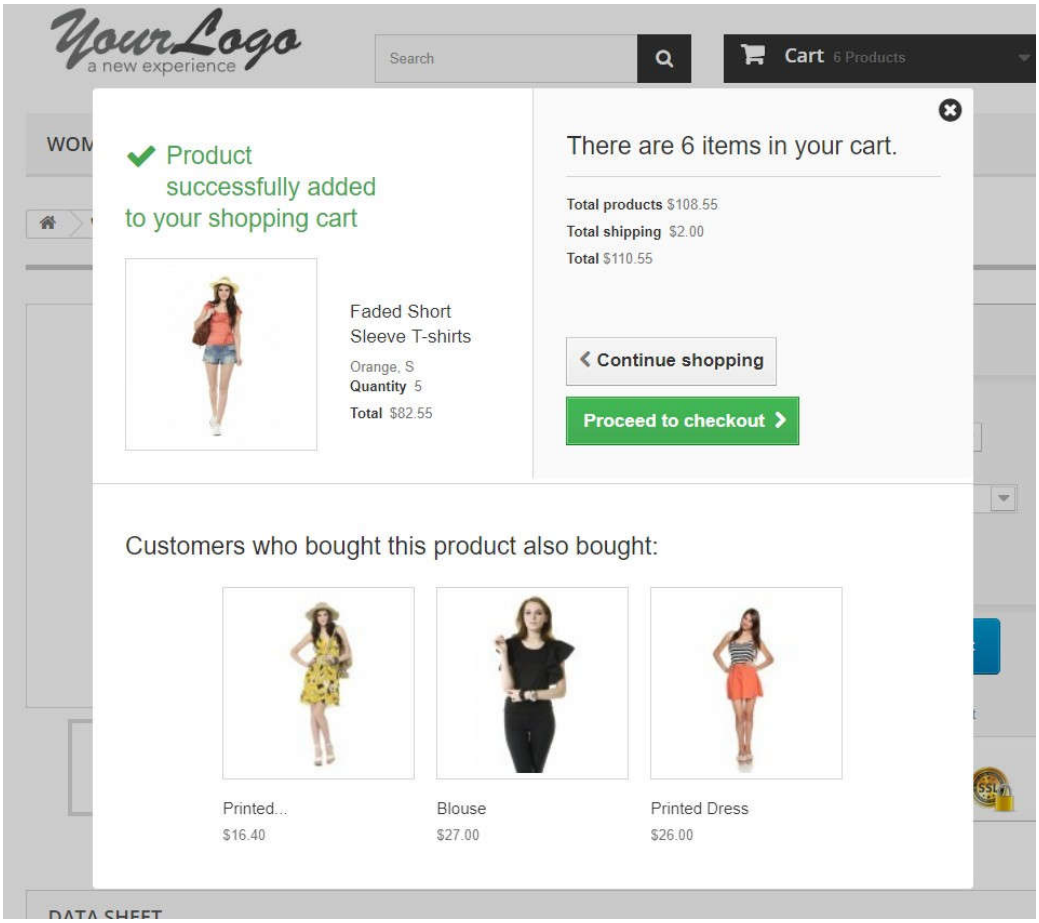

Figura 67: Ventana modal del carrito de comercio electrónico

En el ejemplo anterior, el usuario puede decidir qué camino tomar: proceder al pago o continuar la compra. Además, cualquier cantidad de información adicional puede presentarse en el cuadro de diálogo modal. En este caso, el sitio de comercio electrónico está tratando de tentar al usuario a

comprar algo más en función de lo que colocó en el carro.

Todo el código del diálogo modal se encuentra en el código HTML que apareció en el diálogo modal. Por lo tanto, manipular el diálogo modal es tan simple como encontrar el código en la página de llamada y manipularlo. Esto sigue las mismas reglas que para ubicar y manipular los controles de formulario como analizamos anteriormente.

En este caso, desea hacer clic en el botón *Proceed to Checkout* (Proceder al pago) en este dialogo modal. El primer paso sería determinar la ubicación del código para el diálogo modal. En este caso, el ***ID*** de la sección que representa el elemento modal es ***layer_cart,*** creamos una referencia de objeto WebDriver para este elemento del código de la siguiente manera:

```
modal = driver.find_element_by_id('layer_cart')
```

Figura 68: Encontrar el elemento modal

La siguiente tarea sería identificar el elemento que representa el botón utilizando la funcionalidad *Inspect* del navegador. En este caso, el nombre de clase del botón es ***button_medium***. Una vez más, queremos crear una referencia a este elemento para poder manipularlo. Podemos usar el siguiente código:

```
proceed_button = modal.find_element_by_class_name('button-medium')
```

Figura 69: Encontrar un botón en el elemento modal

Una vez encontrado, presionar el botón es tan fácil como llamar al método ***click()*** para esa referencia:

```
proceed_button.click()
```

Figura 70: Presionar el botón

En este punto, el diálogo modal se cierra y nos movemos a una nueva ubicación en la ventana principal del navegador, en este caso: la página de resumen del carrito de compras como se muestra a continuación:

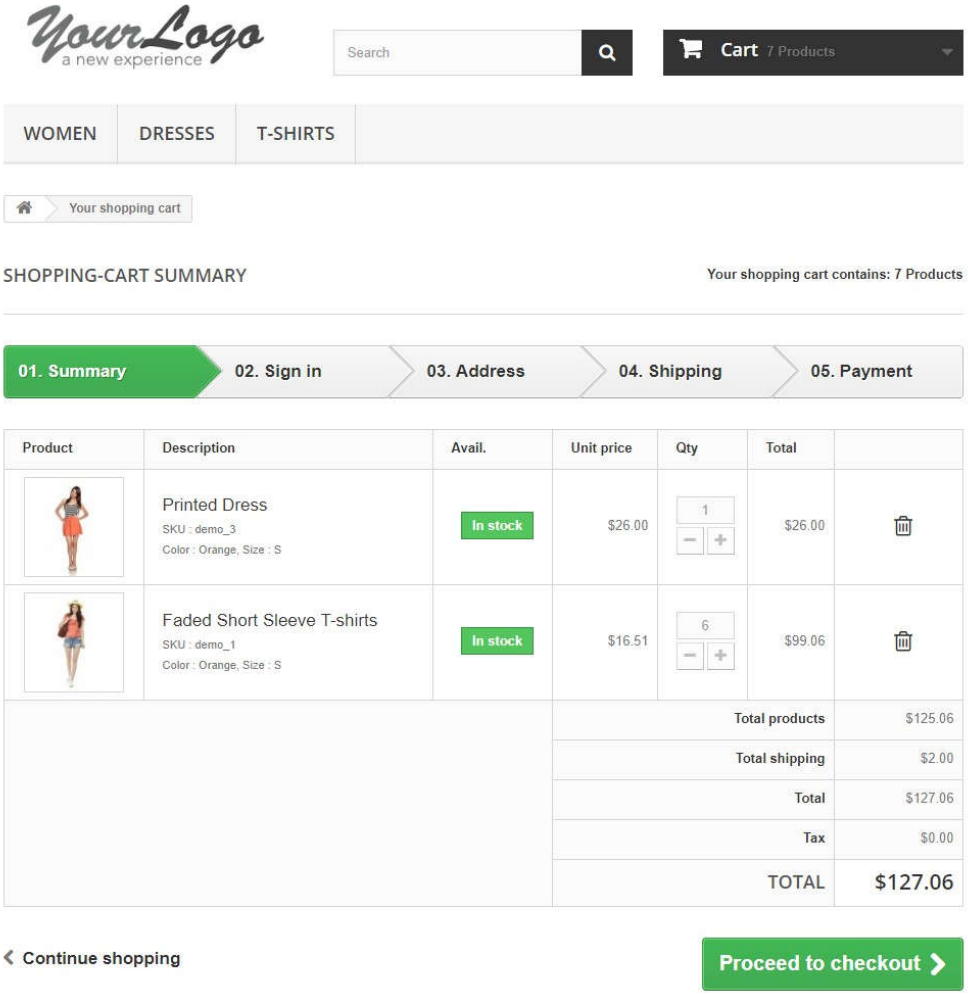

Figura 71: Pantalla del carrito de compras

3.8 Interactuar con los Mensajes de Usuario en los Navegadores Web utilizando los comandos de WebDriver

Los mensajes de los usuarios son ventanas modales que requieren que los usuarios interactúen con ellas antes de que el usuario pueda continuar interactuando con los controles en la propia ventana del navegador.

Para fines de automatización, los mensajes generalmente no se tratan automáticamente. Por lo tanto, si su secuencia de comandos intenta ignorar la solicitud y continúa enviando comandos a la ventana del navegador, la acción generará un error ***unexpected alert open*** (alerta de apertura inesperada).

Cada mensaje de usuario tiene un mensaje de usuario asociado (que puede ser NULO). Este es un campo de texto que puede recogerse usando el código que se muestra a continuación.

W3C define tres diálogos de tipo de alerta diferentes:

- Alert (Alerta)

- Confirm (Confirmar)
- Prompt (Mensaje)

Como todos están definidos de manera que funcionan casi igual, analizaremos el mensaje de alerta.

El cuadro de diálogo de alerta a menudo se usa para asegurarse de que el usuario conoce alguna información significativa.

Como los cuadros de diálogo de alerta no son en realidad parte de la página Web, requieren un tratamiento especial. El WebDriver que usa Python tiene un conjunto de métodos que le permiten controlar el diálogo de alerta desde su guión de automatización. Estos métodos son comunes a los tres mensajes de diálogo del usuario.

Esto primero crea una referencia usando un método del objeto WebDriver llamado ***switch_to***. La sintaxis para usar este método es la siguiente:

```
alert = driver.switch_to.alert
```

Figura 72: Crear un Objeto de Alerta

Podemos obtener el texto de la alerta por medio de:

```
msg_text = alert.text
```

Figura 73: Obtener el texto de la alerta

Para determinar si el texto esperado está en la alerta, cambie a la alerta, obtenga el texto y luego verifique si el texto esperado estaba en la alerta. Si el texto está allí, la variable ***passed*** (paso) se establecerá en ***True*** (verdadero). De lo contrario, la variable ***passed*** se establecerá en ***False*** (Falso).

```
alert = driver.switch_to.alert
msg_text = alert.text
expected_text = 'XYZ'
assert expected_text in msg_text, "The expected text not found"
```

Figura 74: Comparar el texto de la alerta

Los diálogos de alerta se pueden cerrar de dos maneras. Considere el método ***accept()*** (aceptar) como presionar el botón OK, y el método ***dismiss()*** (descartar) como presionar el botón Cancelar, o el botón de cerrar ventana en la esquina superior del aviso.

```
alert = driver.switch_to.alert
alert.accept()
alert.dismiss()
```

Figura 75: Cerrar una Alerta

Capítulo 4 - Preparación de Guiones de Prueba Mantenibles

Palabras Clave

prestación, Patrón de Objetos de Página, persona

Objetivos de Aprendizaje para la Preparación de Guiones de Prueba Mantenibles

STF-4.1 (K2) Comprender qué factores ayudan y afectan la mantenibilidad de los guiones de prueba

STF-4.2 (K3) Usar mecanismos de espera apropiados

STF-4.3 (K4) Analizar la GUI del SUT y utilizar Objetos de Página para hacer sus abstracciones

STF-4.4 (K4) Analizar los guiones de prueba y aplicar los principios de prueba guiada por palabras clave para construir guiones de prueba

4.1 Mantenibilidad de los Guiones de Prueba

En el capítulo 1, sección 2, analizamos la diferencia entre una prueba manual y un guión automatizado. Necesitamos revisar esae análisis porque tiene una influencia directa en la mantenibilidad del software que compilamos llamado automatización.

Reducido a su núcleo esencial, una prueba manual es un conjunto dirigido de instrucciones abstractas que solo son valiosas cuando las usa un probador manual para ejecutar realmente la prueba. Los datos y los resultados esperados también deberían estar allí, pero el quid está en los pasos a seguir. El probador manual agrega contexto y razonabilidad a esas sentencia abstractas, permitiendo que las pruebas de casi cualquier complejidad se ejecuten con éxito.

Si un paso en el guión manual dice hacer clic en un botón, el probador puede hacerlo sin pensarlo mucho. No se detienen a preguntarse si el control está visible, si está habilitado o si es el control correcto para la tarea. Sin embargo, lo cierto es que piensan en esas cosas subconscientemente. Si el control no está visible, pueden intentar hacer algo para hacerlo visible. Si no está habilitado, no intentarán usar el control a ciegas; intentarán descubrir por qué está deshabilitado y verán si pueden arreglarlo. Una vez que puedan llegar al control, pueden hacer lo correcto. Y, si algo incorrecto sucede con ese control, pueden identificar lo que ocurrió, por lo que pueden escribir el informe de defectos y/o modificar el procedimiento de prueba.

Toda herramienta de automatización es tonta. No importa cuán costosas sean, cada una tiene un contexto muy limitado y casi ninguna lógica incorporada en ella. Sin embargo, los automatizadores son seres humanos y son realmente inteligentes. Podemos agregar contexto y razonabilidad a una herramienta programándola en nuestros guiones automatizados.

Sin embargo, cuanto más intentemos programar esa inteligencia en el guión automatizado, más complejo será el guión, y más probable será que haya fallos del guión sencillamente debido a su complejidad innata.

La automatización de la escritura para las pruebas a menudo se ha comparado con el intento de parar una bala con otra bala.

Ese refrán siempre ha querido decir que hay mucha complejidad para la automatización. La complejidad, sin embargo, es una espada de doble filo. Necesitamos incorporar inteligencia en nuestros guiones de manera que simulen mejor a un probador humano que ejecuta una prueba. Necesitamos manejar una cantidad cada vez mayor de complejidad en los SUT en los que trabajamos.

Pero mientras más complejidad agreguemos a nuestra automatización, más fallos de la automatización podemos esperar.

No importa cuán inteligentes seamos como automatizadores, existen límites para las condiciones que podemos controlar a través del guión automatizado.

Un probador manual puede intentar descubrir por qué ese control no está visible. Un automatizador probablemente no pueda; estamos limitados a poner código en el guión para decirle a éste que espere un tiempo finito, con la esperanza de que el problema se resuelva solo. Lo mismo sucede con el control que está habilitado.

Lo que un automatizador puede hacer es intentar asegurarse de que el control está en el estado correcto antes de usarlo, y si no es así, esperar un tiempo determinado, y si aún no se puede utilizar, registrar el error y cerrar la ejecución del guión con elegancia para que podamos pasar a la siguiente prueba. Si podemos acceder al control, podemos verificar si el control se comporta como se espera después de manipularlo. Y, si hubo un problema, nuevamente podemos escribir un mensaje de registro útil, para que podamos solucionar el problema de manera más efectiva y eficiente.

Nos gustaría hacer nuestro trabajo más fácil. Eso significa poner inteligencia en funciones invocables en lugar de tener que programar la inteligencia en cada guión de forma individual. En otras palabras, mueva la inteligencia hasta el TAA y/o TAF y fuera del guión.

Al mover más inteligencia hacia arriba y fuera de los propios guiones, los guiones se vuelven más fáciles de mantener y más escalables. Si nuestro código que agrega inteligencia falla, hará que fallen muchos guiones, pero solucionarlos puede hacerse en un único punto de contacto.

Les proporcionamos algunos ejemplos de como compilar estas funciones invocables. El código que describimos, sin embargo, no era lo suficientemente bueno como para automatizar la producción. Los buenos automatizadores tienden a compilar estas funciones agregadas que algunos automatizadores llaman funciones de envoltura. A continuación aparece un ejemplo de una función de envoltura con inteligencia integrada presentada en forma resumida. Supongamos que quiero hacer clic en una casilla de verificación. Voy a tratar de compilar la función de modo que imite lo que

el probador manual realmente piensa y hace.

Los argumentos para esta función de envoltura susceptibles de incluir son: la casilla de verificación a utilizar, el estado final deseado (marcado o no), tal vez la cantidad de tiempo que estamos dispuestos a esperar si no está lista de inmediato. Entonces las tareas seguirían esta lógica:

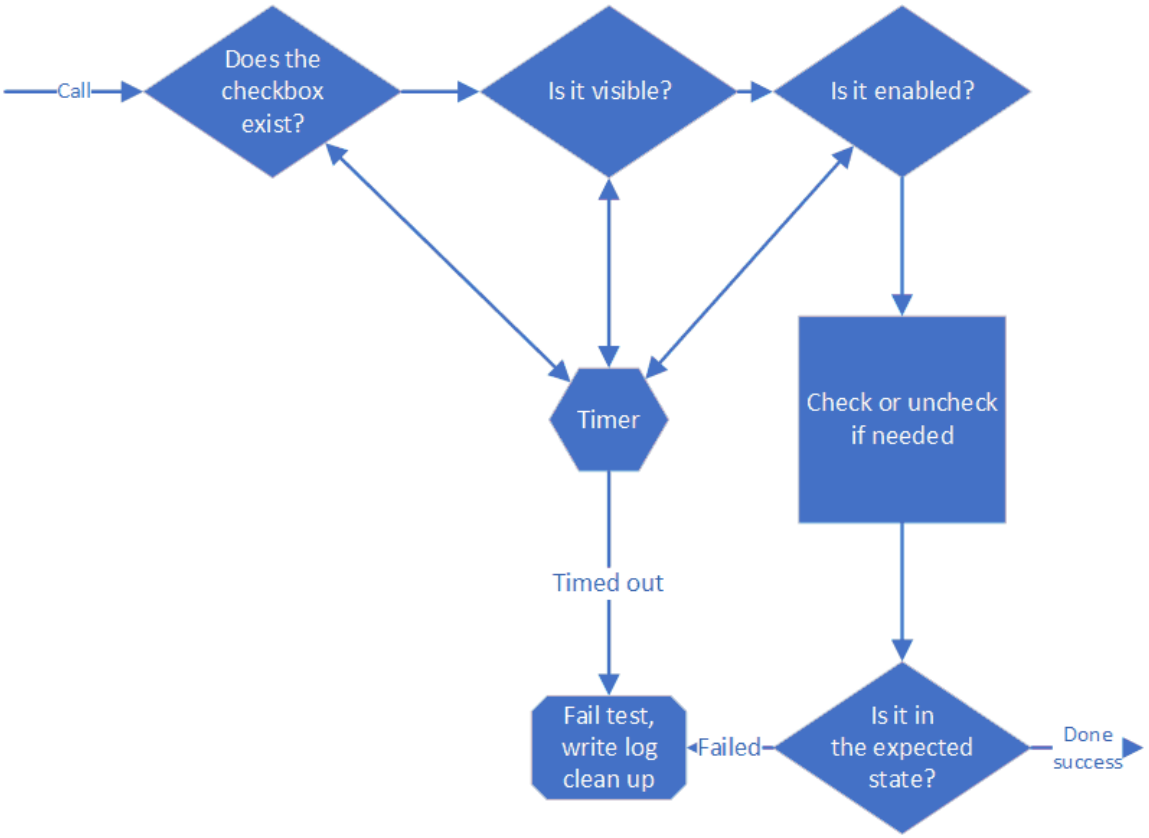

Figura 76: Lógica de las Funciones de Envoltura

Se debe hacer un mayor esfuerzo en el bloque *fallido*. Las instrucciones de registro deben ser exhaustivas para reducir el esfuerzo de resolución de problemas cuando falla una prueba. La funcionalidad de limpieza debe garantizar que el conjunto de pruebas pueda continuar con la siguiente, y la que le sigue después, y la siguiente después de esa prueba. Un automatizador sabio dijo una vez que la prueba más importante que un automatizador necesita considerar es la próxima. Si siempre puede ejecutar la siguiente prueba, puede ejecutar todo el conjunto, sin importar cuántas pruebas fallan durante la ejecución.

Al final del día, los probadores están tratando de averiguar dónde funciona el SUT (y generar confianza cuando parece funcionar correctamente). Si nuestra automatización funciona bien, debemos esperar fallos; necesitamos reunir la información sobre el fallo y pasar a la siguiente prueba.

Recuerde que cada prueba está compilada para examinar el SUT, su entorno y su uso, para que nos diga lo que no sabemos aún. Una prueba que no le diga al probador algo que no sabía no es una prueba valiosa, ya sea automatizada o no. En el programa de certificación de Probador Certificado Nivel Básico de ISTQB®, se abordan técnicas de diseño de pruebas como la segmentación de clases de equivalencia

y el análisis de valores límite. Esas técnicas están diseñadas para reducir la cantidad de pruebas para averiguar la información esencial sobre el SUT lo más rápido posible.

Las envolturas son importantes, pero se pueden compilar otras funciones para mejorar la automatización. Cualquier código que la automatización necesite llamar repetidamente debe estar funcionalizado. Esto sigue las buenas prácticas de desarrollo de descomposición funcional y refactorización. Cree librerías de funciones que se puedan incluir en su código para que usted mismo y sus compañeros de equipo tengan las cajas de herramientas. Recuerde mover todo fuera de los guiones reales a estas librerías invocables siempre que sea posible.

Lleva tiempo y recursos compilar estas funciones. Si desea ser un automatizador exitoso, ésta podría ser la mejor inversión que usted haya hecho.

Al probar un navegador, asegúrese de que al ubicar WebElements con XPath y Selectores de CSS, no utilizar rutas absolutas. Como se analizó en el capítulo anterior, cualquier cambio en el HTML probablemente hará que esas rutas cambien, interrumpiendo la automatización. Si bien las rutas relativas también pueden interrumpirse, tienden a hacerlo con menos frecuencia y, por lo tanto, necesitan menos mantenimiento.

Asegúrese de analizar la automatización con los desarrolladores de su organización. Hágales saber su dolor. Pueden hacer muchas cosas en la forma en que escriben su código para evitar la interrupción constante de su automatización que tiene que ver directamente con el HTML.

Defina nombres globales (variables, constantes, nombres de funciones, etc.) que tengan sentido. Esto hace que el código sea más legible, lo que tiene el efecto secundario de facilitar el mantenimiento del código de automatización. Un mantenimiento de código más sencillo generalmente significa menos defectos de regresión cuando se hacen los cambios. Se necesitan unos segundos más para encontrar buenos nombres, pero puede ahorrar horas en los servicios de fondo.

Comentarios. Muchos comentarios. Comentarios importantes. Muchos automatizadores creen que es su código y recordarán por qué lo escribieron de la manera en que lo hicieron. O piensan que la automatización es «diferente», no es como el código real. Esas personas están equivocadas. Puede olvidar de un día para otro la cosa inteligente que hizo para solucionar un problema. No se oblique a reinventar la rueda. Cuando su proyecto de automatización comience a tener éxito, otras personas comenzarán a poner sus dedos en su código. Hágaselo fácil.

Cree cuentas de prueba y dispositivos específicos para la automatización. No los comparta con los probadores manuales. La automatización necesita tener certeza cuando se trata de los datos utilizados en la automatización; es probable que los probadores manuales al realizar cambios interrumpan su automatización.

Estas cuentas y accesorios no deben vincularse con ninguna persona específica. Las cuentas genéricas que imitan a las cuentas reales son mucho mejores para aislarlas de los cambios. Debe tener suficientes cuentas diferentes para que las pruebas múltiples no interfieran con los datos de los demás.

Cree suficientes datos en estas cuentas para que simulen cuentas reales. No olvide utilizar diferentes

personas para estas cuentas. Si su SUT tiene diferentes tipos de usuarios (p. ej., principiantes, friki de la tecnología, usuarios avanzados, etc.) estos se deben modelar en sus cuentas de prueba.

Analizamos el registro en el capítulo tres, sección uno. Tome el registro en serio. Si su automatización es un éxito, su administración querrá más. Mucho más. Una de las formas de hacer que la automatización sea más escalable es hacer un buen registro desde el principio. No querrá renovar todos, sus cientos (¿miles?) de guiones con un buen registro.

Python tiene un conjunto robusto de recursos de registro; o puede crear el suyo propio. Cuando considere la posibilidad de iniciar sesión, piense hacerlo fuera de la caja de diálogo. Los registros sólidos pueden reducir la resolución de problemas de tiempo cuando ocurren fallos. Tenga presente las necesidades de su organización. Si su SUT es crítico para la misión o para la seguridad, es posible que sus registros deban ser lo suficientemente completos como para ser auditados.

Intente modelar los procesos de pensamiento del probador manual. ¿Qué aprenden y quitan del SUT cuando ejecutan una prueba manual? Vea si puede obtener y registrar esa misma información. Recuerde, puede ser tan creativo como desee cuando determine cómo, cuándo y dónde hacer su registro.

Controle los archivos que cree. Use convenciones de nomenclatura consistentes y guarde los archivos en carpetas consistentes. Considere hacer una carpeta con una marca de tiempo en el nombre y colocar allí todos los archivos de una ejecución. Si varias máquinas están ejecutando la automatización o si se están probando en diferentes entornos, considere agregar el nombre de la estación de trabajo o los nombres de los entornos a los nombres de las carpetas.

Piense en incluir marcas de tiempo en los nombres de los archivos. Eso garantizará que su automatización no sobrescriba los resultados de ejecución de prueba anteriores. Si los archivos no son necesarios en el futuro, colóquelos en directorios que puedan destruirse sin daño alguno. Si los archivos deben guardarse, asegúrese de que la estructura de su carpeta incluya esa información.

Use nombres de archivos consistentes. Reúnase con otros automatizadores y proponga acordar un convenio de estilo y convención de nombres. Enseñele esos estándares y pautas a los nuevos automatizadores incorporados al equipo. No lleva mucho tiempo crear un caos cuando todos hacen lo suyo al crear artefactos persistentes.

Los automatizadores tradicionalmente han sido los rebeldes del desarrollo. Codifique a los vaqueros (y las vaqueras) a quienes les gusta hacer lo suyo y presionar la envoltura. Sin embargo, la inversión en automatización tiende a ser realmente grande como factor, tanto en recursos y como en tiempo. Deben adoptarse normas y pautas mínimas para garantizar que la inversión tenga al menos la posibilidad de ser rentable.

4.2 Mecanismos de Espera

Cuando un probador manual ejecuta una prueba, la consideración de la espera no es realmente un problema. Por ejemplo, supongamos que el probador abre un archivo. Si el archivo se abre de manera oportuna (oportunamente definido por el probador), no se piensa más en ello. Cada vez que un probador manual hace clic en un control o manipula el SUT, hay un reloj implícito en su cabeza. Si piensan que algo lleva demasiado tiempo, es probable que realicen la prueba de nuevo, prestando atención explícita a la temporización.

Algo que nunca ha sucedido, sin embargo, es que un probador sentado durante horas esperando pacientemente que ocurra una acción específica.

Hemos analizado varias veces en este programa de estudios sobre el contexto que un comprobador manual agrega a una prueba. La temporización es una de esas cosas contextuales que debemos entender.

Supongamos que el probador está abriendo un archivo muy pequeño: un par de cientos de bytes. Si no se abre instantáneamente, es probable que el probador esté preocupado y abra un informe de incidencias sobre él. Supongamos que el mismo probador intenta abrir un archivo de dos gigabytes; si demoran treinta segundos en abrirse, es posible que no se sorprendan en absoluto. Si reciben un mensaje informativo de que el archivo demorará en abrirse, cancelarán el mensaje y continuarán esperando sin pestañear.

El contexto importa.

En la automatización, sin importar la herramienta, hay poco o ningún contexto integrado.

Generalmente, la herramienta tiene un tiempo incorporado y está dispuesto a esperar que ocurra una acción. Si la acción esperada no ocurre dentro de ese marco de tiempo, entonces la herramienta registra un fallo y continúa (hacia donde se mueve depende de la herramienta, la configuración y una variedad de otras cosas).

Si la herramienta está configurada para esperar 3 segundos, una acción que ocurra en 3.0001 segundos, quizás dentro del rango de tiempo contextual del probador manual, se considerará un fallo.

Si el automatizador elimina toda consideración de tiempo, entonces es posible que la herramienta literalmente espere por siempre a que ocurra una acción. Hay algunas emociones peores que venir el lunes por la mañana y descubrir que el juego de automatización que configuraró para que comenzara el viernes por la noche antes de partir, está en la prueba dos y ha estado ahí tres minutos después de que usted se marchó el viernes.

Un automatizador debe comprender las necesidades para su automatización y la forma en que funcionan sus herramientas.

Selenium WebDriver con Python tiene varios mecanismos de espera diferentes que un automatizador puede usar al configurar la sincronización para su automatización. Un mecanismo de espera explícito debe usarse raramente pero es una de las formas más comunes que usan muchos automatizadores.

El siguiente código probablemente sea reconocido por casi cualquier persona que haya automatizado alguna prueba:

```
import time
…
time.sleep(5)
```

Figura 77: Una espera explícita

Si bien puede haber ocasiones en que esta es una buena idea, la mayoría de las veces no lo es. Hemos visto tiempos en los que los automatizadores han utilizado tanto que la automatización es más lenta que cuándo un probador manual ejecuta exactamente la misma prueba.

Este tipo de mecanismo de espera siempre está dirigido al peor de los casos posibles. Dado que el peor caso posible no ocurre tan a menudo, la mayor parte de este tiempo de espera simplemente se desperdicia.

Para este curso, asumiremos que la función ***sleep()*** solo se tendrá en cuenta cuando se depure un ejercicio; aparte de eso, no la use.

Selenium WebDriver tiene dos tipos principales de mecanismos de espera: esperas implícitas y esperas explícitas. Nos enfocaremos principalmente en esperas explícitas, por lo que primero descartaremos las esperas implícitas.

Se establece una espera implícita en WebDriver cuando se crea por primera vez el objeto WebDriver. El siguiente código creará el controlador y configurará la espera implícita:

```
driver = webdriver.Chrome()
driver.implicitly_wait(10)
```

Figura 78: Configurar una Espera Implícita

La espera implícita, como se definió anteriormente, estará vigente hasta que se destruya WebDriver. Esta espera implícita instruye a WebDriver a sondear el DOM durante un cierto período de tiempo, cuando intente encontrar un elemento que no se encuentra inmediatamente. La configuración predeterminada es 0 segundos de espera. Cada vez que un elemento no se encuentra inmediatamente, el código anterior le dice a WebDriver que sondee durante diez segundos, preguntando (conceptualmente) cada tantos milisegundos, "¿Ya llegó?". Si el elemento aparece dentro de ese período de tiempo, la secuencia de comandos continúa ejecutándose. La espera implícita funcionará para cualquier elemento o elementos que estén definidos en el guión.

Las esperas explícitas requieren que el automatizador defina (generalmente para un elemento específico) exactamente cuánto tiempo WebDriver debe esperar para ese elemento en particular, a

menudo para un estado específico de ese elemento. Como se mencionó anteriormente, el caso extremo de una espera explícita es el método ***sleep()***. Usar este método es muy parecido al uso de una motosierra en lugar de un bisturí para cirugía cerebral.

Aparte de ***sleep()***, las esperas explícitas son fáciles de usar, ya que los enlaces de Python, Java y C # incluyen métodos convenientes que funcionan con esperas para condiciones esperadas específicas. En Python, estas esperas se codifican utilizando el método WebDriver, ***WebDriverWait()***, junto con una ***ExpectedCondition.*** Las condiciones esperadas se definen para muchas condiciones comunes que pueden ocurrir y se accede al incluir el módulo selenium.webdriver.support como se muestra a continuación:

```
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
```

Figura 79: Importación de métodos de espera explícitos

Lo que hace este código es poner a disposición del automatizador la capacidad de utilizar esperas previstas predefinidas que estén disponibles. Llamar a la espera explícita se puede hacer utilizando el siguiente código (suponiendo que se realizan las importaciones anteriores):

```
wait = WebDriverWait(driver, 10)
element = wait.until(EC.element_to_be_clickable((By.ID, 'someID')))
```

Este código esperará hasta 10 segundos, verificando cada 500 milisegundos, para que un elemento sea identificado por un ***ID*** definido como 'someid'. Si el WebElement no se encuentra después de 10 segundos, el código mostrará una TimeoutException (Excepción de desconexión por tiempo agotado). Si se encuentra el WebElement, se colocará una referencia en el *elemento* variable y el código continuará.

Muchos automatizadores incluirán este código en las funciones de envoltura para que cada elemento esté protegido contra WebElements de lento funcionamiento.

Hay una variedad de estas condiciones esperadas definidas para su uso en Python, que incluyen:

- title_is
- title_contains
- presence_of_element_located
- visibility_of_element_located
- visibility_of
- presence_of_all_elements_located
- text_to_be_present_in_element
- text_to_be_present_in_element_value
- frame_to_be_available_and_switch_to_it
- invisibility_of_element_located
- element_to_be_clickable
- staleness_of
- element_to_be_selected

- element_located_to_be_selected
- element_selection_state_to_be
- element_located_selection_state_to_be
- alert_is_present

También se pueden crear condiciones de espera personalizadas, pero eso está más allá del alcance de este programa de estudios.

4.3 Objetos de Página

Anteriormente en el capítulo, recomendamos que, siempre que sea posible, eliminemos la complejidad de los guiones y lo coloquemos en el TAA/TAF. Vamos a abordar ese tema un poco más en esta sección cuando analicemos los objetos de página, que es representativo de un patrón: el *Patrón de Objeto de Página*.

Un objeto de página representa un área en la interfaz de la aplicación Web con la cual su prueba va a interactuar. Hay tres razones principales para usarlos:

- Crea un código reutilizable que puede ser compartido por múltiples guiones de prueba
- Reducen la cantidad de código duplicado
- Reducen los esfuerzos de costo y mantenimiento para los guiones de automatización de la prueba
- Encapsula todas las operaciones en la GUI del SUT en una capa
- Proporciona una clara separación entre el negocio y las partes técnicas para el diseño de automatización de prueba
- Cuando el cambio ocurre inevitablemente, proporciona un único punto para reparar todos los guiones relevantes

Los Objetos de Página y el Patrón de Objeto de Página se han mencionado muchas veces a través de este programa de estudios. El Patrón de Objeto de Página significa el uso de objetos de página en la arquitectura de automatización de prueba, por lo que estos términos pueden usarse de manera casi intercambiable.

Uno de los principios básicos de la construcción de un TAA sostenible es dividirlo en capas. Una de las capas del TAA general de la ISTQB® (como se define en el programa para TAE) es la Capa de Abstracción de la Prueba; esta capa se encarga de la interacción entre la lógica del caso de prueba y las necesidades físicas de conducir el SUT. Los Objetos de Página son parte de esta capa, generalmente extractar la GUI del SUT. Abstracción aquí significa ocultar detalles de cómo estamos controlando la GUI dentro de las funciones que se utilizan en otras secuencias de comandos. A continuación se muestra un fragmento de código que podría aparecer en una secuencia de comandos sin utilizar el Patrón de Objeto de Página. Tenga en cuenta que no es fácil de leer; los localizadores para controles específicos están íntimamente ligados con el código.

```
first_name = self.browser.find_element_by_css_selector("#id_first_name")
first_name.send_keys("Clem")
last_name = self.browser.find_element_by_css_selector("#id_last_name")
last_name.send_keys("Kaddidlehopper")
password = self.browser.find_element_by_css_selector("#id_password")
password.send_keys("QWERTY")
email = self.browser.find_element_by_css_selector("#id_email")
email.send_keys("test+43@example.com")
product = self.browser.find_element_by_css_selector("#id_the_product")
product.send_keys("test")
self.browser.find_element_by_css_selector('#create_account_form button').click()
self.browser.find_element_by_css_selector('#next-step').click()
self.browser.find_element_by_css_selector('#next-step').click()
self.browser.find_element_by_css_selector('#next-step').click()
```

Figura 80: Algunos Códigos Sin Sentido antes de Objetos de Página

Ahora, supongamos que toma el mismo código después de crear un Objeto de Página:

```
signup_form = homepage.getSignupForm()
signup_form.setName("Clem", "Kaddidlehopper")
signup_form.setPassword("QWERTY")
signup_form.setEmail('test+43@example.com')
signup_form.setProductName('test')
onboarding_1 = signup_form.submit()
onboarding_2 = onboarding_1.next()
onboarding_3 = onboarding_2.next()
items_page = onboarding_3.next()
```

Figura 81: Mismo Código que Utiliza Objetos de Página

Bastante más legible y sucinto.

Todas las cosas en el primer fragmento de código todavía existen. Se acaba de encapsular en el objeto de página y salió a la superficie como llamadas a funciones en el segundo fragmento de código. El Objeto de la Página se vería como algo así:

```
class SignupPage(BasePage):
    url = "http://localhost:8000/account/create/"

    def setName(self, first, last):
        self.fill_form_by_id("id_first_name", first)
        self.fill_form_by_id("id_last_name", last)

    def setEmail(self, email):
        self.fill_form_by_id("id_email", email)

    def setPassword(self, password):
        self.fill_form_by_id("id_password", password)
        self.fill_form_by_id("id_password_confirmation", password)

    def setProductName(self, name):
        self.fill_form_by_id("id_first_product_name", name)

    def submit(self):
        self.driver.find('#create_account_form button').click()
        return OnboardingInvitePage(self.driver)
```

Figura 82: Ejemplo de Objeto de Página

Hemos abstraído gran parte de la complejidad del guión y lo hemos trasladado al TAA. La complejidad sigue ahí, sin duda alguna. La complejidad es lo que nos permite crear una automatización útil y la necesitamos. Al esconderlo de la vista, sin embargo, podemos hacer que sea más fácil para los escritores de guiones crear guiones más rápidos y valiosos.

Este encubrimiento de la complejidad, de tratar con ella donde y cuando queramos, es una mejor práctica en la programación. Tenga en cuenta que, hace muchos años, la primera programación se realizaba conectando cables de conexión que conectaban directamente las entradas al procesador. Más tarde, todos los programadores escribieron en un ensamblador, creando código que esencialmente hablaba el mismo lenguaje que el procesador de la computadora pero que no necesitaba conexiones físicas. Luego vinieron los lenguajes de orden superior, Fortran, Cobol, C, que eran mucho más fáciles de programar; el código fue compilado para crear el código objeto que funcionaría con el procesador, pero el programador no necesitaba saber el microcódigo del procesador. Los lenguajes orientados a objetos, C ++, Delphi, Java, llegaron y lo hicieron aún más fácil.

En esta clase usamos Python, que oculta la mayor parte de la complejidad de tratar con los objetos del navegador. Usar el Patrón de Objeto de Página es solo un paso más.

Un Objeto de Página es una clase, un módulo u otro conjunto de funciones que contiene la interfaz para un formulario, una página or un fragmento de la página del SUT que el automatizador de prueba quiere controlar.

Un Objeto de Página exporta operaciones comerciales que se utilizan como pasos de prueba en la capa de ejecución de prueba. Un Patrón de Objeto de Página es en realidad una de las implementaciones de pruebas controladas por palabras clave, que permite que un proyecto reduzca el esfuerzo de mantenimiento. En la próxima sección analizaremos las pruebas basadas en palabras

clave.

El Patrón de Objeto de Página puede ser especialmente importante para mantener los guiones de prueba y puede reducir sustancialmente el tiempo necesario para actualizarlos después de los cambios en la GUI del SUT.

Tenga en cuenta que el Patrón de Objetos de Página no es una panacea para los problemas con el mantenimiento de la automatización de prueba. Si bien puede reducir los costos, en algunas situaciones, como cambiar la lógica de un SUT, la actualización de los guiones de prueba puede tomar mucho tiempo. Tenga en cuenta que este no es solo un problema de automatización. Si cambia la lógica del SUT, sus casos de prueba manual también deberían cambiar. Afortunadamente, el cambio de la lógica del SUT no ocurre a menudo, porque obligaría a los usuarios de esa aplicación a volver a aprender a usarla.

Al dividir la TAA en capas y diseñar objetos de página, debe observar varias reglas generales:

- Los Objetos de Página no deben contener aserciones comerciales ni puntos de verificación
- Todas las aserciones y verificaciones técnicas con respecto a la GUI (p. ej., verificar si una página ha terminado de cargarse) deben hacerse solo dentro de los Objetos de Página.
- Todas las esperas deben estar encapsuladas en Objetos de Página
- Solo el de Página debe contener llamadas a funciones de Selenium
- Un Objeto de Página no necesita abarcar toda la página, se puede aplicar solo a una sección de un formulario. Puede controlar una sección u otra parte específica de ella.

En una TAA bien diseñada, un Objeto de Página en una Capa de abstracción de prueba encapsula las llamadas a los métodos de Selenium WebDriver, de modo que la Capa de ejecución de prueba solo se ocupe de vínculos de negocio. En tal TAA (es decir, bien diseñada), no hay importación o uso de la librería de Selenium en la capa de ejecución de prueba.

4.4 Prueba Guiada por Palabras Clave

En el capítulo uno, sección dos de este programa de estudios, debatimos la eficacia de un caso de prueba manual. Nosotros escribimos:

> *En su mínima expresión, un guión de prueba manual tiende a tener información en tres columnas. La primera columna contendrá una tarea abstracta a realizar. Resumen, por lo que no es necesario cambiarlo cuando cambia el software real. Por ejemplo, la tarea "Agregar registro a la base de datos" es una idea abstracta. No importa en qué versión de qué base de datos se puede realizar esta tarea abstracta, suponiendo que un tester manual tenga el conocimiento del dominio para traducirlo en acciones concretas*

La idea de una *tarea abstracta* a realizar es una idea muy valiosa. Considere la tarea de abrir un documento en un procesador de textos. La llamaremos *OpenFile*. Tuvimos que abrir un archivo en procesadores de texto en DOS. Windows 3.1. Windows 95. Macintosh. Unix. Cada versión de cada procesador que se haya hecho requería que se realizara esa tarea. El "cómo" abrir un archivo ha cambiado para cada procesador de textos que se haya creado, pero el "qué" sigue siendo el mismo.

OpenFile es un prototipo para una palabra clave.

El quid de las pruebas controladas por palabras clave (KDT) es la idea de que cada aplicación tiene un conjunto de tareas diferentes que deben realizarse para usar la aplicación. Abrimos y guardamos archivos. Creamos, leemos, actualizamos y eliminamos registros de bases de datos. Estas tareas tienden a ser una representación abstracta de lo que un usuario debe hacer para interactuar con el software. Como se trata de ideas abstractas, rara vez cambian a medida que se actualizan las versiones del software.

Las pruebas manuales aprovechan esta abstracción; las tareas funcionales para casi cualquier aplicación rara vez cambian. Una vez que el usuario aprende la interfaz, las tareas funcionales que necesita hacer con la aplicación casi siempre serán las mismas. La forma en que hacen las tareas individuales puede cambiar, pero los conceptos detrás de las tareas no cambian. No es necesario cambiar las pruebas manuales en cada
entrega porque el "qué" que probamos tiende a ser muy estable en comparación con el "cómo".

Las palabras clave que creamos son esencialmente un metalenguaje que podemos usar para las pruebas. Al igual que los probadores manuales que escriben guiones de prueba manuales, identificamos estas tareas de "qué" en el software y las identificamos con un término descriptivo que se convierte en nuestra palabra clave. Los analistas de prueba son las personas ideales para definir las palabras clave que necesitan para probar una aplicación.

Uno de los requisitos previos más importantes para implementar guiones de prueba mantenibles (manuales o automatizados) es una buena estructura donde las tareas que se realizarán permanecen abstractas, de modo que rara vez cambian. Una vez más, encontramos que la separación de la interfaz física de la tarea abstracta es una excelente técnica de diseño.

En el capítulo uno, sección cinco, cuando se describió la Arquitectura de Automatización de Pruebas, enfatizamos la división bien definida de la automatización de pruebas en capas. El caso de prueba que se describe en la capa de definición de prueba en términos abstractos. La capa de adaptación de prueba que interactúa entre el caso de prueba y la GUI física del SUT. La capa de ejecución de prueba que incluye la herramienta que realmente ejecutará las pruebas contra el SUT.

La ISTQB® define a la KDT como una técnica de programación que utiliza archivos de datos que contendrán no solo datos de prueba y resultados esperados, sino también palabras clave (estas sentencias abstractas de lo "qué" se supone que debe hacer el sistema). Las palabras clave a menudo también se llaman *Palabras de Acción*. La definición de la ISTQB® también dice que las palabras clave son interpretadas por guiones de apoyo especiales que son llamados por el guión de control para la prueba.

Veamos esta definición más de cerca. KDT es una técnica programación, lo que significa que es una forma de automatizar las pruebas. Los archivos KDT contienen datos de prueba, resultados esperados y palabras clave.

Las palabras clave son acciones comerciales o pasos de un caso de prueba. Estas son exactamente iguales a la columna uno en un caso de prueba manual.

Al implementar la automatización de pruebas con los principios de KDT, los casos de prueba, en sí mismos, no contienen acciones específicas que deben tomarse en el SUT (el "cómo" de la acción). Los casos de prueba automatizados contienen guiones de prueba que generalmente son acciones comerciales abstractas de alto nivel (p. ej., "iniciar sesión en un sistema", "realizar un pago", "buscar un producto"). Las acciones físicas exactas en el SUT están ocultas dentro de la implementación de las palabras clave (p. ej., en los Objetos de Página como se describe en la sección anterior). Tenga en cuenta que esto duplica la separación entre un caso de prueba manual y el probador humano que va a agregar contexto y razonabilidad a la prueba mientras manipula físicamente la GUI para ejecutar la prueba.

Este enfoque tiene varias ventajas:

- El diseño del caso de prueba está desacoplado de la interfaz del SUT
- Se hace una distinción clara entre la ejecución de la prueba, abstracción de la prueba y las capas de definición de prueba
- Una división del trabajo casi completa:
    - Los analistas de prueba diseñan casos de prueba y escriben guiones usando palabras clave, datos y resultados esperados
    - Los analistas de prueba técnica (automatizadores) implementan las palabras clave y el marco de ejecución necesario para realizar las pruebas
- Reutilización de palabras clave en diferentes casos de prueba
- Legibilidad mejorada de los casos de prueba (se parecen mucho a los casos de prueba manual)
- Menos redundancia
- Costo de mantenimiento y esfuerzo más bajo
- Si la automatización es fuera de línea, un probador manual puede realizar la prueba manualmente directamente desde el guión automatizado
- Dado que las pruebas de palabras clave son abstractas, los guiones que utilizan palabras clave se pueden escribir mucho antes de que el SUT esté disponible para la prueba (al igual que las pruebas manuales)
- La viñeta anterior significa que la automatización puede estar lista antes y ser llevada a pruebas funcionales y no solo a pruebas de regresión
- Unos pocos automatizadores técnicos pueden ayudar a muchos analistas de prueba a compilar pruebas que hagan que la arquitectura sea totalmente escalable
- Debido a que los guiones están completamente separados del nivel de implementación, se pueden usar diferentes herramientas de ejecución de manera intercambiable

A veces, las librerías de palabras clave usan otras palabras clave, por lo que toda la estructura de palabras clave puede volverse bastante compleja. En dicha estructura, algunas palabras clave tendrán un alto nivel de abstracción, p. ej., "Agregue un producto X a la factura con precio Y y cantidad Z", y algunas de ellas tendrán un nivel bastante bajo, por ejemplo, "Click >>Cancel<< button".

Por lo tanto, las palabras clave tienen varios lugares de definición y uso en la arquitectura general de automatización de pruebas:

- Las palabras clave de bajo nivel se implementan como Objetos de Página en la Capa de

adaptación de prueba, hacen las acciones en el SUT.

- Las palabras clave de bajo nivel se utilizan como partes de palabras clave de nivel superior en la Librería de prueba de la Capa de ejecución de prueba
- Las palabras clave de alto nivel se implementan en la Librería de prueba
- Las palabras clave de alto nivel se usan como pasos de prueba de los Procedimientos de prueba en la Capa de ejecución de prueba

La definición de la ISTQB® también dice que las palabras clave son interpretadas por guiones de apoyo especiales. Eso es cierto cuando se implementa la KDT utilizando herramientas especializadas como Cucumber o RobotFramework.

Sin embargo, es posible observar los principios de KDT al implementar un TAF con lenguajes de programación de propósito general como Python, Java o C #. En este caso, cada palabra clave se convierte en la invocación de una función o método de los objetos de la Librería de pruebas.

Hay dos formas de implementar la KDT en la práctica. La primera KDT clásica descrita por Dorothy Graham en su libro de automatización de pruebas[2] es un enfoque de arriba hacia abajo. Su primer paso es diseñar casos de prueba, ya que estarían diseñados para pruebas manuales. Luego, los pasos de estos casos de prueba se abstraen para convertirse en palabras clave de alto nivel, que pueden descomponerse en palabras clave de bajo nivel o implementarse en una herramienta o un lenguaje de programación. Este método es más rentable cuando la Librería de pruebas ya contiene muchas funciones/métodos que realizan tareas contra el SUT. También puede ser útil cuando los guiones automáticos se convierten a partir de casos de prueba manuales en lugar de escribirse desde cero.

La segunda forma es la implementación de guiones en una dirección ascendente. Esto significa que los guiones son grabados por una herramienta (p. ej., Selenium IDE) y luego son refactorizados y reestructurados en una arquitectura de automatización de prueba KDT adecuada. Este enfoque le permite a los equipos de prueba escribir rápidamente varios guiones de automatización de prueba y ejecutarlos contra el SUT.

Desafortunadamente, escribir más de veinte o treinta guiones usando este enfoque chapucero sin refactorizarlos como un TAF bien diseñado expondrá el proyecto de automatización a un riesgo de costos y esfuerzos de mantenimiento en el futuro. Además, las pruebas que se escriben por primera vez sin seguir los casos de prueba manual corren el riesgo de no ser buenas pruebas (es decir, pueden no mitigar los riesgos importantes que deben probarse).

Al implementar la TAF basándose en los principios de KDT, considere también los siguientes aspectos:

- Las palabras clave de grano fino permiten escenarios más específicos, pero a costa de la complejidad del mantenimiento de guiones
- Cuanto más precisa sea una palabra clave, más probable es que esté íntimamente ligada a la interfaz del SUT y menos abstracta (p. ej., el ejemplo anterior, “Click >>Cancel<< button”).
- El diseño inicial de palabras clave es importante, pero en última instancia se necesitarán palabras clave nuevas y diferentes, lo que implica tanto la lógica empresarial como la funcionalidad de la automatización para ejecutarla.

Cuando se hace correctamente, la KDT ha demostrado ser un buen enfoque de automatización que puede producir guiones con costos de mantenimiento consistentemente bajos. Permite construir marcos de automatización de pruebas bien estructurados y utiliza las mejores prácticas de diseño de software.

---

2 "Software Test Automation", Mark Fewster and Dorothy Graham, Addison-Wesley Professional, 1999

Apéndice - Glosario de Términos de Selenium

Atributo de clase: Un atributo HTML que apunta a una clase en una hoja de estilo CSS. También puede ser utilizado por un JavaScript para realizar cambios en elementos HTML con una clase especificada.

Comparador: Una herramienta para automatizar la comparación de los resultados esperados con los resultados reales.

Selector de CSS: Los selectores son patrones dirigidos a los elementos HTML que desee diseñar.

Modelo de Objetos de Documento (DOM): Una interfaz de programación de aplicaciones que trata un documento HTML o XML como una estructura de árbol en la que cada nodo es un objeto que representa una parte del documento.

Accesorio: Un accesorio de prueba es un objeto o entorno simulado que se utiliza para probar constantemente algún elemento, dispositivo o elemento de software.

Marco: Proporciona un entorno para la ejecución de guiones de prueba automatizados; incluyendo herramientas, librerías y accesorios.

Función: Una función de Python es un grupo de declaraciones reutilizables que realizan una tarea específica.

Anzuelo: Una interfaz que se introduce en un sistema que se crea predominantemente para proporcionar una capacidad de prueba mejorada para ese sistema.

HTML (Lenguaje de Marcas de Hipertexto): El lenguaje de marcado estándar para crear páginas web y aplicaciones web.

ID: Un atributo que especifica una cadena de identificación única para un elemento HTML. El valor debe ser único dentro del documento HTML.

iframe: Un marco HTML en línea, utilizado para incrustar otro documento dentro de un documento HTML.

Diálogo modal: Una ventana o cuadro que obliga al usuario a interactuar con esta antes de que pueda acceder a la pantalla subyacente.

Patrón de Objetos de Página: Un patrón de automatización de prueba que requiere que la lógica técnica y la lógica de negocios se traten en diferentes niveles.

Paradoja del pesticida: Un fenómeno en el que repetir la misma prueba varias veces hace que encuentre menos defectos.

Persona: Un perfil de usuario creado para representar un tipo de usuario que interactúa con un sistema de una manera común.

Pytest: Un marco de prueba de Python.

Etiqueta: Los elementos HTML están delineados por etiquetas, escritos usando paréntesis en ángulo.

Deuda técnica: Implica el costo adicional de la restauración causada por la elección de ignorar malos diseños o implementaciones en el corto plazo.

WebDriver: La interfaz contra la cual se escriben las pruebas de Selenium. Se pueden controlar

diferentes navegadores a través de diferentes clases de Java, por ejemplo, ChromeDriver, FirefoxDriver, etc.

Envoltura: Una función en una librería de software cuyo principal propósito es llamar a otra función, a menudo añadiendo o mejorando la funcionalidad mientras oculta la complejidad.

XML (Lenguaje de Formateo Extensible): Un lenguaje de marcado que define un conjunto de reglas para codificar documentos en un formato legible por el hombre y legible por máquina.

XPath (Lenguaje de Ruta XML): Un lenguaje de consulta para seleccionar nodos desde un documento XML.