Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Pruebas funcionales automatizadas para aplicaciones Web: Usando Selenium para aplicar pruebas de regresión automatizadas

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Adriano Tobías Vega Llobell

Tutores: Patricio Orlando Letelier Torres

María Carmen Penadés Gramage

Curso 2017-2018

Resumen

En este trabajo se aborda la definición e implantación de un proceso de pruebas funcionales automatizadas para aplicaciones web en el ciclo de desarrollo de una empresa. Dicho proceso cubre el ciclo completo de las pruebas: la automatización, la ejecución y la posterior revisión de sus resultados. Además, se realiza el diseño y desarrollo de una herramienta basada en Selenium para dar soporte a este proceso, junto con una batería de pruebas para una de las aplicaciones de la empresa.

Palabras clave: Selenium, pruebas automatizadas, aplicaciones web, desarrollo de software, mejora de proceso.

Abstract

This paper describes the definition and implementation of a process for automated functional testing of web applications in the development cycle of a company. This process covers the whole tests cycle: the automation, execution and posterior revision of the results. In addition, a tool based on Selenium is designed and developed to support this process, alongside a test suite for one of the company's applications.

Keywords: Selenium, automated tests, web applications, software development, process improvement.

Tabla de contenidos

1 Introducción 8
1.1 Motivación 8
1.2 Objetivos 8
1.3 Relación con trabajos anteriores 9
1.4 Estructura del trabajo 9
2 Pruebas funcionales para aplicaciones web 11
2.1 Introducción a las pruebas del *software* 11
2.2 Pruebas funcionales para aplicaciones web 13
2.2.1 Elección de los navegadores 14
2.2.2 Diseño *Responsive* y diseño adaptativo 14
2.3 Estrategia elegida para las pruebas 15
3 Estado del arte. Herramientas para pruebas funcionales web 16
3.1 Requisitos de la herramienta 16
3.2 Comparativa entre las herramientas 17
3.2.1 Entorno basado en JavaScript: Puppeteer, Mocha y Chai 18
3.2.2 Katalon Studio 20
3.2.3 Proyecto *Ad hoc* basado en Selenium 23
3.2.4 Conclusiones de la comparativa entre las herramientas 25
4 Propuesta para automatización de pruebas basada en Selenium 26
4.1 Tecnologías utilizadas 26
4.2 Estructura de la solución .NET 26
4.2.1 El proyecto *Base* 27
4.2.2 El proyecto *Application* 29
4.2.3 El proyecto *ApplicationTests* 32
4.2.4 El proyecto *Launcher* 34
5 Automatización de pruebas funcionales 37
5.1 Diseño de la prueba 38
5.1.1 Acciones previas 38
5.1.2 Pasos de la prueba 39
5.1.3 Resultado esperado 42
5.2 Automatización de la prueba 43

5.3 ¿Pruebas "desechables" o pruebas "estructuradas"? 48
6 Proceso de pruebas automatizadas para aplicaciones web 50
6.1 Actividad de desarrollo iterativo e incremental 50
6.2 Automatización y mantenimiento de la batería 52
6.3 Lanzamiento de la batería y revisión 52
6.3.1 Lanzamiento de la batería 52
6.3.2 Revisión de los resultados de la batería 53
6.4 Gestión de los fallos 57
7 Validación del proceso y la herramienta 59
7.1 Automatización de pruebas 59
7.1.1 Deficiencias encontradas por el *tester* 60
7.2 Lanzamiento de la batería de pruebas 60
7.2.1 Automatización de pruebas 60
7.2.2 Estadísticas de ejecución de la batería durante el periodo de preproducción 61
7.2.3 Fallos detectados 62
8 Resumen y cronograma del desarrollo 64
9 Conclusiones y trabajos futuros 66
9.1 Conclusiones 66
9.2 Relación con asignaturas cursadas en la carrera 67
9.3 Conclusiones personales 68
9.4 Trabajos futuros 68
9.4.1 Integración con la infraestructura existente 68
9.4.2 Ejecución de las pruebas en entornos móviles 69
10 Referencias 70
Anexos 71
I. Ejemplo del código de una prueba no estructurada 71
II. Ejemplo de código de una prueba *estructurada* 75

Tabla de figuras

Figura 1: Modelo en V para el control de la calidad del software. Obtenida de [7]. 11
Figura 2: Pirámide del testing. Basada en la imagen de [10]. 13
Figura 3: Estadísticas de uso de navegadores en el escritorio (izquierda) y móvil (derecha) a marzo de 2018, en porcentaje de usuarios. [13]..14
Figura 4: Ejemplo de visualización de una aplicación web adaptada a distintos dispositivos. 15
Figura 5: Resultado de la ejecución de la prueba del entorno basado en JavaScript. ... 20
Figura 6: Lista de acciones registradas por Katalon Studio con el grabador.................21
Figura 7: Resultado de la ejecución de la prueba de ejemplo en Katalon Studio. 22
Figura 8: Ejemplo de metadatos inválidos que se utilizaban en la búsqueda del enlace, e impedían que se completara correctamente. 22
Figura 9: Resultado de la ejecución de la prueba de Selenium usando MSTest. 24
Figura 10: Esquema de la estructura interna los proyectos que componen la herramienta. Las flechas indican dependencia entre proyectos. 27
Figura 11: Flujo de ejecución de una prueba. 28
Figura 12: Ejemplo de la estructura en árbol del proyecto Application. 29
Figura 13: Ejemplo de formulario del que querríamos obtener datos. Resaltado en rojo sale el campo de ejemplo. 30
Figura 14: Tabla de controles de diuresis, de la que deseamos obtener una fila. 31
Figura 15: Ejemplo de la estructura de un proyecto ApplicationTests......................... 33
Figura 16: Diagrama que representa el orden de sucesos en la ejecución de una batería de pruebas. 35
Figura 17: Ejemplo de código para programar el lanzamiento de dos pruebas, en Chrome y Edge en las resoluciones Desktop y Tablet. 36
Figura 18: Contenido de la carpeta de logs de la batería de ejemplo. 36
Figura 19: Log resultante de la ejecución de la batería de pruebas de ejemplo. 36
Figura 20: Prueba de aceptación que se tomará de ejemplo para la automatización.....37
Figura 21: La opción 'Impedir la introducción de controles de enfermería de fecha futura', necesaria para la prueba, marcada. 38
Figura 22: Pantalla de inicio de sesión de la aplicación. Iniciaremos sesión con el usuario TestUser02. 39
Figura 23: Vista del home de la aplicación para un usuario genérico.......................... 40
Figura 24: Barra de selección de residente, mostrando el residente seleccionado actualmente. 40
Figura 25: Menú de la sección de controles de enfermería. Se muestran todos los controles para los que el usuario cuenta con permiso de acceso. 40
Figura 26: Ventana del control de diuresis. 41
Figura 27: Diálogo para la creación de un nuevo registro de diuresis. 41
Figura 28: Mensaje de alerta, avisando que no se ha podido insertar el control de diuresis con fecha futura. 42
Figura 29: Fragmento del código HTML de la pantalla LogIn, que representa el campo Nombre de Usuario y el botón de inicio de sesión. 44
Figura 30: Localizador XPath para detectar el campo de nombre de usuario.............. 44
Figura 31: Fragmento de código HTML que representa la lista de opciones del Home. 45
Figura 32: Ejemplo de cómo seleccionar la opción Controles en la página Home. 45

Figura 33: Log resultante de ejecutar la prueba automatizada......................................47
Figura 34: Operaciones que ofrece el Page Object para el control de enfermería de diuresis. 49
Figura 35: Representación de las actividades que se llevan a cabo durante el desarrollo de un sprint. Basado en imagen de [22]..51
Figura 36: Ejemplo de informe que se envía al completar la revisión de la batería. 55
Figura 37: Ejemplo de reporte de un fallo de la aplicación web, detectado durante la automatización de pruebas... 56
Figura 38: El proceso seguido con los fallos detectados durante el proceso de desarrollo. Basado en diagrama proveniente de [22]. 57
Figura 39: Gráfica que representa los resultados finales de la ejecución de la batería de pruebas, mostrando el total de pruebas ejecutadas y el resultado.................................61
Figura 40: Gráfica que representa el número de pruebas que han fallado, agrupadas por el motivo de fallo. 62
Figura 41: Gráfica que representa el número de fallos detectados durante la automatización de pruebas según el mes. 62
Figura 42: Gráfica que representa el número de fallos detectados agrupados por la severidad... 63
Figura 43: Cronograma que representa las fases del trabajo desarrollado.................... 64

Tabla de tablas

Tabla 1: Resumen de los resultados de la compartiva entre las distintas herramientas.25

1 Introducción

1.1 Motivación

Las aplicaciones web, o *web apps,* son aquellas aplicaciones que están basadas en tecnologías web para realizar tareas a través de Internet. Muchas de ellas se ejecutan dentro de los propios navegadores [1]. Gracias a esto, se puede dar soporte a un amplio abanico de dispositivos, como ordenadores, móviles o *tablets*, sin necesidad de instalación por parte del usuario. Además, si la aplicación dispone de almacenamiento "en la nube", nos permitirá acceder a nuestra información almacenada desde cualquier lugar en el mundo.

Hoy en día, su uso por parte de empresas para ofrecer servicios a través de Internet está muy extendido: redes sociales, como Facebook y Twitter; o suites ofimáticas on-line, como Google Docs. Uno de los principales sectores que las utilizan es el comercio electrónico, o *e-commerce*. Ya sean escaparates virtuales, servicios de reserva, u otros, han tenido un importante impacto en los hábitos de consumo de las personas. Se estima que, en el año 2017, las ventas del comercio electrónico a nivel mundial supusieron un total 2,3 billones de euros [2].

Como la mayoría de los programas, las *web apps* suelen estar sometidas a constantes cambios y mejoras: nuevas funcionalidades, adaptaciones a nuevos entornos, etc. Este mantenimiento conlleva el riesgo de introducir defectos en el producto. Para detectarlos, deberían realizarse pruebas frecuentemente, que se dividirán en dos frentes distintos: por un lado, deberán ayudarnos a comprobar el correcto funcionamiento de las nuevas funcionalidades; y por otro, deben asegurar que el comportamiento que ya funcionaba no se ha visto afectado. Esto último se denomina pruebas de regresión y deberían aplicarse frecuentemente.

Hacerlo manualmente puede resultar ineficiente. De ahí surge el interés por su automatización. Gracias a la automatización de pruebas se puede ejecutar un gran número de pruebas en cualquier momento, mucho más rápido de lo que podrían aplicarse manualmente. Por ello, es importante integrar dentro del proceso de desarrollo la automatización de pruebas y su posterior ejecución y revisión.

Este trabajo de final de grado (TFG) se ha desarrollado en el marco de prácticas en una empresa que desarrolla un sistema ERP para el sector sociosanitario. Durante la realización de las prácticas, surgió la necesidad de contar con pruebas de regresión automatizadas para aplicaciones web complementarias al ERP, lo que generó interés por el desarrollo de este.

1.2 Objetivos

El propósito de este TFG es establecer un proceso de automatización de pruebas funcionales para aplicaciones web e integrarlo en el proceso de desarrollo de la empresa donde se han realizado dichas prácticas. Debía abarcar el ciclo completo de las pruebas: la automatización, la ejecución y la posterior revisión de sus resultados. Para ello, se tomará como base una aplicación web de la empresa, a la cual se le aplicará todo el proceso descrito anteriormente.

Otro de los objetivos fue el establecer herramientas que den soporte a dicho proceso. Después de analizar varias alternativas disponibles, como se explicará en capítulos posteriores, se decidió por el desarrollo de una herramienta *ad hoc* basada en Selenium.

Se propuso también desarrollar una batería de pruebas funcionales para una de las aplicaciones de la empresa, que contara con alrededor de unas cincuenta pruebas. Así se podría verificar el correcto funcionamiento de la herramienta, y podría aplicarse en condiciones reales el proceso definido.

1.3 Relación con trabajos anteriores

Algunos de los trabajos relacionados con este proyecto son, por ejemplo, [3] y [4]. En [3] se describe el desarrollo de un marco de trabajo para la automatización de pruebas funcionales para aplicaciones web. En ese caso, se utilizó una tecnología distinta, Protractor[1], y se centra solamente en las aplicaciones desarrolladas con AngularJS[2]. También se integró con una herramienta de integración continua, para ejecutar estas pruebas cada vez que se realizara un cambio en la aplicación.

En cambio, [4] trata la definición de un proceso de pruebas automatizadas. En ese caso, para pruebas funcionales de aplicaciones de escritorio usando IBM Rational Functional Tester[3]. Dicho proceso cubría el ciclo completo de las pruebas: la definición del caso de prueba, la automatización, el lanzamiento y la revisión. Aun así, en este caso no se aborda en profundidad la integración proceso definido con el proceso de desarrollo.

1.4 Estructura del trabajo

En el capítulo 2 – *Pruebas funcionales para aplicaciones web*, se hace una pequeña introducción a las pruebas funcionales y al estado del arte en el ámbito de las aplicaciones Web. Se describirá la estrategia de pruebas establecida para la aplicación web que tomaremos como base.

A continuación, en el capítulo *3 – Herramientas disponibles para pruebas funcionales web,* se presentará una comparativa entre algunas de las herramientas más utilizadas. Esta comparativa se basará en la automatización de una prueba de ejemplo con cada una de ellas, resaltando así sus características.

En el capítulo *4 - Propuesta para automatización de pruebas basada en* Selenium, se explica el diseño y la estructura de la herramienta desarrollada para soportar el proceso de pruebas automatizadas. Se detallará el papel de cada uno de los módulos que lo componen y se describirá mediante un ejemplo el funcionamiento de la ejecución de una batería.

---

[1] Protractor – Página oficial: https://www.protractortest.org
[2] AngularJS – Página oficial: https://angularjs.org/
[3] IBM Rational Functional Tester – Página oficial: https://www.ibm.com/us-en/marketplace/rational-functional-tester

En el capítulo *5 - Automatización de pruebas funcionales,* se explicará mediante un ejemplo el proceso completo de la automatización de una prueba. También se describirá su implementación, y se hará una descripción de las ventajas de utilizar el patrón *Page Object* para automatizar pruebas "estructuradas".

En el capítulo *6 - Proceso de pruebas automatizadas para aplicaciones web*, se describe la integración del proceso de pruebas automatizadas dentro del ciclo de desarrollo. Se hará hincapié en el lanzamiento de la batería y su posterior revisión durante el periodo previo al lanzamiento de una versión.

En el capítulo *7 - Validación del proceso y la herramienta,* se presentan las pruebas realizadas sobre el propio proceso y la herramienta. Por ejemplo, el desarrollo de una prueba por un *tester* externo al proyecto. También se presentan estadísticas del lanzamiento de la batería y los fallos detectados.

En el capítulo *8 - Resumen y cronograma del desarrollo,* se presenta una recapitulación del desarrollo del proyecto. Es un resumen de las fases por las que pasó, como la definición de la estrategia de pruebas o el desarrollo e implantación del proceso y la herramienta.

Finalmente, en el capítulo *9 - Conclusiones y trabajos futuros,* se presentan las conclusiones sobre el trabajo realizado y una reflexión sobre los objetivos cumplidos. También se detallarán las posibles mejoras o trabajos futuros que se puedan realizar sobre la herramienta y el proceso.

2 Pruebas funcionales para aplicaciones web

2.1 Introducción a las pruebas del *software*

La necesidad de probar el *software* surge de la necesidad de crear *software* de calidad. Es decir, un programa que logre cumplir con las necesidades del cliente, buscando también que se logre dentro del presupuesto y el tiempo acordado. La validación y la verificación de los productos *software* son aspectos esenciales en el control de calidad: necesitamos comprobar que el producto que estamos desarrollando es acorde a la especificación (*verificar*), y que cumple las necesidades del cliente (*validar*) [5].

El modelo en V, mostrado en la Figura 1, es un modelo de proceso para la validación y verificación del *software* basado en el clásico ciclo en cascada. Define los tipos de pruebas de que se deben aplicar para comprobar el trabajo realizado en las distintas fases de desarrollo [6]. Las pruebas asociadas a cada una de las fases deberán definirse, preferiblemente, durante las mismas. Respecto al orden de prueba, estas pruebas se ejecutarán en orden secuencial desde el momento en que termina la fase de codificación, empezando por las pruebas unitarias y acabando en las pruebas de aceptación.

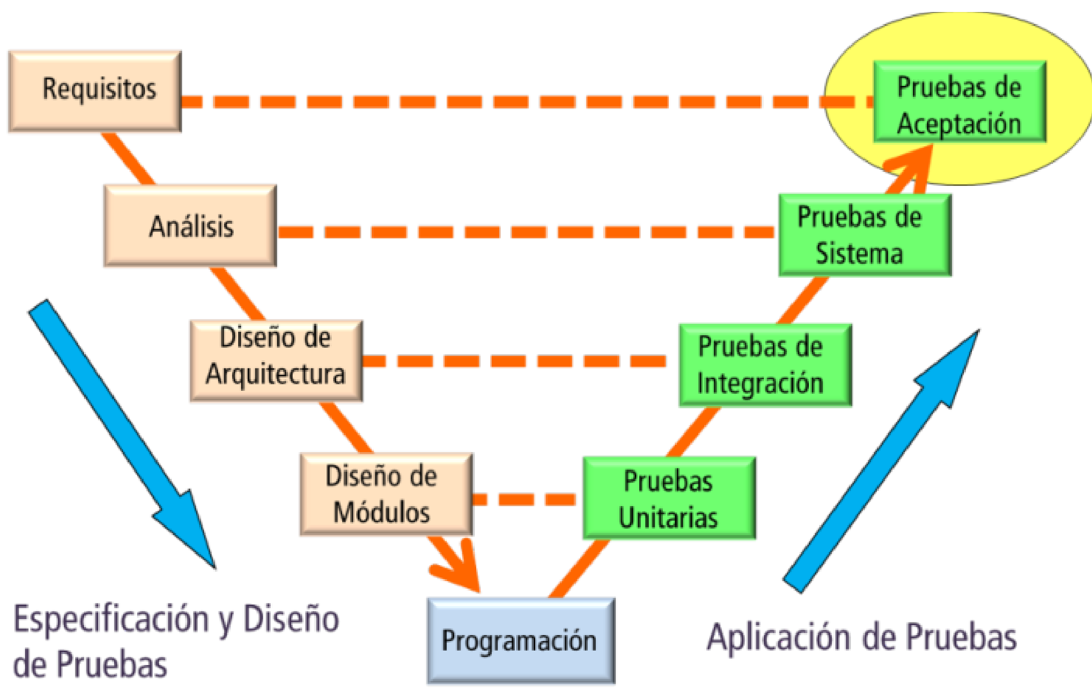

Figura 1: Modelo en V para el control de la calidad del software. Obtenida de [7].

La definición de las pruebas comienza en el análisis del sistema, donde a partir de la especificación de los requisitos, deberemos especificar las pruebas de aceptación (PA). Una PA se define como una serie de condiciones que debe pasar el producto para poder considerar que cumple con un requisito del cliente.

Cada prueba está asociada a un escenario de uso desde el punto de vista del usuario. A partir de un estado determinado del sistema, se define una secuencia de pasos de un caso de uso y el resultado esperado de esta ejecución. Las PA pueden definirse tanto para requisitos funcionales como no funcionales [7]. Estas pruebas tienen como objetivo la validación del producto.

La principal desventaja de este tipo de prueba es que deben hacerse normalmente sobre la interfaz gráfica de usuario. Suelen requerir mantenimiento, debido a que son muy vulnerables a cambios en la interfaz o la funcionalidad [8]. Estos cambios pueden alterar completamente el recorrido de nuestras pruebas y provocar falsos fallos. Será necesario actualizarlas para contemplar el nuevo comportamiento. Aun así, es recomendable automatizar aquellas que ejerciten el comportamiento esencial y, además, sean lo más rápidas posibles.

A partir de la definición de una PA, podemos diseñar uno o más casos de prueba, instanciaciones de la prueba con valores concretos. Se generarán variando las distintas condiciones de entrada del sistema y modificando, si fuera necesario, el resultado esperado de dicha ejecución [7]. Estas serán principalmente las pruebas funcionales en las que se centrará el trabajo posterior de esta memoria.

A continuación, se definen las pruebas que realizan comprobaciones desde el punto de vista del equipo de desarrollo. Este tipo de prueba está más orientado a la búsqueda de fallos de aplicación, por lo que son pruebas de verificación.

Basándonos en la descomposición del sistema en sus distintos módulos y la forma en que interactúan entre ellos, definiremos las pruebas de integración. En ellas realizaremos comprobaciones relacionadas a la interacción entre dos o más módulos distintos y el flujo de datos entre ellos [9]. Podría considerarse una prueba de integración el comprobar que, en la interacción de un módulo de mensajería con otro de impresión, al seleccionar la opción de “Imprimir mensaje” se genere un documento y se envíe correctamente al módulo de impresión.

Por otro lado, las pruebas de integración pueden resultar difíciles de orquestar. Como su objetivo no es probar el sistema completo, tendremos que emular el comportamiento de algunas de sus partes, mediante *stubs* [6]. Para lograrlo, deberemos tener presente durante la fase de diseño la intención de realizar estas pruebas, y diseñar el sistema modular. Así podremos sustituir elementos para hacer las pruebas de forma aislada. Las conexiones a bases de datos serían un ejemplo de ello, donde simularíamos la información que provendría de una conexión real.

Finalmente, el último tipo de pruebas de este modelo son las pruebas unitarias. Estas se definen a partir de la especificación. Son las más básicas, ya que comprueban el funcionamiento de pequeñas secciones del código [10].

Existen distintos criterios para desarrollar las pruebas unitarias, según la cobertura que busquemos alcanzar. La cobertura indica el porcentaje del código que está incluido en al menos una prueba. Por ejemplo, la cobertura a nivel de sentencia, donde basta ejecutar todas las instrucciones al menos una vez. Otra es la cobertura de camino, donde buscamos cubrir los posibles caminos que puede seguir la ejecución del código [6].

Entre las ventajas de realizar pruebas unitarias, tenemos que son las más fáciles de definir e implementar por su sencillez: se concentran en una parte muy específica del código (una unidad), y no necesitan probarse utilizando la interfaz de usuario. También, en algunos entornos de desarrollo, se podrán ejecutar en el momento en el que modifiquemos el código[4]. Así nos será más fácil detectar rápidamente las situaciones en las que un cambio pudiera provocar un fallo.

En conclusión, cada tipo de prueba tiene un propósito distinto. Ninguna de ellas será suficiente por sí misma para desarrollar una estrategia de pruebas efectiva para un producto *software*. Según el proyecto, habremos de priorizar que tipos se ajustan más a las características de este: ya sea dependiendo de parámetros como el tamaño, complejidad, modularidad, etc.

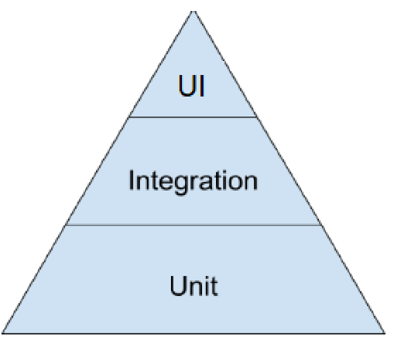

Figura 2: Pirámide del testing. Basada en la imagen de [10].

En el caso de las aplicaciones web, ingenieros de Google recomiendan aproximadamente una distribución 70/20/10: 70% pruebas unitarias, 20% de integración y 10% de aceptación [10]. Así, la distribución total de las pruebas debería formar una pirámide, como podemos ver en la Figura 2.

2.2 Pruebas funcionales para aplicaciones web

En el ámbito de las aplicaciones web, se pueden aplicar distintos tipos de pruebas, según los objetivos de la estrategia que hayamos definido. Por ejemplo, se puede probar el funcionamiento de las actividades críticas para el usuario (pruebas funcionales); el acceso a ficheros que requieran autorización (pruebas no funcionales de seguridad); o si la aplicación se adapta a los estándares de accesibilidad, como soportar *alt text* en las imágenes lectores *text-to-speech* usados por los usuarios invidentes (pruebas no funcionales, usabilidad).

Para la realización de este proyecto, nos centraremos en las pruebas funcionales. Las más sencillas que se pueden realizar son las de navegación, donde simplemente probamos el desplazamiento entre los distintos formularios a los que un usuario accedería normalmente [11]. Esto nos servirá para comprobar la correcta carga de las páginas, para detectar algún error inesperado durante la misma.

También debería considerarse la implementación de pruebas básicas de interacción, como podría ser introducir datos en un formulario y comprobar que no ocurre ningún fallo inesperado [11]. Estas deben también ser sencillas, y realizar el mínimo imprescindible de comprobaciones para verificar el correcto funcionamiento.

A lo largo de este apartado se describirán otros aspectos que pueden afectar al proceso de automatización de pruebas de navegador, estos serán importantes a la hora de definir la estrategia de pruebas.

---

[4] Por ejemplo, en Visual Studio 2017 está la característica *Live Unit Testing*.
Página Oficial - https://docs.microsoft.com/es-es/visualstudio/test/live-unit-testing

2.2.1 Elección de los navegadores

Una de las principales preocupaciones en el desarrollo web es el correcto funcionamiento en el máximo número de navegadores y revisiones posible. Pese a que ya no son tan frecuentes como antaño, pueden darse incompatibilidades de nuestra aplicación web con algunos navegadores, ya sea debido a que la implementación de algunas características es diferente, distintos niveles de soporte de tecnologías, entre otras [12].

De aquí surge la necesidad de que hayamos analizado previamente y establecido los navegadores para los que queremos ofrecer soporte. Esto puede condicionar tanto el desarrollo como la estrategia de pruebas. Tendremos entonces, que planificar la ejecución de las pruebas en cada uno de los navegadores soportados.

Usualmente se suele preparar la ejecución de pruebas en los principales navegadores. En la Figura 3 podemos ver las estadísticas de los navegadores más usados en a fecha de marzo de 2018. Chrome, Firefox e Internet Explorer son los más usados en escritorio. Conjuntamente son un 85,42% de la cuota de mercado [13]. En cuanto a dispositivos móviles, Chrome, Safari y UC Browser suman la mayor cuota de mercado.

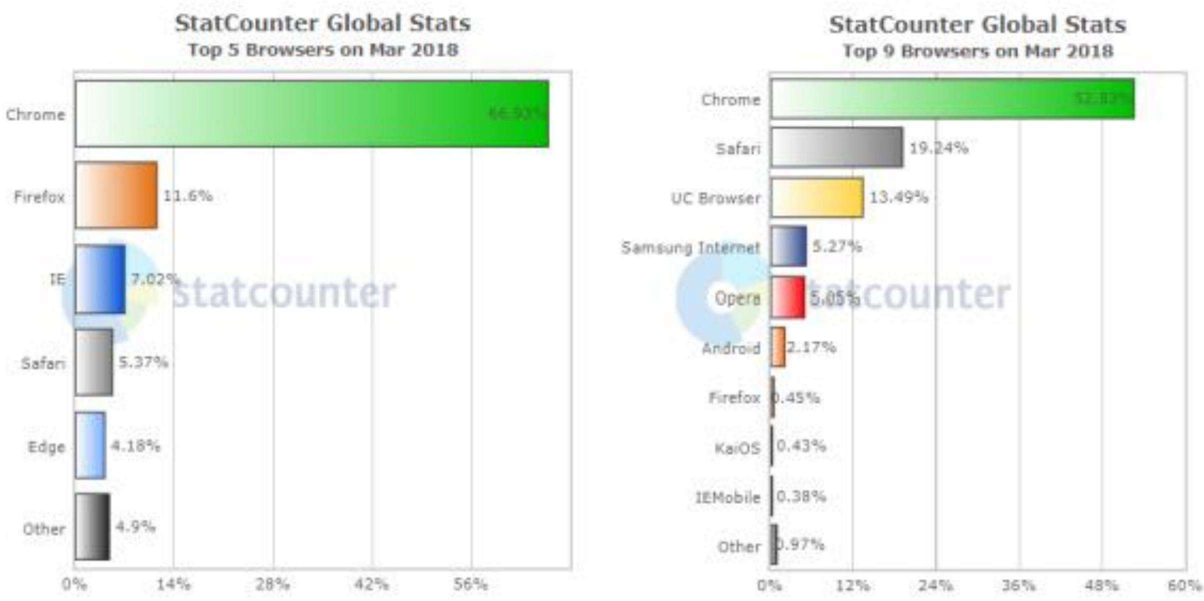

Figura 3: Estadísticas de uso de navegadores en el escritorio (izquierda) y móvil (derecha) a marzo de 2018, en porcentaje de usuarios. [13]

2.2.2 Diseño *Responsive* y diseño adaptativo

Las aplicaciones web suelen tener como objetivo ofrecer el mismo servicio a un amplio abanico de dispositivos, como pueden ser equipos de escritorio y dispositivos móviles. Puede haber grandes diferencias de resolución y relación de aspecto entre ellos, como en la Figura 4, y necesitamos asegurarnos de que la aplicación se adapte correctamente. Para ello, hay distintas alternativas, entre ellas: El diseño *responsive* y el diseño adaptativo. Según el diseño que se haya adoptado en la aplicación, necesitaremos optar por una estrategia u otra para probarla.

El diseño *responsive* consiste en ofrecer la misma interfaz para distintos tamaños de pantalla. El dispositivo se encargará de adaptar la forma de mostrar los contenidos según su resolución [14]. En este caso, es probable que no necesitemos modificar las pruebas, y podamos utilizar las ya existentes. Aun así, será recomendable ejecutarlas también en resoluciones en dispositivos móviles, para detectar elementos que no se adapten bien. Por ejemplo, un botón podría quedar oculto detrás de una imagen que no ha reducido correctamente su tamaño.

Figura 4: Ejemplo de visualización de una aplicación web adaptada a distintos dispositivos.

En cambio, el diseño adaptativo consiste en diseñar una o más interfaces distintas para determinados tamaños de pantalla. El servidor suministrará la correcta según el dispositivo o navegador que utilice el cliente [14]. Por ejemplo, se podría tener una interfaz para escritorio y otra para dispositivos móviles. En cuanto a las pruebas, será necesario automatizar una distinta para cada interfaz, lo que puede aumentar considerablemente el trabajo.

2.3 Estrategia elegida para las pruebas

Teniendo en cuenta los factores que podían afectar a la automatización de pruebas descritos anteriormente, se procedió a definir una estrategia inicial para las pruebas de aplicaciones web. Se planificó basándose en una aplicación concreta de la empresa. Es muy probable que la estrategia evolucione una vez comience la automatización de pruebas para otras aplicaciones, para adaptarse a sus requisitos.

La aplicación en concreto es una aplicación complementaria de un ERP sociosanitario. Consiste en un portal web que permite la introducción de datos, como pueden ser controles de enfermería. En el momento de redacción de la memoria, no cuenta con pruebas automatizadas unitarias o de integración. Se decidió por tanto preparar una batería básica de pruebas funcionales de interfaz para comprobar que cambios en el programa principal no afecten negativamente a su contraparte web.

En cuanto al soporte de navegadores, se decidió que las pruebas se ejecutarían sobre los navegadores soportados más utilizados por los clientes de la empresa. En este caso Google Chrome y Microsoft Edge.

También se determinó que las pruebas deberían ejecutarse en modo *resposive*, es decir, en distintas resoluciones para simular su ejecución en distintos tamaños de pantalla. Las resoluciones elegidas fueron 1920x1080 como resolución de ordenador de escritorio, y 768x1024, para simular el uso desde una *tablet*.

3 Estado del arte. Herramientas para pruebas funcionales web

Actualmente, existe una gran variedad de herramientas en el mercado para poder automatizar pruebas de páginas web. Desde aquellas que se basan en la programación manual de las acciones de la prueba, como Selenium[5]; hasta otras de más alto nivel que permiten la grabación de acciones en un navegador, como Katalon Studio[6].

3.1 Requisitos de la herramienta

La búsqueda de la herramienta comenzó con la definición de los requisitos de automatización. Recordemos que el objetivo principal era encontrar una herramienta que permitiera cubrir el proceso completo de las pruebas: La automatización, lanzamiento y posterior revisión de sus resultados.

Además, se contaba con los siguientes requisitos:

- **Ejecución *cross-browser* de las pruebas**: soporte para la ejecución en los principales navegadores como Google Chrome, Internet Explorer y Microsoft Edge.
- **Desarrollo agnóstico de las pruebas**: el código de la prueba debe ser independiente del navegador en que se ejecutará. Así, la misma prueba podrá utilizarse en cualquiera de ellos.
- **Soporte a pruebas *responsive***: Soporte para la ejecución de las pruebas en distintas resoluciones, para comprobar si la aplicación se adapta correctamente a distintos tamaños de pantalla.
- **Reutilización de código entre las pruebas**: ya sea reutilizando acciones de las pruebas, utilizando patrones, como *Page Object* o mediante otras técnicas que provea la herramienta.
- **Restauración de *scripts* SQL**: para la preparación de datos del entorno de ejecución de una prueba, deberán poder restaurarse las bases de datos a de la aplicación a un estado concreto.
- **Generación de *logs* que reflejen la evolución de la ejecución**: así se facilitará su posterior revisión. Posiblemente, que deberán incluir capturas de pantalla en caso de error.

[5] Selenium – Página oficial: https://docs.seleniumhq.org/
[6] Katalon Studio – Página oficial: https://www.katalon.com

Aunque no tan prioritarios, se describieron otros requisitos adicionales, que facilitarían la integración de las pruebas funcionales web a los procesos ya existentes:

- **Estructura de los proyectos similar a los de las pruebas de escritorio**. De esta forma, a los encargados de automatización les será útil la experiencia con uno para poder aprender a automatizar pruebas del otro.
- **Integración con la infraestructura existente de la empresa:** posibilidad de integrar la herramienta con la infraestructura de la gestión de pruebas automatizadas ya existente.

3.2 Comparativa entre las herramientas

A continuación, comentaremos algunas de las herramientas más populares que investigamos a la hora de buscar la tecnología para el proyecto: Puppeter[7], Katalon Studio y Selenium. Para ello, haremos una comparativa entre ellas que, aparte de tener en cuenta los requisitos descritos anteriormente, está basada en la automatización de una prueba de ejemplo.

Esta prueba se ha diseñado pensando en las acciones básicas que nos resultarían necesarias para la automatización de pruebas. Entre ellas, encontrar un elemento o rellenar campos de un formulario. Se comparará el código final obtenido de la automatización.

La prueba de ejemplo consiste en los siguientes pasos:

1. Acceder a la dirección https://www.google.com/.
2. Localizar la barra de búsqueda e introducir el texto 'Hello World'.
3. Localizar el botón 'Buscar con Google' y hacer clic.
4. Localizar la entrada de Wikipedia, que tiene como título: 'Hola mundo - Wikipedia, la enciclopedia libre'.
5. Sacar una captura de pantalla de la entrada y guardarla como 'Screenshot.jpg'.

Como podemos observar, la prueba está compuesta por lo que se consideran acciones básicas para un *framework* de automatización de pruebas:

1. Navegación entre páginas web: Paso 1, paso 5.
2. Localizar elementos en el documento: Pasos 2 y 3.
3. Interactuar con elementos de la página web (Clic, introducir texto, etc).
4. Deberán incluirse las esperas necesarias durante la carga de las páginas.
5. Capturas de pantalla.

[7] Puppeteer -Página oficial: https://developers.google.com/web/tools/puppeteer/

3.2.1 Entorno basado en JavaScript: Puppeteer, Mocha y Chai

La primera de las opciones que se investigó fue la preparación de un entorno basado en JavaScript, compuesto por las herramientas Puppeteer, Mocha[8] y Chai[9]. Este entorno se preparó siguiendo un tutorial [15].

Pupeteer es una biblioteca de Google basada en NodeJS[10]. Provee una API para controlar instancias del navegador Google Chrome, tanto normales como *headless*. Las instancias *headless* son aquellas que no dibujan la interfaz del navegador, mejorando el rendimiento y reduciendo así el tiempo de ejecución de la prueba [16]. Se trata, por tanto, de una biblioteca ideal para utilizar en el contexto de un entorno de integración continua, donde se ejecuten automáticamente las pruebas después de cada integración.

Una clara desventaja de esta biblioteca es que solamente es compatible con un navegador, por lo que perdemos la posibilidad de realizar pruebas *crossbrowser*. Aunque este era uno de los requisitos principales, se decidió seguir investigando esta herramienta.

No es una biblioteca de *testing* de por sí, ya que únicamente ofrece operaciones para controlar el navegador. Será necesario usarla en conjunto con otras que sí estén enfocadas en las pruebas, como pueden ser Mocha o Jest. Para esta comparativa, hemos decidido utilizar Mocha, que gestionará la ejecución de las pruebas y sus resultados.

Para hacer las comprobaciones a lo largo de la prueba, utilizamos Chai, una biblioteca de aserciones. Nos permitirá escribir fácilmente comprobaciones complejas con una sintaxis clara y concisa, que se evaluarán durante la ejecución

A continuación, se ofrece el código de la automatización de la prueba de ejemplo, en el lenguaje JavaScript:

```
const puppeteer = require("puppeteer");
const should = require("chai").should();

let launchOptions = {
  headless: false
};

describe("Browser test", async () => {
  let browser, page;
  // Antes de ejecutar la prueba: creamos la instancia del navegador.
  before("Create an instance of the browser.", async () => {
    browser = await puppeteer.launch(launchOptions);
  });
```

[8] Mocha – Página oficial: https://mochajs.org/
[9] Chai – Página oficial: http://www.chaijs.com/
[10] NodeJS – Página oficial: https://nodejs.org/

```
  // Al terminar de ejecutar la prueba: se cierra la instancia del navegador.
  after("Close down the instance of the browser.", async () => {
    browser.close();
  });

  //Creamos una nueva pestaña y accedemos a 'https://www.google.com'
  it("Navigate to 'https://www.google.com/'", async () => {
    page = await browser.newPage();
    await page.goto("https://www.google.com/");
  }).timeout(0);

  // Localizar la barra de búsqueda e introducir el texto ‘Hello World’.
  it("Write text 'Hello World' in the search field.", async () => {
    await page.click("#lst-ib");
    await page.type("#lst-ib", "Hello World");
  });

  // Localizar el botón ‘Buscar con Google’ y hacer clic.
  // Esperamos que termine la carga de los resultados.
  it("Click the button ‘Buscar con Google’.", async () => {
    await page.click("[name='btnK'");
    await page.waitForNavigation();
  });

  // Comprobamos que existe la entrada de Wikipedia y hacemos
  // clic en el primer resultado, que debería ser la entrada de Wikipedia para
'Hola mundo'
  it("Check if the Wikipedia entry is present and navigate to it.", async ()
=> {
    const linkText = await GetElementText(page, ".r a");
    linkText.should.equal("Hola mundo - Wikipedia, la enciclopedia libre");
    await page.click(".r");
  }).timeout(0);

  // Hacemos la captura de pantalla.
  it("Capture a screenshot of the Wikipedia entry.", async () => {
    await page.screenshot({ path: "screenshot.png" });
  });
});

function GetElementText(page, locator) {
  return page.$eval(locator, element => {
    return element.innerHTML;
  });
}
```

Como podemos apreciar en el código, la prueba se enmarca en una función *describe*, perteneciente a la biblioteca Mocha. Cada uno de los pasos de la prueba se describe dentro de una función *it*, junto con una descripción de su funcionamiento. También cuenta con unos elementos *before* y *after*, para la creación de la instancia del navegador y su cierre, antes y después la ejecución de la prueba respectivamente.

La mayor dificultad encontrada durante la automatización de la prueba fue que la API de Puppeteer puede resultar confusa. Todas las acciones se realizan a través de la interfaz *Browser*, y a veces no resulta muy claro con que objeto estamos interactuando en cada momento.

El resultado de la ejecución fue el mostrado en la Figura 5. Podemos comprobar que cada paso de la prueba aparece en este *log*, con su título y un tic, indicando que se ha completado correctamente.

```
$ npm test
> puppeteer@1.0.0 test /mnt/bibliotecas/Bibliotecas/dev/puppeteer
> mocha --recursive test
 Browser test
   ✓ Navigate to 'https://www.google.com/' (1493ms)
   ✓ Write text 'Hello World' in the search field. (56ms)
   ✓ Click the button ‘Buscar con Google’. (1112ms)
   ✓ Check if the Wikipedia entry is present and navigate to it. (456ms)
   ✓ Capture a screenshot of the Wikipedia entry. (450ms)

 5 passing (4s)
```

Figura 5: Resultado de la ejecución de la prueba del entorno basado en JavaScript.

3.2.2 Katalon Studio

Katalon Studio es una *suite* para la automatización de pruebas de navegador. Se trata de un entorno de desarrollo integrado, basado en Eclipse, que incluye diversas herramientas para la gestión de proyectos de pruebas unitarias, ejecución de *suites* y ver resultados. Está basada en las bibliotecas Selenium y Appium [11], lo que le permite ofrecer soporte tanto para los principales navegadores de escritorio, como para dispositivos móviles basados en Android o iOS.

Una de sus principales características es grabación de pruebas. Así, se ejecutará una instancia de un navegador, que puede ser de escritorio o móvil, con el que podremos interactuar. Todas las acciones que realicemos se grabarán en una prueba, detectando automáticamente todos los controles con los que interactuemos, o los valores que introduzcamos.

[11] Appium – Página oficial: http://appium.io/

Gracias a esta característica, no se requieren conocimientos de programación para utilizar Katalon. Por tanto, los *testers* sin experiencia de programación podrán encargarse de la automatización. Los usuarios más avanzados podrán modificar el código autogenerado y adaptar la prueba para necesidades más específicas.

Respecto a la ejecución, podremos ejecutar las pruebas tanto individualmente como agrupadas en baterías. El IDE nos permitirá elegir los navegadores en los que queramos lanzarlas, y se encargará de crear las instancias y de su ejecución. Al finalizar, nos mostrará una lista de los resultados, y detallando, si las hubiera, todas las incidencias que se produjeron durante la misma.

Pasaremos ahora a comentar de la automatización de la prueba de ejemplo. En este caso, se realizó utilizando el modo de grabación descrito anteriormente. La herramienta generó una instancia del navegador Firefox y se realizaron manualmente los pasos de la prueba. Se generaron automáticamente las acciones siguientes, mostradas en la Figura 6.

Item	Object	Input
1 - Open Browser		""
2 - Navigate To Url		"https://www.google.com/"
3 - Set Text	input_q	"Hello World"
4 - Click	input_btnK	
5 - Click	a_Hola mundo - Wikipedia la en	
6 - Take Screenshot		"D:\Bibliotecas\Desktop\Screenshot.jpg"
7 - Close Browser		

Figura 6: Lista de acciones registradas por Katalon Studio con el grabador.

Internamente, estas acciones quedaron representadas por el código autogenerado que se muestra a continuación. Por defecto, se genera en el lenguaje Groovy, basado en Java.

```
'Crear instancia del navegador.'
WebUI.openBrowser('')

'1. Acceder a la dirección https://www.google.com/.'
WebUI.navigateToUrl('https://www.google.com/')

'2. Localizar la barra de búsqueda e introducir el texto 'Hello World'.'
WebUI.setText(findTestObject('Page_Google/input_q'), 'Hello World')

'3. Localizar el botón 'Buscar con Google' y hacer clic.'
WebUI.click(findTestObject('Object Repository/Page_Google/input_btnK'))

'4. Localizar la entrada de Wikipedia, que tiene como título: 'Hola mundo - Wikipedia, la enciclopedia libre'.'
WebUI.click(findTestObject('Page_Hello World - Buscar con Googl/a_Hola mundo - Wikipedia la en'))

'5. Sacar una captura de pantalla de la entrada y guardarla como 'Screenshot.jpg'.'
WebUI.takeScreenshot('D:\\Bibliotecas\\Desktop\\Screenshot.jpg')

'Cerrar instancia del navegador.'
WebUI.closeBrowser()
```

El resultado de la ejecución de la prueba fue el mostrado en la Figura 7.

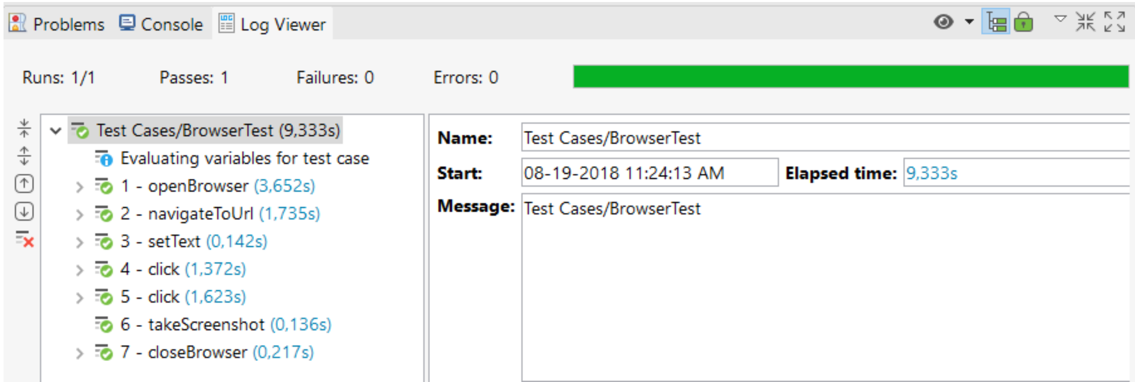

Figura 7: Resultado de la ejecución de la prueba de ejemplo en Katalon Studio.

Cabe destacar que, durante la automatización de la prueba, se produjeron problemas con el reconocimiento de controles. Una vez grabada, al ejecutar la prueba para comprobar su funcionamiento, no lograba encontrar el enlace 'Hola mundo' a Wikipedia. Esto fue debido a que incluía en la consulta XPath el enlace completo, que incluía metadatos de la sesión. Eliminando esta información de la consulta, la prueba se completaba correctamente. En la Figura 8 se muestra resaltada la cadena que provocaba el fallo.

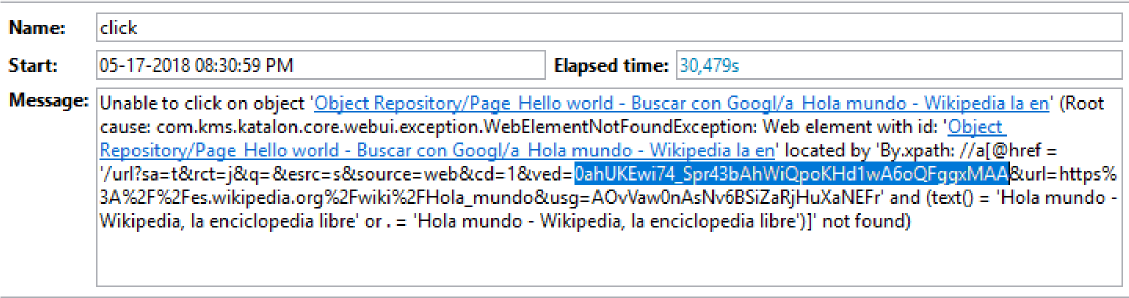

Figura 8: Ejemplo de metadatos inválidos que se utilizaban en la búsqueda del enlace, e impedían que se completara correctamente.

3.2.3 Proyecto *Ad hoc* basado en Selenium

Selenium es una *suite* de herramientas orientadas a la automatización del control de navegadores. Se trata actualmente de una de las más extendidas en el ámbito de pruebas para aplicaciones web [17]. Es también tomada como base para el desarrollo de herramientas de más alto nivel, como Katalon Studio o Oxygen [12].

Para este estudio nos centraremos en la nueva versión, denominada WebDriver. Se trata de una API que nace con el objetivo de permitir el control directo del navegador [18]. Ha sido estandarizado por el World Wide Web Consortium (W3C), y cualquier navegador puede implementar esta interfaz para permitir su control remotamente. [19] Gran variedad de ellos da soporte a Selenium WebDriver, entre los que se encuentran Chrome, Edge, Internet Explorer y Firefox.

Utiliza el concepto de *WebElement* para representar cualquier elemento del documento HTML. Toda interacción que queramos automatizar requerirá que primero localicemos el elemento (por su identificador, CSS, XPath u otro medio) y realizar la acción sobre él (clic, introducir texto, etc.).

La principal ventaja de desarrollar utilizando Selenium directamente, es la posibilidad de trabajar a más bajo nivel. De esta forma, podremos desarrollar una herramienta que se ajuste mejor a nuestras necesidades, sin tener la sobrecarga y la complejidad de un *framework* completo ya existente.

Dado que Selenium no gestionará la ejecución y los resultados de las pruebas, tendremos que recurrir a una biblioteca de pruebas. Para esta comparativa, utilizaremos MSTest[13], una biblioteca de pruebas integrada en Visual Studio[14].

Siguiendo con el ejemplo utilizado anteriormente, se ha automatizado la prueba genérica. En este caso, se ejecutará en el navegador Google Chrome, aunque el código se podría reutilizar para cualquiera de los navegadores soportados. Bastaría con cambiar el tipo de WebDriver usado.

```
namespace SeleniumHelloWorld
{
    [TestClass]
    public class SeleniumTest
    {
        [TestMethod]
        public void SeleniumHelloWorld()
        {
            // Creamos la instancia del Navegador.
            ChromeDriver chromeDriver = new ChromeDriver("./");

            // 1. Acceder a la dirección https://www.google.com/.
            chromeDriver.Navigate().GoToUrl(new Uri("https://www.google.com"));
```

[12] Oxygen – Página oficial: http://oxygenhq.org/
[13] MS Test – Página oficial: https://docs.microsoft.com/es-es/dotnet/core/testing/unit-testing-with-mstest
[14] Visual Studio – Página oficial: https://visualstudio.microsoft.com/

```
            // 2. Localizar la barra de búsqueda e introducir el texto
            //    'Hello World'.
            IWebElement searchBar = chromeDriver.FindElement(By.Id("lst-ib"));
            searchBar.SendKeys("Hello World");
            searchBar.SendKeys(Keys.Escape);

            // 3. Localizar el botón 'Buscar con Google' y hacer clic.
            IWebElement searchButton = chromeDriver.FindElement(By.Name("btnK"));
            searchButton.Click();

            // 4. Localizar la entrada de Wikipedia, que tiene como título:
            //    'Hola mundo - Wikipedia, la enciclopedia libre'.
            IWebElement wikipediaLink = chromeDriver.FindElement(
                   By.LinkText("Hola mundo - Wikipedia, la enciclopedia libre"));
            wikipediaLink.Click();

            // 5. Sacar una captura de pantalla de la entrada y guardarla como
            //    'Screenshot.jpg'.
            Screenshot screenShot =
                ((ITakesScreenshot)chromeDriver).GetScreenshot();
            screenShot.SaveAsFile("./Screenshot.jpg",
                                  ScreenshotImageFormat.Jpeg);

            // Cerrar la instancia del navegador.
            chromeDriver.Close();
        }
    }
}
```

Analizando el código, podemos ver que gracias a la interfaz *IWebElement*, se puede delimitar cada elemento con el que queramos interactuar. Por ejemplo, la barra de búsqueda o el botón 'Buscar con Google'. Una vez localizados, disponemos de distintas operaciones para interactuar con ellos, como introducir texto o hacer clic, respectivamente.

El resultado de ejecutar la prueba se muestra en la Figura 9. Como podemos apreciar, a diferencia de las otras herramientas, no se ofrece un desglose de las distintas acciones o comprobaciones que se hicieron durante la prueba.

```
Test Name:  SeleniumHelloWorld
Test FullName: SeleniumHelloWorld.SeleniumTest.SeleniumHelloWorld
Test Source:      D:\dev\VisualStudio\SeleniumHelloWorld\
                      SeleniumHelloWorld\UnitTest1.cs : line 12
Test Outcome:     Passed
Test Duration:    0:00:09,1908503
```

Figura 9: Resultado de la ejecución de la prueba de Selenium usando MSTest.

3.2.4 Conclusiones de la comparativa entre las herramientas

A continuación, se presenta el resultado de la comparativa entre las herramientas, resumido en forma de tabla:

	Puppeteer	**Katalon Studio**	**Selenium**
Soporte para todas las acciones básicas de la prueba de ejemplo.	Sí	Sí	Sí
Ejecución de pruebas en distintos navegadores.	No. Solo soporta Chrome.	Sí	Sí
Posibilidad de reutilización de código entre pruebas.	Sí	Sí	Sí
Ejecución de las pruebas en múltiples resoluciones para pruebas *responsive*.	Sí	Sí	Sí
Generación de logs sobre la evolución de la ejecución de la prueba.	Sí (Usando biblioteca externa).	Sí	Sí (Usando biblioteca externa.)
Posibilidad de restaurar Script SQL.	Usando biblioteca externa.	Usando biblioteca externa.	Usando biblioteca externa.
Posibilidad de agrupar los proyectos similarmente a los ya existentes para escritorios.	Sí	No	Sí
Posibilidad de integrar con la infraestructura existente de la empresa.	Sí	No	Sí

Tabla 1: Resumen de los resultados de la compartiva entre las distintas herramientas.

Como puede apreciarse en la tabla, la herramienta que mejor se adaptaba a los requisitos fue la basada en la biblioteca Selenium. Fue elegida, principalmente por su simplicidad, ya que no eran necesarias herramientas muy sofisticadas; y porque facilitaría la integración con la infraestructura existente para la gestión de los resultados de las pruebas.

Respecto a las otras, Puppeteer se desestimó al soportar únicamente el navegador Chrome, una gran desventaja frente a las otras. Aunque Katalon Studio cumplía con la mayoría de los requisitos detallados, no era compatible con la infraestructura ya existentes, y no se podría integrar fácilmente. Además, desarrollar las pruebas en Katalon impediría poder unificar el proceso de pruebas de escritorio y de navegador en un futuro.

En capítulos sucesivos, se detallará el proceso de desarrollo *ad hoc* de una herramienta basada en Selenium y el de una pequeña batería de pruebas haciendo uso de ella.

4 Propuesta para automatización de pruebas basada en Selenium

En este capítulo se presentará el diseño de la herramienta para la automatización y ejecución de las pruebas. Se describirá la estructura de la solución, detallando el propósito de cada uno de los módulos o proyectos que la componen. También se expondrá un ejemplo del proceso de ejecución de una batería de pruebas.

4.1 Tecnologías utilizadas

La herramienta se desarrolló usando el lenguaje C#, para ejecutarlo en la plataforma .NET Framework [15]. En concreto, se usará se creará como una solución de .NET, dividida en distintos proyectos o módulos, que se explicarán en el siguiente apartado.

La elección de la tecnología fue influida principalmente por las herramientas de automatización de pruebas de escritorio ya existentes, que están basadas en ella. Así, se facilitaría la transición de los usuarios de una a otra; además nos permitió reaprovechar utilidades ya desarrolladas, como la biblioteca encargada de la restauración de *scripts* SQL.

Se incluyeron en la solución las bibliotecas de Selenium para .NET, para poder hacer uso de esta tecnología en los proyectos. Se instaló la biblioteca *WebDriver* de cada navegador al que se dio soporte: Chrome, Edge, Internet Explorer y Firefox. Todas se instalaron a través del repositorio NuGet[16], de forma que fuera más fácil gestionar sus versiones.

4.2 Estructura de la solución .NET

La herramienta está compuesta por los siguientes proyectos o módulos, representados en la Figura 10:

- **Proyecto Base**: Proyecto que contiene utilidades básicas para todos los proyectos. Incluye la clase *Test* y *Page*. Todos los proyectos dependen de él.
- **Proyectos *Application*** Proyecto que contiene todos los mapeos de la aplicación. Este proyecto es el que hará de interfaz entre la prueba y la aplicación en sí. Se creará un proyecto *Application* distinto para cada uno de los programas que deseemos probar.
- **Proyectos *ApplicationTests***: Contiene las pruebas de una aplicación determinada. Va siempre asociado a un proyecto *Application*. Cada aplicación que queramos probar deberá contar con su propio proyecto *ApplicationTests*.
- **Proyecto Launcher**: Es el proyecto encargado de gestionar la ejecución de las pruebas. Recibe los parámetros de la ejecución de la suite, como la lista de pruebas a ejecutar, los navegadores y las resoluciones destino.

[15] .NET Framework – Página oficial: https://docs.microsoft.com/es-es/dotnet/framework/
[16] NuGet - Página oficial: https://www.nuget.org/

En caso de que se quieran automatizar pruebas para otras aplicaciones web distintas, bastaría con añadir un proyecto del tipo *Application* y el correspondiente *ApplicationTests*. También será necesario añadir en *Launcher* la referencia al proyecto de *ApplicationTests*.

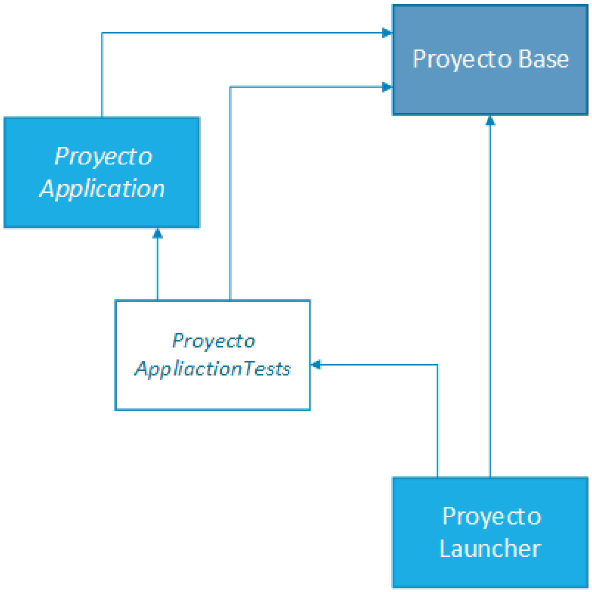

Figura 10: Esquema de la estructura interna los proyectos que componen la herramienta. Las flechas indican dependencia entre proyectos.

Cabe destacar que se apostó por un diseño modular, ya que se debía facilitar en lo posible una futura integración con la infraestructura establecida actualmente en la empresa. Dicha infraestructura gestiona toda la información relacionada al lanzamiento de las pruebas automatizadas de escritorio, y las integra dentro de otras herramientas. Debido a limitaciones en el tiempo disponible para llevar a cabo este proyecto, se pospuso esta integración para más adelante

A continuación, se explicará detalladamente la estructura interna y las particularidades de cada uno de los proyectos.

4.2.1 El proyecto *Base*

El proyecto Base contiene las clases comunes de la solución, que son requeridas por varios de los proyectos. Aunque algunas de ellas tengan poca relación entre sí, se decidió agruparlas en un mismo proyecto para evitar las dependencias cíclicas entre paquetes.

A. La clase Test

Contiene una de las clases centrales de la herramienta, *Test*, la clase genérica usada como base para implementar una prueba. Ofrece únicamente un método público, *Execute*, que gestiona la ejecución de la prueba. La clase se diseñó de manera que el código de una prueba fuera agnóstico al navegador en el que se ejecuta. Serán las clases del proyecto *Application* las que gestionen, si fuera necesario, las acciones relativas a un navegador concreto.

Por tanto, el lanzador solo tendrá que instanciar la prueba una vez, y pasarle sus parámetros para cada ejecución: la instancia del navegador ya configurada, el *log* donde registrar su evolución y la conexión a base de datos de la aplicación. Posteriormente se explicará en más detalle la ejecución de las pruebas.

La ejecución de una prueba se divide en tres etapas distintas, representadas por los métodos *SetUp*, *TestSteps* y *Teardown*. En primer lugar, se encuentra la fase de *SetUp,* donde se prepara el entorno para ejecutar la prueba: restaurar *scripts* SQL para preparar los datos del programa; instanciar todos los de todos los controles y formularios con los que se interactuará la prueba, etc.

TestSteps es el método donde se ubicarían los pasos de prueba: el inicio de sesión, navegación por los formularios, etc. También se ubicarán aquí todas las comprobaciones relacionadas con la prueba, como los valores que se muestran por pantalla, los elementos visibles o invisibles, entre otras.

Finalmente, en el *TearDown*, se ubican las acciones para restaurar el estado del sistema en caso de que fuera necesario. Por ejemplo, si durante la preparación del entorno fuera necesario configurar mal algún elemento del sistema para probar su comportamiento, durante el *TearDown* se restauraría el elemento a su estado original. Esta acción se ejecutará incondicionalmente, para asegurar que las restauraciones tienen lugar.

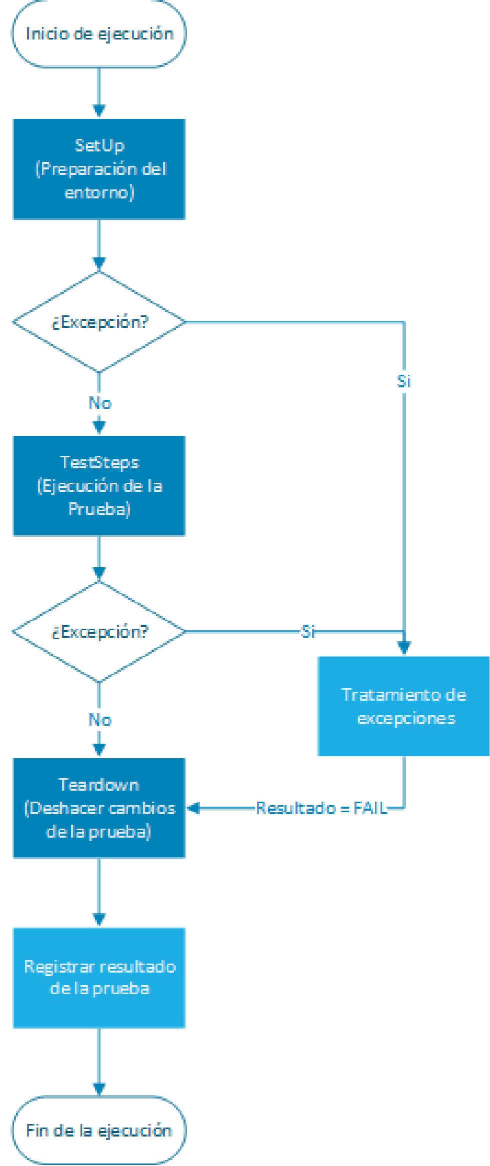

Figura 11: Flujo de ejecución de una prueba.

B. *La clase Page*

La clase *Page* sirve como base para la representación de cualquier formulario o componente de la aplicicación web. Se implementan siguiendo el patrón *Page Object*, que consiste en ofrecer a las pruebas una interfaz para interactuar con la página, ocultando la interacción con los controles. Internamente se gestionan todas las operaciones directas con el *WebDriver*.

De esta forma, podemos reutilizar gran parte del código entre las pruebas, evitando automatizar las mismas acciones para cada una de ellas. Además, al centralizar el código del formulario, en caso de que hubiera un cambio que provoque fallos en las pruebas, es más sencillo repararlas, frente a tener que modifcarlas una a una.

Más adelante, en el capítulo *5.3 "¿Pruebas "desechables" o pruebas* "estructuradas", se explicará con más detalle las ventajas de utilizar este patrón.

4.2.2 El proyecto *Application*

El proyecto *Application* contiene la información referente a la aplicación para la que deseamos automatizar pruebas. Es decir, es el proyecto que hace de interfaz entre la prueba y la aplicación web. Contiene los formularios y las opciones de configuración que nos puedan ser necesarias, como referencias a las bases de datos.

La estructura del proyecto refleja la navegación entre los formularios de la aplicación, de forma similar a un árbol. Se eligió para que a la hora de automatizar las pruebas sea más fácil encontrar los a formularios utilizar. También nos ayudará a comprobar si ya han sido o no automatizados.

En la Figura 12 tenemos un ejemplo que representa como se puede navegar desde el *Home* de la aplicación a los apartados 'Actividades', 'Controles de enfermería' o 'Seguimientos'. Dentro de 'Controles de enfermería', podemos acceder a 'Controles de diuresis'.

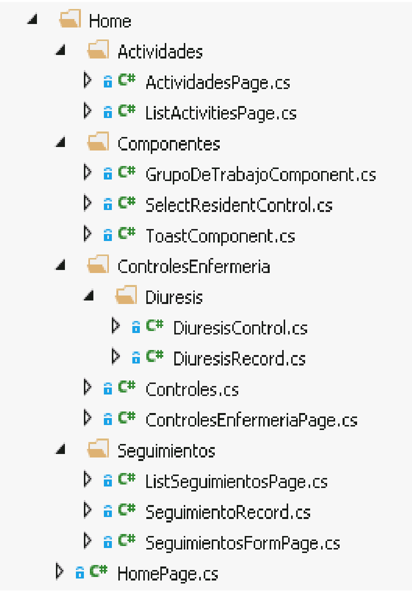

Figura 12: Ejemplo de la estructura en árbol del proyecto Application.

Como ya se ha comentado antes, las interfaces de los formularios están implementadas siguiendo el patrón *Page Object* [20]. Una política que se ha seguido a la hora de implementarlos fue que no deben contener ninguna verificación, salvo que sea en cuanto a la carga inicial. Esto se debe a que todas las verificaciones y comprobaciones deben encontrarse en el código de las pruebas [21].

A. Registros de Datos: Obtención de datos de un formulario.

A la hora de automatizar las pruebas, tendremos que preparar comprobaciones sobre la información que contenga una página en determinado momento de la ejecución. Para obtenerla, deberemos crear operaciones en las clases *Page* que nos permitan extraerla a partir de sus controles: ya sea el contenido de un simple campo de texto, una fecha, el estado de un *checkbox*, etc.

Un ejemplo sería si quisiéramos obtener el valor del nombre de usuario que ha introducido un seguimiento de enfermería, mostrado en la Figura 13. Para hacerlo, dentro de la clase del formulario habremos de añadir el código para:

- Localizar el elemento
- Obtener su valor
- Procesarlo si fuera necesario.

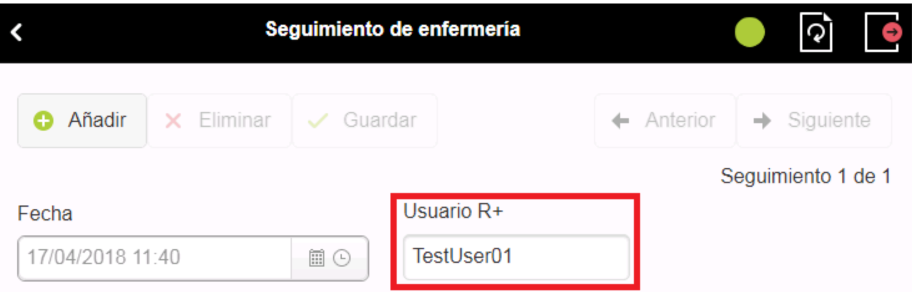

Figura 13: Ejemplo de formulario del que querríamos obtener datos. Resaltado en rojo sale el campo de ejemplo.

El código para obtener el valor de este elemento sería el siguiente:

```
public string GetResiUserValue()
{
    IWebElement userField = getSeguimientosForm().FindElement(
       By.XPath(".//form[contains(@class,'thereAreProgressNotes')]
                   //input[contains(@data-bind, 'value: User')]"));
       return userField.GetAttribute("value");
}
```

En ocasiones, obtener los datos valor a valor puede resultar muy engorroso. Por ejemplo, cuando debemos obtener muchos datos relacionados, como los registros de una tabla. Para obtener esta información estructuradamente, hemos decidido agruparla en registros (*record)* inmutables. Así, se simplifica la introducción y la obtención de datos.

Para demostrarlo, tomaremos como base el formulario de controles de enfermería. Concretamente, el de controles de diuresis. Como podemos apreciar en la Figura 14, es una tabla paginada con una serie de registros. Si quisiéramos obtener los valores de uno de sus registros, se podría programar para acceder manualmente al valor de cada una de las columnas de la fila; o crear un *record* para los seguimientos de diuresis.

Figura 14: Tabla de controles de diuresis, de la que deseamos obtener una fila.

La clase *DiuresisRecord* representa el contenido de una fila de la tabla:

```
public class DiuresisRecord
{

      public DateTime date { get; }
      public double cantidad { get; }
      public int numPanales { get; }
      public string observaciones { get; }
      public string resiUser { get; }

      public DiuresisRecord(DateTime date, double cantidad, int numPanales,
                             string observaciones, string resiUser = null) {//…}
}
```

Podremos entonces utilizarla para obtener datos de una fila concreta de la tabla, a partir de su índice.

```
    public DiuresisRecord getDiuresisRecordAt(int index)
    {
          List<string> elements = getTextOfElementsOfTableRowByIndex(index);

          DiuresisRecord record = new DiuresisRecord(
                DateUtils.parseFullDate(elements.ElementAt(0)),
                Double.Parse(elements.ElementAt(1)),
                Int32.Parse(elements.ElementAt(2)),
                elements.ElementAt(3),
                elements.ElementAt(4)
          );

          return record;
    }
```

También podemos utilizarla para rellenar los datos de un formulario.

```
public void AddNewDiuresisRecord(DiuresisRecord diuresis)
{
       AddNewRecord();

       WriteDateTime(diuresis.date);
       WriteCantidad(diuresis.cantidad);
       WritePanales(diuresis.numPanales);
       WriteObservaciones(diuresis.observaciones);

       ConfirmAddNewRegistry();
}
```

Habrá ocasiones en las que esta aproximación pueda resultar ineficiente, como aquellos formularios en los que aparezcan muchos campos, o tenga campos con poca relación entre sí. En estos casos, será necesario determinar si es posible desgranar de alguna forma en la información en registros más pequeños. En caso de no ser posible, se recurrirá a la obtención o inserción valor a valor.

4.2.3 El proyecto *ApplicationTests*

El proyecto *ApplicationTests*, siendo *Application* el nombre de la aplicación a probar, es un tipo de proyecto donde se encuentran todas las pruebas de una aplicación. Al igual que pasaba con el proyecto *Application* cada aplicación para la que queramos desarrollar pruebas deberá tener su propio proyecto.

La estructura interna del proyecto agrupa las pruebas en dos niveles. En primer lugar, los agrupa en función de la incidencia en la que se automatizaron. Esto es a causa de que en la infraestructura existente se crea una incidencia de automatización, a la que se le asocian todas las pruebas de aceptación automatizadas en la misma. Se marcan así por trazabilidad y para facilitar la consulta de información relacionada, como la definición de la prueba, los diseños de los casos de prueba, etc.

El segundo nivel es la agrupación a nivel de prueba de aceptación, donde para cada una de ellas crearemos una carpeta. Dentro de esta carpeta se encontrarán todas las pruebas automatizadas asociadas a ella. Por ejemplo, para una prueba con la condición de tener o no activa una opción de configuración, podríamos definir dos casos de prueba, uno donde estuviera activa, y otro donde no.

Los ficheros relacionados con las pruebas ya sean *scripts* SQL, ficheros de resultados esperados, entre otros; deberán ubicarse en el nivel más general posible. Es decir, si un fichero es único para una única instancia de una prueba, deberá ubicarse dentro de la carpeta de la propia prueba, por ejemplo, ID25456 > PA023714 > PS006261 > Fichero_VP.txt

En caso de que un fichero pueda compartirse entre más de una prueba, podrá utilizarse a nivel más general, por lo que podríamos guardarlo a nivel de la carpeta de la PA. Por ejemplo, la ruta podría ser ID25456 > PA023714 > Script.sql. Y en caso de que se estén automatizando pruebas muy relacionadas en una misma incidencia, que requieran de los mismos ficheros, podrá guardarse a nivel de incidencia.

En la Figura 15 presentamos un ejemplo de la organización explicada anteriormente.

```
▷ ID29426
◢ ID29563
    ◢ PA023714
        PA023714_DatosResidencia_01.sql
        PA023714_DatosUsuarios.sql
      ▷ PS006261.cs
    ◢ PA023715
        PA023715_DatosResidencia_01.sql
        PA023715_DatosUsuarios.sql
      ▷ PS006262.cs
    ▷ ID29563_Base.cs
▷ ID29768
```

Figura 15: Ejemplo de la estructura de un proyecto ApplicationTests.

A. Reutilización de Código

Durante la automatización, encontraremos en ocasiones algunas pruebas que realizan operaciones muy similares, e incluso cuenten con algunas fases de la ejecución idénticas. Se buscará siempre que sea posible generalizar el comportamiento creando pruebas *base*. Estas clases base implementan la funcionalidad común a todas las pruebas que hereden de ellas, evitando así repetirla.

Un ejemplo de esto sería el caso de la *PA0235696*, donde se nos pide probar la aparición de un diálogo durante el inicio de sesión. La prueba nos pide verificar que el funcionamiento es el mismo tanto el caso en que un módulo esté activo, como el caso en el que este inactivo.

Así, los pasos de la prueba y la verificación serán los mismos. La única diferencia será restaurar unos u otros *scripts SQL* durante la fase de *SetUp*. Podemos entonces, crear una clase base, *PA23569_Base*. Dicha clase base implementará todos los métodos de una prueba normal, y dejará para sobrescribir la implementación concreta para cada prueba. En el caso de la imagen, es el método *restoreDatabaseScripts*, que deberán implementar sus herederas.

```
public abstract class PA023596_Base : Test
{
        protected LogInPage login;
        protected GrupoDeTrabajoComponent workGroupSelection;

        protected override void SetUp()
        {
            this.login = new LogInPage(webDriver);
            this.workGroupSelection =
                         new GrupoDeTrabajoComponent(webDriver);

            restoreDatabaseScripts();
        }
```

```
        protected override void TestSteps()
        {
            login.navegarAInicio();
            info("Iniciando sesión como TestUser02");
            login.iniciarSesion("TestUser02", "Selenium");

            // Resto de la prueba
        }

        protected override void Teardown()
        {

             restoreScript($@"{Workspace.WORKSPACE_DIRECTORY}\ID29426\PA023
             596\PA023596_DatosUsuarios_Restaurar.sql", "DatosCodigo");
        }

        protected abstract void restoreDatabaseScripts();
    }
}
```

El código de la PS006258, que prueba el caso en el que el módulo está activo, se reduciría a este fragmento de código, gracias al haber eliminado la duplicación:

```
public class PS006258 : PA023596_Base
{
       protected override void restoreDatabaseScripts()
       {
              restoreScript(
                     $@"{Workspace.WORKSPACE_DIRECTORY}\ID29426\PA023596\PS006258\
                     PS006258_DatosResidencia_01.sql",
                     "DatosResidencia_001_01"
              );
              restoreScript(
                     $@"{Workspace.WORKSPACE_DIRECTORY}\ID29426\PA023596\PS006258\
                     PS006258_DatosUsuarios.sql",
                     "DatosUsuarios"
              );
       }
}
```

4.2.4 El proyecto *Launcher*

El proyecto *Launcher* es el proyecto encargado del lanzamiento y ejecución de las baterías o *suites* de pruebas. Se trata del proyecto de más alto nivel de esta solución.

Su clase principal es la homónima *Launcher* que, a partir de los parámetros de ejecución de la *suite,* se encargará de: buscar las pruebas, preparar todas las dependencias de las mismas, generar logs, y del tratamiento de excepciones. Se incluye a continuación una descripción detallada del flujo de ejecución de una batería:

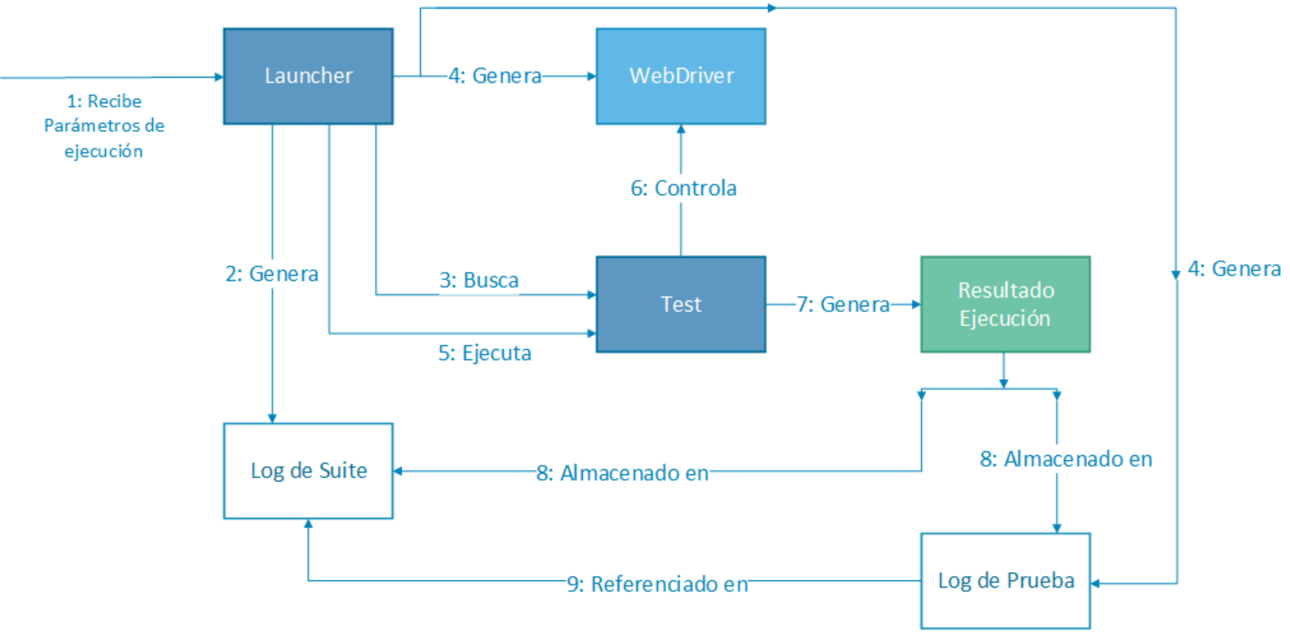

Figura 16: Diagrama que representa el orden de sucesos en la ejecución de una batería de pruebas.

La ejecución de una batería o *suite* de pruebas sigue el proceso representado en la Figura 17. Comienza (1) cuando la clase *Launcher* es invocada, con los parámetros de ejecución: los códigos de las pruebas a ejecutar, los navegadores, y las resoluciones. *Launcher* también genera un *log de Suite* (2), que agrupará los resultados y las estadísticas de la batería.

En el paso (3), *Launcher* buscará la prueba a partir de su código identificador en los proyectos de *ApplicationTest*. En caso de no encontrarlo, se registrará como fallida la ejecución en el *log de Suite*. A continuación, se inicia el bucle de ejecución de una prueba: para una se ejecutará la combinatoria completa de navegadores y resoluciones.

Este bucle inicia con la generación del *WebDriver* (4), que se configurará con la resolución correspondiente, y el *log de Prueba*. Ambos son específicos para esa ejecución determinada. Son pasados como parámetro a *Test* y se inicia la ejecución de la prueba (5). Durante la ejecución, *Test* gestionará internamente su ejecución, controlando el *WebDriver* (6) a través de los *Page*. Su resultado evolucionará a medida que avance, y se registrará dentro del *log de Prueba*.

Una vez finaliza esta ejecución concreta, se registra el resultado final de la prueba, que se almacenará tanto en el *log de Prueba* como en el *log de Suite* (8). El ciclo se reiniciará en el paso 4 hasta que se hayan acabado todas las posibles combinaciones entre una prueba, los navegadores y las resoluciones. Hecho esto, se buscará la siguiente prueba (3) y se repetirá el proceso, hasta que se hayan acabado también las pruebas pendientes.

B. Ejemplo de ejecución de una batería de pruebas

Un ejemplo de *suite* sería ejecutar las pruebas 'PS006257' y 'PS006258', en los navegadores Chrome y Edge, para las resoluciones *Desktop* (1920x1080) y *Tablet* (768x1024). En total, se realizarían 8 ejecuciones distintas. Con estos parámetros se inicia la ejecución.

```
private static void Main()
{
  Launcher launcher = new Launcher();

  List<string> tests = new List<string>();
  tests.Add("PS006257");
  tests.Add("PS006258");

  List<Browser> browsers = new List<Browser>();
  browsers.Add(Browser.Chrome);
  browsers.Add(Browser.InternetExplorer);

  List<Resolution> resolutions = new List<Resolution>();
  resolutions.Add(Resolution.Desktop);
  resolutions.Add(Resolution.Tablet);

  launcher.ejecutarSuite(tests, browsers, resolutions);
}
```

Figura 17: Ejemplo de código para programar el lanzamiento de dos pruebas, en Chrome y Edge en las resoluciones Desktop y Tablet.

Una vez compilada la solución, ejecutaremos la batería. Veremos que se abrirán y cerrarán las distintas instancias de los navegadores y se ejecutarán las pruebas, sin necesidad de intervención del usuario.

Cuando finalice, se habrá generado una carpeta llamada “Execution_Logs” que contiene tanto los *logs* de las pruebas, separados en carpetas, como el *log* de la batería, almacenado en la carpeta “resultados”. La Figura 19 muestra el contenido del *log* resultante de la ejecución de la batería de ejemplo.

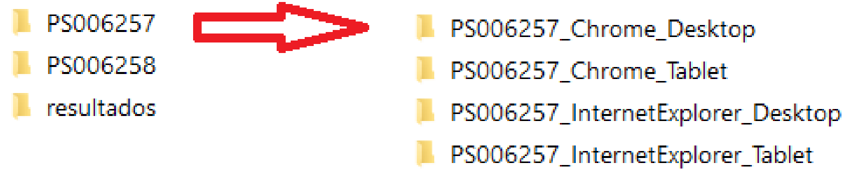

Figura 18: Contenido de la carpeta de logs de la batería de ejemplo.

```
FECHA EJECUCIÓN      CÓDIGO PS   NAVEGADOR         RESOLUCIÓN   RESULTADO   LOG
15/07/2018 21:30:27  PS006257    Chrome            Desktop      PASS        C:\Users\adriano.vega\Desktop\15072018\R
15/07/2018 21:31:32  PS006257    Chrome            Tablet       PASS        C:\Users\adriano.vega\Desktop\15072018\R
15/07/2018 21:32:18  PS006257    InternetExplorer  Desktop      PASS        C:\Users\adriano.vega\Desktop\15072018\R
15/07/2018 21:33:03  PS006257    InternetExplorer  Tablet       PASS        C:\Users\adriano.vega\Desktop\15072018\R
15/07/2018 21:33:44  PS006258    Chrome            Desktop      PASS        C:\Users\adriano.vega\Desktop\15072018\R
15/07/2018 21:34:23  PS006258    Chrome            Tablet       PASS        C:\Users\adriano.vega\Desktop\15072018\R
15/07/2018 21:35:12  PS006258    InternetExplorer  Desktop      PASS        C:\Users\adriano.vega\Desktop\15072018\R
15/07/2018 21:35:56  PS006258    InternetExplorer  Tablet       PASS        C:\Users\adriano.vega\Desktop\15072018\R

ESTADÍSTICAS
Fecha inicio: 15/07/2018 21:29:53 , Fecha Fin: 15/07/2018 21:36:00
Tiempo total ejecución: 0 horas 6 minutos 7 segundos
Total pruebas ejecutadas: 8
Total pruebas PASS: 8
Total pruebas WARN: 0
Total pruebas FAIL: 0
Total pruebas desconocido: 0
```

Figura 19: Log resultante de la ejecución de la batería de pruebas de ejemplo.

5 Automatización de pruebas funcionales

En este capítulo se describirá el proceso de automatización de una prueba utilizando nuestra propuesta basada en *Selenium*. Durante este proceso se define, programa y posteriormente se revisa el funcionamiento de la prueba, hasta finalmente integrarla en las baterías.

El proceso comienza con la definición de la prueba en sí. Como se ha explicado en capítulos anteriores, las pruebas funcionales derivan de las pruebas de aceptación (PA) definidas durante el análisis del cambio. Concretamente, son casos de prueba de la PA con valores dados.

Como resultaría muy costoso automatizar todas las PA del producto, se deben seleccionar aquellas que se desee automatizar. Esta tarea normalmente recae sobre los *testers*, que indicarán aquellas pruebas de funcionalidades que consideren críticas. También se pueden elegir pruebas de las que se hayan detectado fallos, de forma que una vez corregido, podamos comprobar que no se repite.

La prueba elegida como ejemplo es una simple comprobación en los formularios de controles de enfermería. Describe como se debe tratar el caso de la introducción de un control con una fecha en el futuro.

CONDICIÓN

- Tener activo el bloqueo (casilla marcada) 'Impedir la introducción de controles de enfermería de fecha futura' (Configuración / Datos Generales / SocioSanitario / Configuración / Registros de enfermería).

PASOS

- Intentar introducir un control de enfermería con una fecha y hora superior a la del sistema.[1]

RESULTADO ESPERADO

- No es posible, no se inserta ningún nuevo registro. Aparece un mensaje con el texto “No es posible registrar controles a futuro”.

OBSERVACIONES

1. No podremos seleccionar en el calendario una fecha superior a la actual, tampoco podremos seleccionar en el desplegable de hora, una superior a la del sistema. No obstante, podremos escribir una fecha y hora superior a la actual.

Figura 20: Prueba de aceptación que se tomará de ejemplo para la automatización.

5.1 Diseño de la prueba

Una vez se ha elegido la prueba de aceptación, se procede a diseñar el caso de prueba. Cada prueba está compuesta por tres apartados: acciones previas, prueba y resultado esperado, que se detallarán a continuación siguiendo ese orden.

5.1.1 Acciones previas

Para poder realizar la prueba, habrá ocasiones en las que dependeremos de que una u más opciones de configuración estén activadas, existan determinados registros en la base de datos, etc. Esto es el estado previo del sistema o acciones previas en nuestro diseño. Inicialmente, la aplicación puede no estar correctamente configurada o la ejecución de otras pruebas haya hecho cambios que afecten a la ejecución de la actual. Será necesario restaurar el estado cada vez que ejecutemos la prueba.

Hay varias alternativas para la preparación del sistema. Una opción podría ser el incluir la configuración del sistema dentro de la ejecución de la prueba. Es decir, automatizar el acceso a las opciones de configuración y manualmente modificar aquellas que necesitemos. También registrar los datos que podamos necesitar inicialmente para la ejecución, entre otros. Esto resultar muy costoso y aumentar excesivamente el tiempo de ejecución de las baterías de pruebas.

Por otro lado, se puede optar por usar *scripts* SQL: antes de automatizar la prueba, se prepara el entorno manualmente y se vuelca el estado de las bases de datos del programa en *scripts*. Así, antes de cada ejecución se restauran las bases de datos, restaurando así el estado del programa. En comparación con la otra opción, reduce considerablemente el tiempo de ejecución.

Por ejemplo, en la definición de la prueba de aceptación de la Figura 20, podemos ver que se hace referencia a tener activa una opción llamada *Impedir la introducción de controles de enfermería de fecha futura*, que se encuentra en la aplicación principal. Para no tener que abrir el programa cada vez que vayamos a ejecutar esta prueba para activarla, la activaremos manualmente y generaremos un *script* SQL.

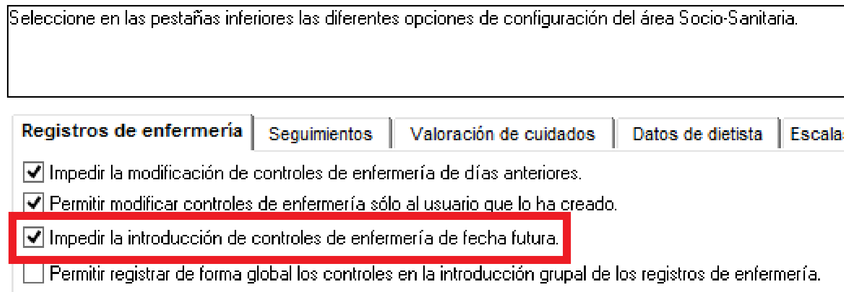

Figura 21: La opción 'Impedir la introducción de controles de enfermería de fecha futura', necesaria para la prueba, marcada.

Otros parámetros que deberemos tener en cuenta para esta prueba en concreto serán: el usuario que se utilizará debe tener los permisos necesarios y debe existir un residente sobre el que se generará el control. En este caso hemos creado un residente y un usuario con acceso a los controles de diuresis de este. Una vez preparado todo, podremos volcar los *scripts*. Se obtuvieron los scripts 'ID29426_DatosResidencia_01.sql' y 'ID29426_DatosUsuarios.sql'.

Finalmente, el contenido del apartado acciones previas de esta prueba contendría la siguiente información:

Restaurar los scripts de base de datos:

- ID29426_DatosResidencia_01.sql
- ID29426_DatosUsuarios.sql

Estado previo del sistema:

- Utilizar el usuario TestUser02, que tiene permisos para ver y crear controles de diuresis.
- Tener activo el bloqueo (casilla marcada) *Impedir la introducción de controles de enfermería de fecha futura*.
- Existe un residente llamado 'Nuevo Residente 00001'.

5.1.2 Pasos de la prueba

En el apartado *pasos de la prueba* se definen las acciones que componen el caso de prueba y las comprobaciones que deberán llevarse a cabo. Antes de comenzar a definir la prueba, es recomendable realizarla manualmente para familiarizarnos con la funcionalidad a probar. Esto también nos ayudará a completar el apartado de acciones previas.

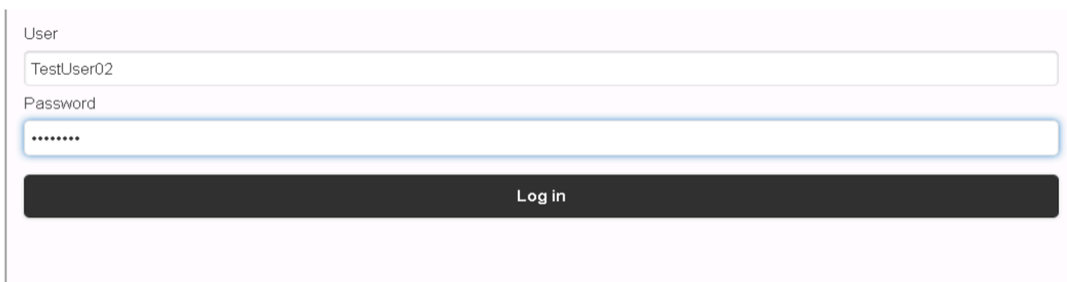

Figura 22: Pantalla de inicio de sesión de la aplicación. Iniciaremos sesión con el usuario TestUser02.

Las primeras acciones que se describen en este apartado suelen formar parte de la navegación al formulario en el que se llevará a cabo la prueba. Comenzaremos por el acceso a la web de la aplicación, y el inicio de sesión. En el apartado anterior configuramos al usuario *TestUser02* con los permisos requeridos por la prueba. Por tanto, iniciamos sesión usándolo, como aparece en la Figura 22.

Una vez iniciada la sesión, accederemos al *home* de la aplicación (Figura 23). Este es el centro de navegación a las funcionalidades que ofrece, como el apartado de mensajería, donde podremos acceder a los mensajes del usuario; el apartado de tratamientos, para ver los tratamientos pautados a los residentes; entre otras. En este caso nos interesa la opción de Controles, que agrupa los controles de enfermería.

Para poder acceder a los controles, primero tendremos que seleccionar un residente existente. En la Figura 23, se puede apreciar en la zona inferior una barra con el título 'Seleccione un residente'. Es en realidad un menú desplegable. Aquí deberemos seleccionar a 'Nuevo Residente 00001', que creamos durante los pasos previos. Veremos que el menú se ha replegado y ahora muestra el nombre del residente seleccionado (Figura 24).

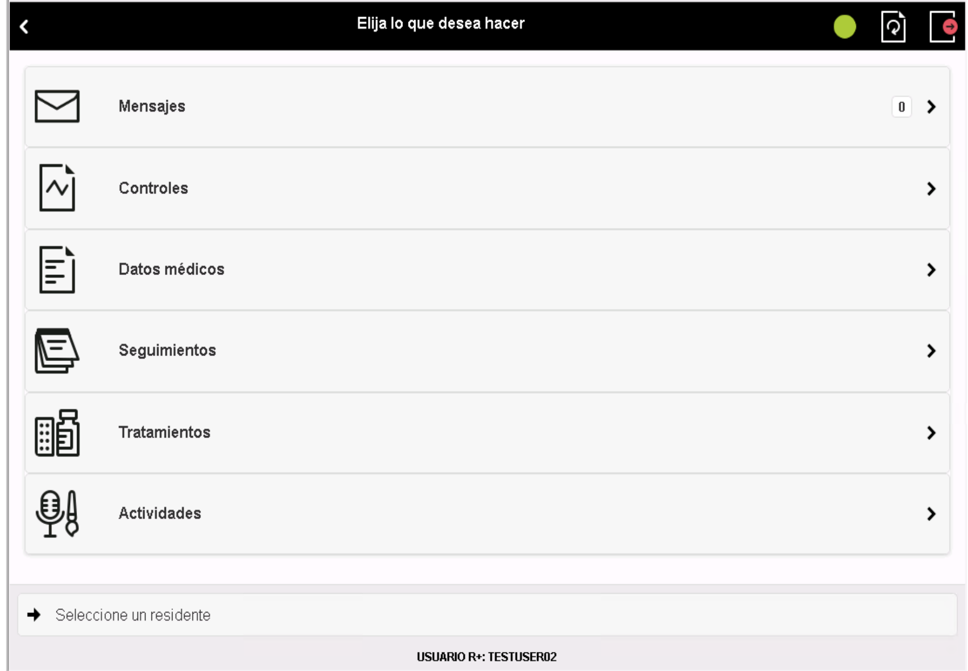

Figura 23: Vista del home de la aplicación para un usuario genérico.

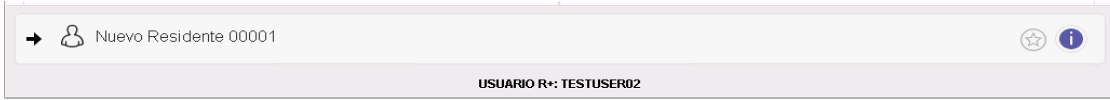

Figura 24: Barra de selección de residente, mostrando el residente seleccionado actualmente.

Hecho esto, ya podremos acceder al área de controles de enfermería, haciendo clic sobre la opción. Nos llevará a un listado de los controles de la aplicación para los que contamos con permiso de acceso. Para esta prueba, utilizaremos el control de diuresis.

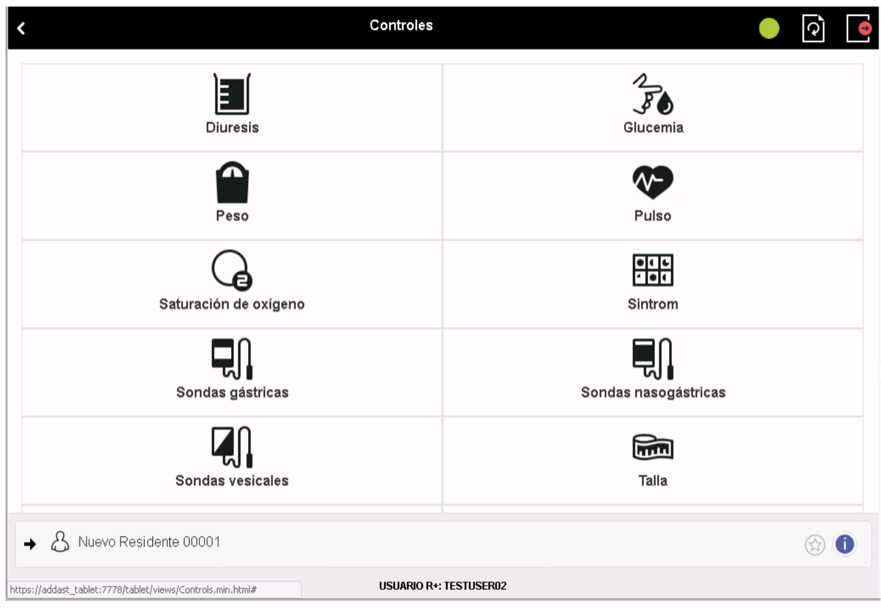

Figura 25: Menú de la sección de controles de enfermería. Se muestran todos los controles para los que el usuario cuenta con permiso de acceso.

Una vez hayamos alcanzado esta ventana, se dará por finalizada la fase de navegación. Nos encontramos en el formulario en el que se llevará a cabo la prueba, el control de diuresis (Figura 26). Podemos observar que consiste simplemente en una tabla con los registros del control, y botones para poder añadirlos, modificarlos o eliminarlos.

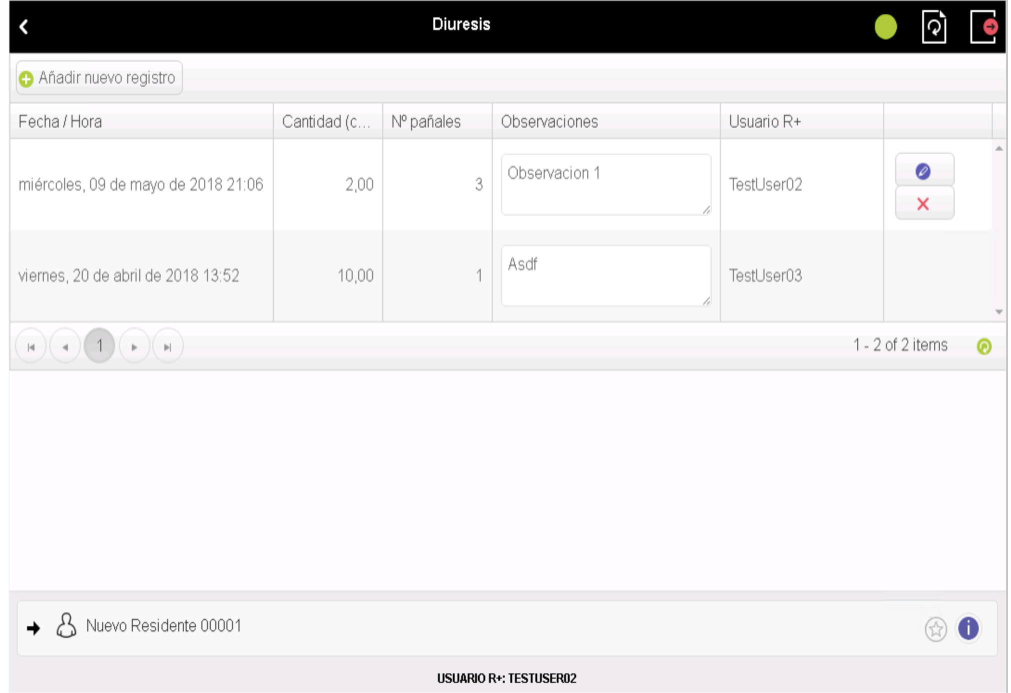

Figura 26: Ventana del control de diuresis.

Para esta prueba, únicamente nos interesa la capacidad de crearlos. Haciendo clic en el botón 'Añadir nuevo registro', nos aparecerá una ventana emergente con campos para rellenar. En la Figura 27, podemos ver que por defecto aparece la fecha actual. La prueba requiere que utilicemos una fecha en el futuro. Lo haremos sumando un año a la fecha mostrada. El resto de los campos podremos rellenarlos con cualquier valor.

Figura 27: Diálogo para la creación de un nuevo registro de diuresis.

Al aceptar la introducción con el botón 'Añadir', veremos que no se inserta nada, y en su lugar aparece un mensaje de alerta, avisando que no es posible añadir registros con fecha futura. En la prueba tendremos que comprobar la aparición y el contenido de este mensaje.

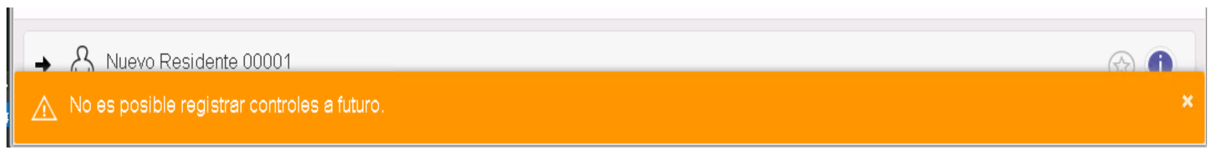

Figura 28: Mensaje de alerta, avisando que no se ha podido insertar el control de diuresis con fecha futura.

El último paso definido en la prueba era cerrar la sesión, utilizando el botón de la esquina superior derecha que aparece en la Figura 26.

En resumen, los pasos que deberá realizar la prueba son:

1. Iniciar sesión con TestUser02.
2. Seleccionar el residente 'Nuevo Residente 00001'.
3. Acceder a Controles → Diuresis.
4. Crear un nuevo registro con fecha un año mayor a la actual.
5. Comprobar que ha aparecido un mensaje de alerta con el texto "No es posible registrar controles a futuro".
6. Comprobar que no se ha insertado ningún registro nuevo en la tabla.
7. Cerrar la sesión.

5.1.3 Resultado esperado

El último apartado del diseño de una prueba de sistema es el resultado esperado. Aquí se describirán los resultados que se espera obtener al realizar las acciones o comprobaciones que se hagan a lo largo de la prueba. Por ejemplo, se podría detallar

Siguiendo con nuestro ejemplo, podemos ver que en los pasos de la prueba hemos descrito dos comprobaciones:

1. La aparición de un mensaje de información con el texto 'No es posible registrar controles a futuro'. (Paso 5)
2. No se ha introducido ningún nuevo registro con los datos de ejemplo. (Paso 6)

5.2 Automatización de la prueba

Siguiendo el diseño descrito anteriormente, procederemos a la automatización de la prueba. Se detallará paso a paso el proceso, resaltando los aspectos más relevantes.

El primer paso es crear una nueva prueba en el proyecto *ApplicationTests* de la aplicación. Deberá crearse en un directorio que siga la siguiente jerarquía: ID(*Nº Incidencia)* / PA(*Nº PA*) / PS(*Nº Prueba),* donde cada uno de esos campos se corresponde con el número de incidencia, el número de la prueba de aceptación y el código de la propia prueba, respectivamente.

Una vez creada la clase para nuestra prueba, deberemos extender de la clase *Test* del proyecto *Base*. Nos obligará a sobrescribir los métodos *SetUp, TestSteps* y *Teardown*. Estos métodos, como se explicó en capítulo anterior, se refieren a la preparación del entorno, los pasos de la propia prueba y la restauración del entorno.

```
public class PS006257 : Test
{
    protected override void SetUp()
    {
    }

    protected override void TestSteps()
    {
    }

    protected override void Teardown()
    {
    }
}
```

En primer lugar, añadiremos en la fase de preparación del entorno la restauración de los dos *scripts* SQL que generamos. Recordemos que estos *scripts* contenían los datos iniciales de la prueba. Los copiaremos en la carpeta en la que se encuentra la prueba, y utilizaremos una utilidad incluida dentro de la clase *Test*, llamada *ScriptRestorer*.

```
restoreScript($@"{ Workspace.WORKSPACE_DIRECTORY}
\ID29426\ID29426_DatosResidencia_01.sql", "DatosResidencia_001_01");
restoreScript($@"{ Workspace.WORKSPACE_DIRECTORY}
\ID29426\ID29426_DatosUsuarios.sql", "DatosUsuarios");
```

El siguiente paso será automatizar las acciones de la prueba. Empezaremos por acceder a la página web de la aplicación. Bastará con indicarle el *WebD*river la dirección de la página. En este caso, se hace referencia al servidor local de pruebas que hospeda el programa.

```
string baseURL = "https://addast_tablet:7778/tablet/views/LogIn.aspx";
webDriver.Navigate().GoToUrl(baseURL);
```

Al navegar a esta dirección, aparecerá la pantalla de *LogIn*. El inicio de sesión con el usuario TestUser02 fue definido como el Paso 1 de la prueba. Como podemos observar en la Figura 22, esta página se compone de tres controles básicos: el campo de usuario, el campo de la contraseña, y el botón 'Log in'. Para poder completar esta acción, deberemos buscar la forma de detectar estos tres componentes, e interactuar con ellos.

A continuación, en la Figura 29, presentamos un extracto del código HTML del formulario. Representa el campo de nombre de usuario y el botón de inicio de sesión. Debido a que este primer elemento no contiene ningún tipo de identificador, ni clase CSS concreta, elegiremos el atributo *data-bind* para localizarlo. Podemos apreciar que hace referencia a una variable 'user', utilizaremos. Para ello, definiremos un localizador *XPath* (Figura 30) basado en estos datos. Una vez encontrado el elemento, es trivial utilizar la operación *sendKeys* proporcionada por el *WebDriver* para introducir el texto. (Figura 30)

```
<input type = "text" data-bind = "textInput: user, hasFocus: userHasFocus,
enable: isCompatible" autofocus = "autofocus">

// …

<button id = "logInButton" class="margin-top-20 ui-btn ui-btn-b ui-shadow ui-
corner-all" type="submit" data-theme="b">Log in</button>
```

Figura 29: Fragmento del código HTML de la pantalla LogIn, que representa el campo Nombre de Usuario y el botón de inicio de sesión.

```
IWebElement userField = webDriver.FindElement(
       By.XPath("//input[contains(@data-bind, 'user')]")
);
userField.SendKeys("TestUser02");
```

Figura 30: Localizador XPath para detectar el campo de nombre de usuario.

En cambio, el botón de confirmación sí tiene asociado un identificador. Será trivial localizarlo en el documento a partir de él. En este caso, la interacción con el botón se realizará haciendo clic una vez se hayan introducido tanto el usuario como la contraseña.

```
IWebElement loginButton = driver.FindElement(
       By.Id("logInButton")
);
loginButton.Click();
```

Se nos redireccionará a la pantalla de *Home (*Figura 23*)*, donde se encuentran todas las opciones para acceder a las distintas áreas de la aplicación. Aquí, deberemos automatizar tanto la selección del residente como el acceso al área de controles. Ambas acciones se corresponden con los pasos 2 y 3 de la prueba respectivamente. Debido a la similitud entre estas acciones, se explicará únicamente la selección de la opción de Controles, para elegir un elemento de entre una lista de opciones.

```
▼<div class="ui-content">
  ▼<ul id="operationsList" class="regularList ui-nodisc-icon ui-alt-icon display-none ui-listview ui-listview-
  inset ui-corner-all ui-shadow" data-role="listview" data-inset="true" data-bind="foreach: Operations" style=
  "display: block;">
    ▶<li data-bind="attr: { id: id }, css: { 'ui-state-disabled': !canNavigate() }" class="ui-li-has-count ui-li-
    has-thumb ui-first-child ui-last-child" id="6">…</li>
    ▼<li data-bind="attr: { id: id }, css: { 'ui-state-disabled': !canNavigate() }" class="ui-li-has-count ui-li-
    has-thumb ui-first-child ui-last-child" id="1">
      ▼<a data-bind="click: navigate, event: {keypress: navigateIfEnterKeyPressed}" tabindex="1" class="ui-btn ui-
      btn-icon-right ui-icon-carat-r">
          <img data-bind="attr: { src: photoPath }" src="data:image/x-icon;base64,AA…8AAPgAAAOf/wAA+AAAA7//AAA=">
          <h2 class="position-absolute" data-bind=" text: name">Controles</h2>
          <span class="ui-li-count ui-body-inherit" data-bind="visible: countVisible, text: count" style="display:
          none;">0</span>
          ::after
        </a>
      </li>
    </ul>
  </div>
</div>
```

Figura 31: Fragmento de código HTML que representa la lista de opciones del Home.

Internamente (Figura 31), el panel de opciones es en realidad una *unordered list* (ul), compuesta por *list items* (li). Para obtener el elemento al que queremos acceder, necesitaremos obtener esta lista, y filtrar la opción que queremos por su texto. En este caso, está ubicado en una etiqueta *H2*. Podemos lograrlo obteniendo todos los ítems de esa lista, y recorriéndola hasta encontrar la opción con el texto que buscamos. En la Figura 32 presentamos una posible solución.

```
IReadOnlyCollection<IWebElement> homeOptions =
  webDriver.FindElementById("operationsList")
           .FindElements(
             By.XPath(".//H2")
            );

IWebElement controlsOption = null;
foreach (IWebElement option in homeOptions)
  {
    if (boton.Text == nombreOpcion)
    {
      controlsOption = boton;
      break;
    }
  }

controlsOption.Click();
```

Figura 32: Ejemplo de cómo seleccionar la opción Controles en la página Home.

Completada esta acción, accederemos al panel de controles de enfermería (Figura 25). De forma similar, este panel es también internamente otra lista sin orden. Buscaremos la opción de Diuresis y haremos clic sobre ella.

Una vez hayamos alcanzado el control de diuresis, representado en la Figura 26, podremos comenzar la automatización de la prueba en sí. Recordando los pasos definidos en el diseño, en este formulario debemos registrar un nuevo control con una fecha mayor a la actual (paso 4), comprobar la aparición del mensaje de alerta (paso 5) y verificar que no se ha creado ningún nuevo registro (paso 6).

Comenzaremos entonces, por la creación del nuevo registro. Haciendo clic en el botón 'Añadir Nuevo Registro', veremos que nos aparece el diálogo emergente de la Figura 27. Necesitaremos detectar todos los campos de datos para la introducción de los valores del control de diuresis.

Aunque en principio podría parecer que se encontrarían igual que como se hizo durante el inicio de sesión, en realidad estos controles son especiales: forman parte de la biblioteca Knockout[17]. Internamente cada uno de ellos es en realidad dos elementos *input,* que se intercambian según esté seleccionado o no el control. Para poder introducir el texto, necesitaremos hacer clic en el elemento contenedor del control para que se active. Hecho esto, podremos localizar el "verdadero" control, y así introducir el texto. A continuación tenemos un ejemplo una posible solución para localizar el campo "Cantidad". Necesitaremos repetir esta acción para todos y cada uno de ellos.

```
webDriver.FindElement(
       By.XPath($"//div[@data-container-for='Quantity' ]/span")
).Click();
IWebElement cantidadInput = webDriver.FindElement(By.Name("Quantity"));
cantidadInput.Clear();
cantidadInput.SendKeys(quantity.ToString());
```

Una vez introducidos los datos, confirmamos y podremos verificar que aparece el mensaje de alerta de la Figura 28. Por suerte, para realizar esta comprobación, contamos con el identificador del mensaje. Así, será sencillo, obtener el texto de este, y comparar si coincide con el mensaje esperado.

```
IWebElement toastMessage =
       webDriver.FindElement(By.Id("toast-container"))
              .FindElement(By.XPath(".//div[@class='toast-message']"));

string toastMessageText = toastMessage.Text;

assert("No es posible registrar controles a futuro." == toastMessageText,
"Comprobamos el texto del mensaje al Toast.");
```

Finalmente, es el momento de comparar los valores que se intentó registrar con los valores presentes en la primera fila de la tabla. De esta forma nos aseguraremos que no se ha registrado el control. Para leer las celdas de la primera fila de la tabla, deberemos localizar la tabla, obtener el primer *table row* (*tr*) y leer el texto de todos los campos que contiene.

[17] Knockout – Página oficial: http://knockoutjs.com/

```
IWebElement tableContent = webDriver.FindElement(By.ClassName("k-grid-content"));
IReadOnlyCollection<IWebElement> tableRows =
       tableContent.FindElements(By.XPath(".//tr"));

List<string> rowFields = new List<string>();

IWebElement row = tableRows.ElementAt(1);
foreach (IWebElement cell in row.FindElements(By.TagName("td")))
{
       rowFields.Add(cell.Text.Trim());
}
```

Para cada uno de los campos del control de diuresis, deberemos comparar sus valores con los introducidos. A continuación, presentamos un ejemplo de la prueba:

```
DateTime detectedDate = DateUtils.parseFullDate(rowFields.ElementAt(0));
string dateFormat = "dd/MM/yyyy HH:mm";

assert(!(date.ToString(dateFormat) == detectedDate.ToString(dateFormat)),
         "Comprobamos que la fecha leída no coincide con la introducida");
```

En el Anexo I presentamos el código completo de esta prueba.

Para comprobar si la prueba funciona correctamente, deberemos ejecutarla sobre los navegadores soportados, en las distintas resoluciones. El log resultante de ejecutar esta prueba debería ser similar al siguiente:

```
[21/07/2018 13:14:30] - INFO: Iniciando ejecución de la PS
[21/07/2018 13:14:38] - INFO: Iniciando sesión como TestUser02
[21/07/2018 13:14:51] - PASS: Comprobar mensaje de alerta del toast.
[21/07/2018 13:14:52] - PASS: Comprobamos que la fecha leída no coincide con la
introducida
[21/07/2018 13:14:52] - PASS: Comprobamos que la cantidad leída no coincide con
la introducida
[21/07/2018 13:14:52] - PASS: Comprobamos que el número de pañales no coincide
con el introducido
[21/07/2018 13:14:52] - PASS: Comprobamos que las observaciones no coinciden con
las introducidas
[21/07/2018 13:14:52] - INFO: Cerrando sesión
[21/07/2018 13:15:02] - PASS: Resultado de la prueba
```

Figura 33: Log resultante de ejecutar la prueba automatizada.

5.3 ¿Pruebas “desechables” o pruebas “estructuradas”?

En el ejemplo anterior, automatizamos una prueba interactuando directamente con el controlador del navegador (*WebDriver*). Si quisiéramos posteriormente crear una prueba similar que utilizara, por ejemplo, el inicio de sesión de usuario o el control de diuresis, tendríamos que duplicar el código en cada una de ellas.

En caso de que hubiera un cambio en alguno de los controles, que provocara fallos en la prueba, necesitaríamos cambiar una a una todas las pruebas. O en todo caso, automatizarlas de nuevo. Por tanto, se tratan de pruebas “desechables”, de las que una vez se han implementado, no se reaprovecha nada. En caso de fallo, se descartan y se automatiza de nuevo su funcionalidad.

La alternativa es programar pruebas “estructuradas” haciendo uso de un patrón habitual en el testeo automatizado: *Page Object***.** Cada página o componente de una página de la aplicación se corresponde con una clase, que ofrece métodos con las operaciones que podamos realizar en la misma [20]. Usándolo, y organizando el proyecto de *Application* como una jerarquía de clases que refleje la estructura de la aplicación, nos será más sencillo reutilizar el código.

Por ejemplo, la ventana de inicio de sesión podría ser mapeada en una clase llamada *LogInPage*, que ofrecería la acción de identificación en la aplicación. El código siguiente se extrajo del ejemplo de prueba automatizada del capítulo anterior. Se generalizó para poder utilizarlo desde cualquier prueba y se modificó para poder introducir nuestros propios credenciales.

```
public void logIn(string username = "Administrador", string password = "Admin")
{
    IWebElement userField = driver.FindElement(
       By.XPath("//input[contains(@data-bind, 'user')]"));
    IWebElement passwordField = driver.FindElement(
       By.XPath("//input[contains(@data-bind, 'password')]"));
    IWebElement loginButton = driver.FindElement(By.Id("logInButton"));

    userField.SendKeys(username);
    passwordField.SendKeys(password);

    loginButton.Click();
}
```

La prueba completa se ha refactorizado usando este patrón, y se incluye íntegramente en el Anexo II: *Ejemplo de código de una prueba estructurada*. Se puede apreciar una importante reducción del código necesario respecto a la prueba original.

```
LogInPage login = new LogInPage(webDriver);
login.iniciarSesion("TestUser02", "Selenium");
```

Por tanto, una ventaja importante de la aplicación de este patrón es la gran reutilización de código que se puede conseguir al agrupar todas las operaciones de un formulario en una misma clase. En la Figura 34, podemos apreciar las operaciones que ofrece la clase del control de diuresis, extraídas la mayoría de la prueba anterior.

Gracias también a la reutilización de código, nos será más sencillo gestionar uno de los mayores problemas de las pruebas de interfaz: los cambios en la interfaz de la aplicación. En caso de cambio en un formulario, como las operaciones estarán centralizadas en esta clase, simplemente deberá realizarse el cambio en un único lugar. Así evitaremos tener que modificar manualmente todas las pruebas que hacían uso de él.

```
/// <summary> Class representing the Controles -> Diuresis form. </summary>
public class DiuresisControlPage : ControlEnfermeriaPage
{
    public DiuresisControlPage(RemoteWebDriver driver) : base(driver)
    {
    }

    public void AddNewDiuresisRecord(DiuresisRecord diuresis) { /*...*/ }

    public void EditDiuresisAt(int index) { /*...*/ }

    public void DeleteDiuresisAt(int index) { /*...*/ }

    public DiuresisRecord GetDiuresisRecordAt(int index) { /*...*/ }
}
```

Figura 34: Operaciones que ofrece el Page Object para el control de enfermería de diuresis.

Otra de las ventajas, es que abstrae a las pruebas del funcionamiento del controlador del navegador. En su lugar, cada *Page Object* se encargará de localizar cada control con el que se deba interactuar y realizar acciones sobre los mismos (como clics, introducir texto, etc.) [21]. Además, gestionarán internamente las esperas de las cargas de los formularios.

6 Proceso de pruebas automatizadas para aplicaciones web

En este capítulo presentaremos el ciclo de desarrollo de la empresa y cómo se integró el proceso de pruebas automatizadas dentro del mismo. El ciclo de desarrollo de una versión de la aplicación, desde el punto de vista de las pruebas automatizadas, puede descomponerse en tres actividades fundamentales:

- La primera de ellas es la actividad de desarrollo, en la que tiene lugar el análisis de las funcionalidades, su implementación y las pruebas manuales de la aplicación. Es la actividad principal del ciclo.
- La segunda, es la actividad de automatización y mantenimiento de la batería. Tiene lugar en paralelo con la actividad de desarrollo. En ella, los encargados de la automatización desarrollan más pruebas funcionales. También se reparan o adaptan aquellas pruebas que hayan fallado en lanzamientos anteriores de la batería, ya sea por cambios en la aplicación o por estar mal programadas.
- Finalmente, tenemos la actividad de lanzamiento y revisión de la batería. Durante el periodo previo al lanzamiento de una nueva versión, una vez han concluido las pruebas manuales, se ejecuta la batería de pruebas automatizadas. Posteriormente se revisa las pruebas fallidas, en busca de fallos que puedan ser provocados por errores de aplicación.

6.1 Actividad de desarrollo iterativo e incremental

La Figura 35, es la actividad principal el ciclo de desarrollo de una nueva *release* del producto. Está basada en metodologías ágiles y sigue un desarrollo basado en *sprints*. Un *sprint* es periodo acotado de tiempo en el que tiene lugar el desarrollo. Cada *sprint* tiene una duración de un mes aproximadamente y está compuesto por unidades de trabajo, llamadas incidencias (ID). Cada *sprint* se corresponde con una versión de la aplicación, que se pone en producción al final de este [22].

El *sprint* comienza con la planificación: El *product manager* se reúne con los equipos de las distintas áreas funcionales, e identifica y prioriza las incidencias que se desarrollarán durante el *sprint*.

La primera actividad de una incidencia es su análisis. En esta actividad, el analista encargado escribirá un documento de análisis con la especificación de la incidencia. También identificará y definirá las pruebas de aceptación (PA) que la componen, además de marcar otras PA ya existentes como pruebas de regresión.

A continuación, un programador llevará a cabo el diseño y la implementación de la incidencia, basándose en la especificación y las pruebas asignadas a la ID. Una vez termine de programar el comportamiento, deberá aplicar manualmente todas las PA asociadas a la prueba, y así comprobar que cumple con el funcionamiento esperado.

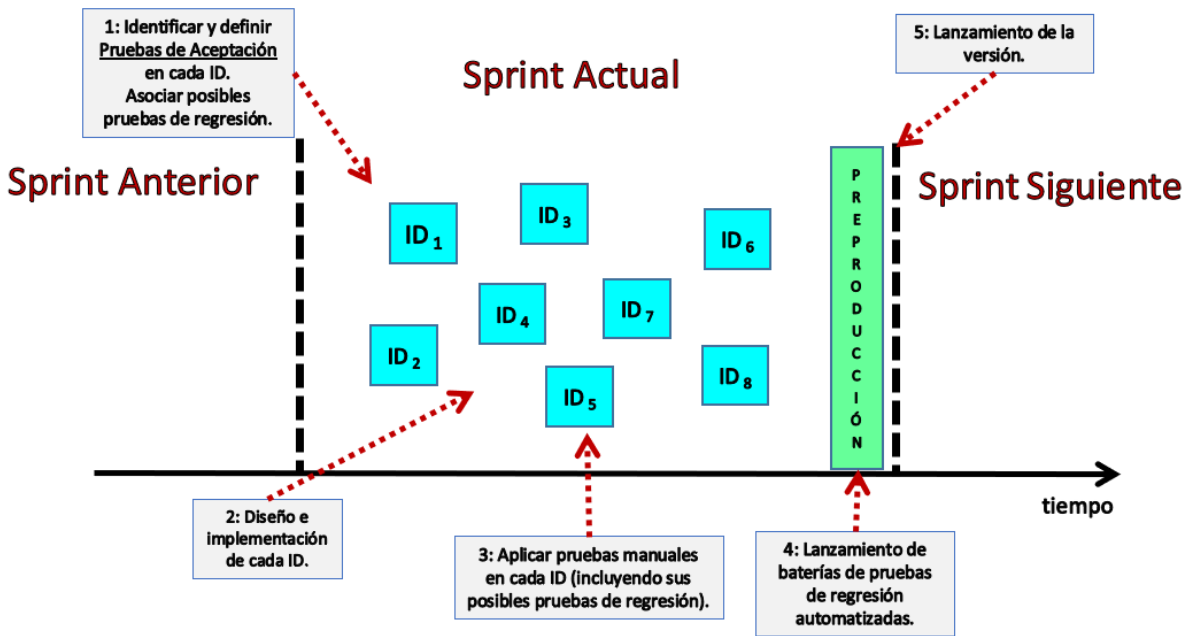

Figura 35: Representación de las actividades que se llevan a cabo durante el desarrollo de un sprint. Basado en imagen de [22].

Una vez haya terminado la implementación de todas las incidencias, se congela la versión e inicia la fase de preproducción: un periodo de aproximadamente dos semanas, donde tendrá lugar una fase de pruebas manuales de regresión cada incidencia. Los *testers* manuales aplicarán las mismas pruebas asociadas a la incidencia. También tendrá lugar la actividad de lanzamiento y revisión de las pruebas de regresión funcionales, que se explicará en el apartado 6.3 **-** *Lanzamiento de la batería y* revisión**.**

En caso de que fuera detectado un fallo, se seguirá el procedimiento descrito en el apartado 6.4 - *Gestión de los fallos*. Si el fallo está relacionado con la incidencia en la que se detectó, se avisará al programador y volverá a la fase de diseño e implementación para solucionarlo. Una vez se solucione, se repetirán todas las pruebas.

El diseño y la implementación de una incidencia puede o no tener lugar en el mismo *sprint* en el que se analizó la incidencia. En cambio, las pruebas sí que tendrán lugar la misma versión en la que se implementó.

El periodo de preproducción se dará por finalizado cuando los fallos hayan sido corregidos y se hayan completado todas las incidencias. El paso final será desplegar la versión de la aplicación a los clientes. En ese momento, dará comienzo el trabajo para la nueva versión.

6.2 Automatización y mantenimiento de la batería

Como el proceso de automatización ya ha sido explicado en detalle anteriormente en el capítulo 5 - *Automatización de pruebas funcionales*, aquí simplemente haremos un breve resumen de esta actividad. Tiene lugar en paralelo con el desarrollo. Durante esta actividad, se automatizan nuevas pruebas funcionales y se reparan aquellas que hayan fallado en lanzamientos de baterías anteriores.

Para automatizar una prueba, los *testers* manuales comunican aquellas funcionalidades que consideren prioritarias para automatizar pruebas. Hecho esto, los encargados de automatización diseñarán los casos de prueba a partir de las pruebas de aceptación definidas por el analista. Estos casos de prueba son instancias de la ejecución de una prueba de aceptación dados unos valores concretos.

Para su diseño, se seguirá las pautas descritas en el capítulo 5 - *Automatización de pruebas funcionales*, definiendo las acciones previas, los pasos de la prueba y los resultados esperados. Una vez haya sido diseñada e implementada, se comprobará que funciona correctamente en todos los navegadores soportados y se añadirá a la batería de pruebas. A partir de ese momento, la prueba se ejecutará en todos los futuros lanzamientos de esta con las demás pruebas.

Respecto a la reparación, aquellas pruebas que se hayan fallado en ejecuciones de la batería con un fallo de automatización deberán ser revisadas. Este tipo de fallo puede darse ya sea por un cambio de funcionalidad, que provoque problemas con la detección de elementos del formulario; o simplemente porque hay errores de implementación en la prueba. La prueba deberá ser modificada para adaptarla a la funcionalidad actual. Cuando sea corregida, y se ha comprobado su correcto funcionamiento, volverá a incluirse en la batería de pruebas.

6.3 Lanzamiento de la batería y revisión

Durante el periodo de preproducción, descrito dentro de la actividad de desarrollo, se ejecutará la batería de pruebas automatizadas. A continuación, se describirá la preparación previa del entorno de ejecución, el proceso de lanzamiento y la posterior revisión de los resultados.

6.3.1 Lanzamiento de la batería

Los encargados de automatización no comenzarán con la ejecución de las pruebas automatizadas hasta que los *testers* manuales hayan terminado sus pruebas. Gracias a esto, algunos de los errores más importantes ya habrán sido detectados y corregidos. Se evitarán así los fallos en cascada de las pruebas. También se les comunicará a los encargados de automatización los errores ya conocidos, de forma que podrá reducirse el tiempo de la revisión una vez se hayan clasificado los fallos.

El proceso de lanzamiento comienza con la preparación del entorno de ejecución. Primero tendrá actualizarse la aplicación en el servidor que la hospede. Ejecutando la herramienta *Actualizador* del programa principal, se instalará la versión de preproducción que se encuentra bajo pruebas.

Dado que se busca probar con las últimas versiones de los navegadores, deberán ser actualizados en las máquinas en las que se ejecuten las pruebas. Además, deberán actualizarse las bibliotecas de la herramienta basada en Selenium, para que sean compatibles con las versiones más recientes de los navegadores.

A continuación, se preparará la batería de pruebas a ejecutar. Deberán elegirse las pruebas que la compondrán. Normalmente se incluirán todas aquellas que no hayan fallado debido a un fallo de automatización en lanzamientos previos. Luego, se establecerán los parámetros de ejecución. A fecha de julio de 2018, consiste en ejecutar las pruebas en los navegadores Chrome y Edge en las resoluciones *Desktop* (1920x1080) y *Tablet* (768x1024).

La solución de la herramienta se compilará en modo *release* y se copiarán todos los artefactos generados a un zip. Ese zip se copiará en las máquinas virtuales actualizadas anteriormente. Dará comienzo la ejecución de las pruebas.

Los errores de entorno que pueden interrumpir la pruebas son frecuentes, como podría ser la aparición de un dialogo del sistema operativo. Debido a esto, se requiere cierto grado de supervisión durante este lanzamiento. Por ejemplo, revisar periódicamente que ninguna prueba se haya quedado detenida.

Terminada la primera ejecución de la batería, se relanzarán todas las pruebas fallidas, para descartar errores de entorno. Finalmente, se procede a revisar las pruebas que continúen fallando.

6.3.2 Revisión de los resultados de la batería

El objetivo de la revisión de los resultados de la batería es determinar el motivo por el cual han fallado las pruebas. Para ello, el equipo de encargados de automatización se dividirán las pruebas fallidas para e irán determinando una a una el motivo de fallo.

Los posibles motivos por los que puede fallar una prueba son los siguientes:

- **Fallos de entorno**: Causados por algún elemento del sistema externo a la prueba o la aplicación bajo pruebas. Por ejemplo, que se interponga un diálogo del sistema operativo, y no pueda interactuar con la aplicación.
- **Fallos de automatización**: Causados porque hay un error en la implementación de la prueba. Ya sea porque se implementó incorrectamente desde el principio o porque ha habido un cambio en el funcionamiento de la aplicación.
- **Fallo en la aplicación**: Causados por un fallo de la aplicación bajo pruebas. Deberá reportarse para ser corregido.

Debido a que la batería de la aplicación web en estos momentos cuenta con muy pocas pruebas, una sola persona podría realizar la revisión rápidamente. Por tanto, nos hemos basado en la operativa de revisión utilizada para las pruebas de escritorio.

Una estrategia para facilitar la revisión es intentar agrupar las pruebas fallidas por funcionalidad. De esta forma, agrupamos pruebas que probablemente compartan causa de fallo. Así aumentaremos las posibilidades de acotar su causa. Hecho esto, los encargados de automatización se dividen las pruebas para revisar, y comprobarán uno a uno los *logs* de las pruebas.

Como ya se ha visto en capítulos anteriores, los logs de pruebas contienen la información de la evolución del resultado de la prueba: los resultados de las validaciones, los mensajes de información y, lo más importante, los mensajes de error. También se incluyen imágenes del estado del navegador, en caso de que fuera posible obtenerlas.

En la mayoría de los casos, los mensajes de error serán suficientes para determinar el motivo de fallo de la prueba, ya que nos indicará la línea del código de la prueba en la que ocurre y su mensaje. Aun así, llegado el caso en que la información disponible no fuera suficiente, se recurrirá a ejecutar la prueba de forma local, e incluso depurarla paso a paso.

Una vez se haya podido averiguar el motivo de fallo, se clasificará con el resto de las pruebas y se añadirá a la categoría correspondiente. Según el tipo de fallo, deberán llevarse unas u otras acciones respecto a la prueba. Aquellas pruebas que hayan sido afectadas por un error de entorno no sufrirán ninguna modificación. Se lanzarán en el siguiente lanzamiento de la batería sin ningún cambio.

Aquellas que hayan fallado por error de automatización, deberán ser eliminadas de la batería de pruebas y reparadas. Una vez se compruebe su correcto funcionamiento, serán añadidas de nuevo.

Si se encontrara una o más pruebas cuyo fallo haya sido provocado por un fallo de la aplicación, se procederá a informar sobre él. Se elaborará un reporte de fallo, siguiendo las pautas descritas en el siguiente apartado, *A - Elaboración de un reporte de fallos*.

Cuando finalice el proceso de revisión, se redactará un informe sobre la ejecución de la batería. Deberá incluir los parámetros de ejecución, el total de pruebas ejecutadas y el número de pruebas que han fallidas. En caso de detectar un fallo de aplicación, también se incluirá el identificador de la incidencia, y una breve descripción de este. Un ejemplo de informe sería similar al presentado en la Figura 36.

Este informe se enviará a todo el equipo de *testers* y al *product manager*. También se guardará en un repositorio, para posterior consulta, como podría ser sacar estadísticas de la ejecución de las baterías.

Hola,
Hemos completado la revisión para la versión 3.8.004.

Participantes: Adriano Vega Llobell.

Parámetros de ejecución:

- **Navegadores**: Chrome, Internet Explorer y Edge
- **Resoluciones**: Desktop (1920x1080) y Tablet (768x1024).
- **Pruebas ejecutadas**: 14

TOTAL = 84 ejecuciones.

Estadísticas de la ejecución:

- Pruebas - PASS: 13
- Pruebas - FAIL: 1

Fallos Detectados:

- **I-29564**: En seguimientos de un profesional, la vista se queda "atascada" al navegar entre seguimientos bloqueados de días anteriores. (*Detectado durante la automatización*).

Pruebas con fallo al iniciar el lanzamiento: **-**
Pruebas con fallo al finalizar la revisión: **1**

Saludos

Figura 36: Ejemplo de informe que se envía al completar la revisión de la batería.

A. *Elaboración de un reporte de fallos*

Para comunicar la detección de un posible fallo de aplicación, se elaborará un reporte de fallo. Este documento servirá para registrar en el sistema de incidencias el fallo. También para notificar a las personas pertinentes, como podría ser el *tester* encargado del área funcional al que pertenece, al programador o al *Product Manager*.

Para facilitar el reporte del fallo, se ha creado una plantilla que está compuesta por los siguientes apartados. En primer lugar, se incluye una pequeña ficha para introducir los datos de incluyendo la versión en la que se detectó, la aplicación y la funcionalidad a la que afecta. Estos datos se incluyen para facilitar la trazabilidad de los fallos.

A continuación, se indicarán los pasos de reproducción, empezando una descripción de los datos de entrada y las opciones de configuración activas que puedan ser relevantes para reproducirlo, si las hubiera. Esto será de mucha utilidad para después poder descartar motivos de fallo. Se pueden llegar a adjuntar *scripts* SQL dentro de la incidencia.

Se explicarán detalladamente los pasos de reproducción del fallo, incluyendo si fuera necesario imágenes aclaratorias. Deberán añadirse tanto el resultado esperado de las acciones realizadas como el resultado real. Si se considera conveniente, también pueden añadirse observaciones adicionales sobre el reporte. Por ejemplo, si existe alguna forma de evitarlo provisionalmente, mientras se espera a que esté solucionado.

En la Figura 37 se presenta un ejemplo de reporte de fallo de aquel que aparecía descrito en el reporte de la batería de la Figura 36. Se ha obviado la ficha con la información de la aplicación.

Una vez reportado, se seguirá el proceso de gestión de fallos descrito en el siguiente apartado.

Informe de fallo I-29564

Descripción del fallo

Al intentar navegar entre los seguimientos de un profesional de un residente, si tenemos algún seguimiento bloqueado (que no se pueda modificar ni eliminar), se queda bloqueada la navegación y no es posible desplazarse.

Reproducción del fallo

Configuración relevante

Tener desactivada la opción de permitir modificar los seguimientos de días anteriores del profesional.

Pasos de reproducción

1. Crear un nuevo seguimiento (en este caso seguimiento de enfermería). Ponerle una fecha anterior a la fecha actual y guardar.
2. Crear otro seguimiento. Esta vez mantener la fecha actual.
3. Navegar al seguimiento creado en el paso 1. (Clic en el botón 'siguiente').
4. Comprobar que el seguimiento está bloqueado y no nos permite modificar ningún dato del mismo.
5. Intentar navegar al seguimiento creado en el paso 2. (Clic en el botón 'anterior').

Resultado esperado

El seguimiento mostrado actualmente cambia por el seguimiento creado en el paso 2.

Resultado Real

El seguimiento mostrado no cambia, se mantiene el seguimiento bloqueado creado en el paso 1.

Figura 37: Ejemplo de reporte de un fallo de la aplicación web, detectado durante la automatización de pruebas.

6.4 Gestión de los fallos

Una vez se ha recibido un reporte de fallo proveniente de las pruebas manuales o de las pruebas automatizadas, comienza un proceso para la gestión y la solución de los fallos, representado en la Figura 38.

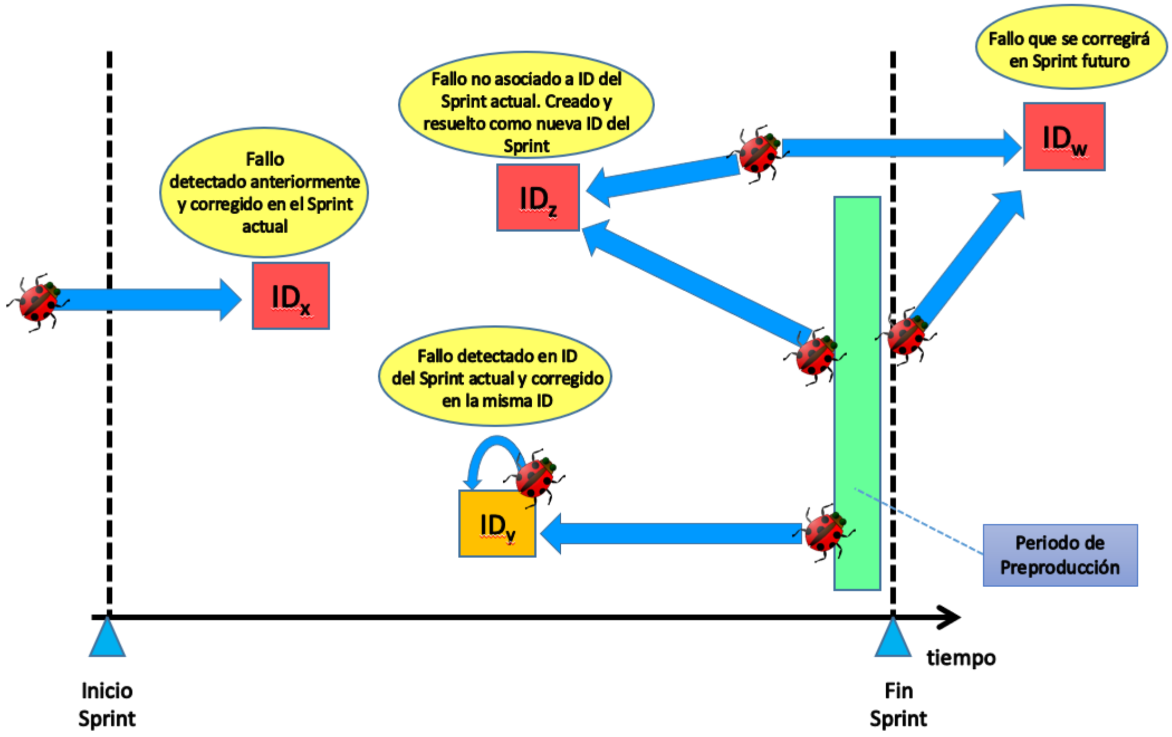

Figura 38: El proceso seguido con los fallos detectados durante el proceso de desarrollo. Basado en diagrama proveniente de [22].

El primer paso del proceso es confirmar si el fallo de aplicación existe realmente. Para ello, se replicarán manualmente los pasos descritos en el reporte de fallo. Estas pruebas manuales se extenderán hasta, al menos, las últimas tres versiones, para determinar aquella en la que se introdujo el defecto. En caso de confirmar la existencia del fallo, se deberá determinar su severidad.

La severidad es una métrica interna de la empresa que mide la importancia del fallo. Para calcularla, se tiene en cuenta la visibilidad del fallo, el número de clientes que resultarían afectados, si provoca la pérdida de datos, entre otras. La resolución de los fallos se prioriza según su severidad. Se divide en cuatro niveles distintos, ordenados de menor a mayor prioridad:

- **Severidad baja**: Fallos que no son muy visibles, debido a que se producen bajo unas circunstancias muy concretas. Pueden evitarse fácilmente hasta que sean corregidos. El número de clientes afectados es muy bajo.
- **Severidad media**: Fallos que ocurren con cierta frecuencia al utilizar alguna funcionalidad, y pueden resultar molestos para el usuario. El número de clientes afectados es relativamente bajo.

- **Severidad alta:** Fallos con una visibilidad alta y que se produzca con gran frecuencia. Posiblemente provoquen que los usuarios consulten al soporte técnico de la empresa. Afectan a un gran número de clientes.
- **Severidad crítica:** Fallos en los que se produzca pérdida de datos, quede inutilizada una funcionalidad esencial de la aplicación o suponga una vulnerabilidad de seguridad. Máxima prioridad.

Una vez determinada, se evaluará como gestionar el fallo. Teniendo en cuenta su severidad, el momento de detección y la dificultad para arreglarlo, se pueden distinguir tres casos distintos: Si el fallo fue provocado por alguna de las funcionalidades que se implementaron o modificaron en la versión actual, se intentará solucionar en la propia versión.

El siguiente caso es aquel en el que el fallo detectado no fue provocado por ninguna de las incidencias realizadas dentro de esta versión. Dependiendo de si su severidad es alta o crítica, se corregirá dentro de esta versión; incluso se podría llegar a aplazar su lanzamiento si fuera necesario.

La última posibilidad se refiere a los fallos poco prioritarios o que se considere que pueden ser difíciles de diagnosticar y arreglar. Dado este caso, se podrá aplazar su solución a una revisión u otra versión posterior. Estos fallos se crearán como incidencias, con su propio ciclo de vida asociado.

7 Validación del proceso y la herramienta

Para verificar el proceso definido y la herramienta que lo soporta, se realizaron diversas pruebas sobre ellos, que detallaremos a lo largo de este capítulo. Entre ellas, se realizó el lanzamiento de la batería de pruebas, junto con su posterior revisión, dentro del periodo de preproducción de tres versiones distintas. Estas pruebas nos han permitido encontrar carencias y mejorar tanto el proceso como la herramienta.

7.1 Automatización de pruebas

Para probar el proceso de automatización de pruebas, se contó con la ayuda de un *tester* encargado de la automatización de pruebas de escritorio. Esta prueba consistió en la definición y la automatización de una prueba de la aplicación web partiendo de su prueba de aceptación.

Se comenzó con una pequeña introducción a la estructura de la herramienta: la función de cada proyecto, las interacciones entre ellos, etc. Usando una prueba ya automatizada, se le mostró el flujo de ejecución de la herramienta, haciendo hincapié en las tres partes de la clase *Test*. Dado que la herramienta tiene una estructura similar a la utilizada en las pruebas de escritorio, no hubo muchos inconvenientes.

Hecho esto, se le explicó el funcionamiento de la aplicación web para familiarizarse con ella. Se navegó por la aplicación siguiendo la prueba de aceptación del analista. Mientras tanto, el *tester* fue anotando los pasos de la prueba, los datos y las condiciones necesarias en el estado inicial del sistema. Teniendo ya una idea más general de los pasos a seguir, se hizo la definición formal de la prueba.

Dado que la funcionalidad que se deseaba probar todavía no contaba con ninguna prueba, el usuario tuvo que automatizar el comportamiento de los formularios correspondientes. Así, se le explicó también el patrón *Page Object* y su utilidad dentro del proyecto.

Terminada la automatización de la prueba, se comprobó que funcionara correctamente en todos los navegadores soportados y se añadió a la batería de pruebas general. A partir de ese momento, la prueba se ejecutará como parte de la batería que se lanza en el periodo de preproducción.

Finalmente, se hizo un ejemplo de despliegue de la batería. Se siguió el proceso de lanzamiento de una batería, que solo contaba con esta prueba. Se compiló la solución y se copiaron los ejecutables del proyecto a la máquina virtual de pruebas. Una vez allí, se lanzó esta batería para todos los navegadores soportados y se hizo la revisión de los resultados.

7.1.1 Deficiencias encontradas por el *tester*

Gracias a esta prueba, se pudieron detectar algunas deficiencias, tanto en la definición del proceso como en la herramienta. Respecto al proceso, había secciones que estaban explicadas de forma muy ambigua o que directamente estaban incompletas. Una vez completadas, se revisaron una vez más con el *tester*.

Entre las mejoras aplicadas a la herramienta, destaca la ejecución incondicional de la fase de *Teardown* de *Test*. Antes, en caso de que ocurriera una excepción durante la preparación o durante los pasos de la prueba, no se llegaría a ejecutar el *Teardown,* y no se desharían los cambios en base de datos. Esto podía provocar que las pruebas que se ejecutaran a continuación pudieran fallar, por encontrarse el sistema en un estado inconsistente.

Se hicieron también mejoras en la gestión de las instancias del navegador, debido a que en ocasiones no se generaban correctamente, y provocaban un fallo crítico de la ejecución de la herramienta. Se procedió a tratar dichas excepciones, liberando los recursos ocupados y registrando el resultado de la prueba como fallida.

7.2 Lanzamiento de la batería de pruebas

La batería de pruebas desarrollada para la aplicación web cuenta con veintitrés pruebas, a fecha de julio de 2018. Durante el desarrollo de este trabajo, se ha ejecutado en un total de tres periodos de lanzamiento de versión, correspondientes con los meses de junio, julio y agosto de 2018. De estas ejecuciones se han derivado unas estadísticas que se presentaran en el apartado 7.2.

Gracias a estos lanzamientos, se han podido detectar y corregir fallos en la herramienta. Por ejemplo, en caso de fallo crítico de la herramienta, se perdían todos los resultados de ejecución, ya que no se gestionaba correctamente su guardado. La corrección del fallo limitó la perdida de datos a únicamente la prueba en ejecución.

Aunque el número de pruebas desarrolladas pueda parecer bajo, se espera que en los próximos meses los encargados de automatización comiencen a desarrollar pruebas para esta y otras aplicaciones web de la empresa. Así, aumentará el tamaño y la cobertura de la batería.

A continuación, se presentan una serie de estadísticas obtenidas durante el desarrollo del proyecto. Se basan en métricas internas utilizadas por la empresa. Por ejemplo, el número de fallos detectados, severidad de los fallos de aplicación, etc.

7.2.1 Automatización de pruebas

La automatización de pruebas comenzó en abril de 2018, cuando la herramienta basada en Selenium aún se encontraba en fases tempranas de su desarrollo. Se decidió aprovechar y utilizar pruebas reales para probar su correcto funcionamiento.

Al inicio del proyecto, se esperaba alcanzar el número de al menos cincuenta pruebas automatizadas. Al final, han acabado siendo veintitrés las pruebas desarrolladas. El principal motivo que impidió alcanzar este objetivo fue la necesidad de pulir el funcionamiento de la herramienta.

Aun así, a fecha de julio de 2018 ha comenzado la formación de un *tester* de la aplicación de escritorio en el uso de la herramienta basada en Selenium. En septiembre 2018 se prevé que comience la automatización de pruebas para esta aplicación web. También paulatinamente se irán incorporando el resto de los encargados de automatización, hasta que finalmente todos conozcan su funcionamiento.

7.2.2 Estadísticas de ejecución de la batería durante el periodo de preproducción

La ejecución de la batería en período previo al lanzamiento de una nueva versión solo pudo realizarse en tres ocasiones: las versiones lanzadas en junio, julio y agosto de 2018. Se ejecutaron todas las pruebas disponibles con los siguientes parámetros de ejecución:

- **Navegadores**: Google Chrome y Edge.
- **Resoluciones**: *Desktop* (1920x1080) y *Tablet* (768x1024).

La gráfica muestra el número total de pruebas ejecutadas, y compara aquellas que han terminado con resultado *PASS* (se han completado correctamente) y *FAIL* (ha ocurrido algún problema o fallo durante su ejecución). Los resultados finales de la ejecución de la batería fueron los siguientes: Una única prueba fallida en cada revisión, mientras que las demás se ejecutaron correctamente. No se detectó ningún fallo de aplicación.

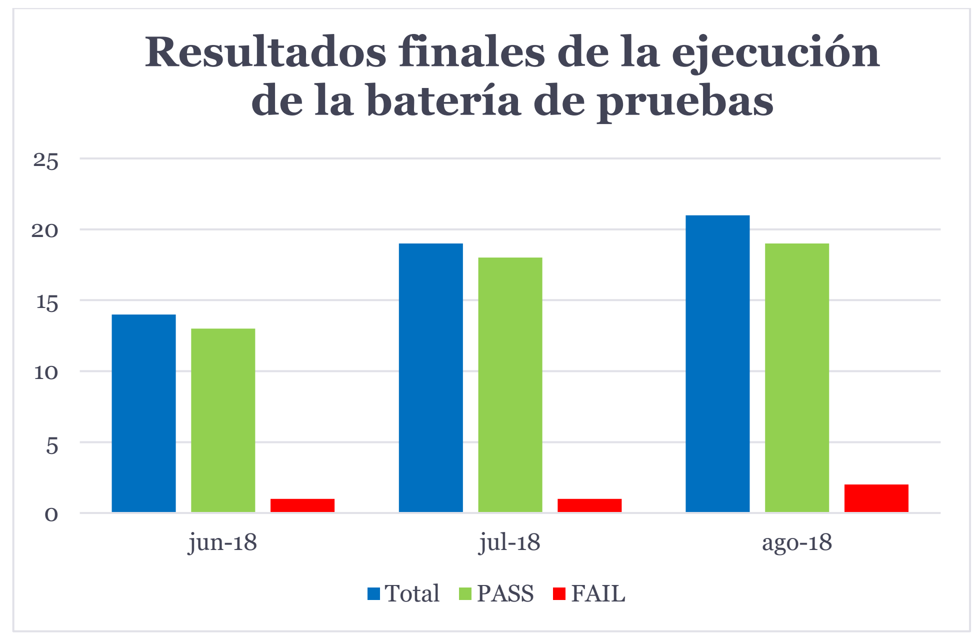

Figura 39: Gráfica que representa los resultados finales de la ejecución de la batería de pruebas, mostrando el total de pruebas ejecutadas y el resultado.

Aunque en las estadísticas anteriores no se refleje, en ocasiones algunas pruebas fallan la primera vez que se lanza la batería debido a errores de entorno. Así, una vez termina su ejecución, se procede a relanzar las pruebas fallidas. En caso de que vuelvan a fallar, se procede a revisar el motivo.

La Figura 40 representa el número de pruebas fallidas agrupadas por el motivo de su fallo. Analizando las estadísticas, podemos ver que no se detectó ningún fallo de aplicación. Los fallos detectados en las pruebas fueron principalmente debido a fallos de entorno. Concretamente, dos pruebas fallaron porque se cerraba el navegador por falta de memoria en el equipo. El fallo de automatización detectado fue provocado por un cambio de la funcionalidad de la aplicación. La prueba hubo de actualizarse para reflejar la nueva funcionalidad.

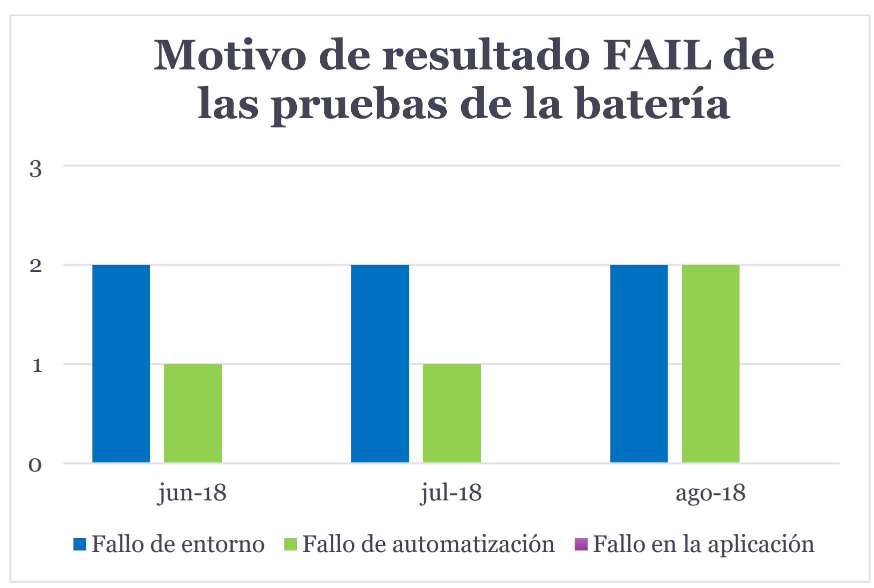

Figura 40: Gráfica que representa el número de pruebas que han fallado, agrupadas por el motivo de fallo.

7.2.3 Fallos detectados

Desde el comienzo del proyecto en marzo de 2018, se han detectado cuatro fallos confirmados en la aplicación web. Todos fueron detectados durante la fase de automatización de pruebas. A continuación, se muestra la estadística del momento en el tiempo que se encontró.

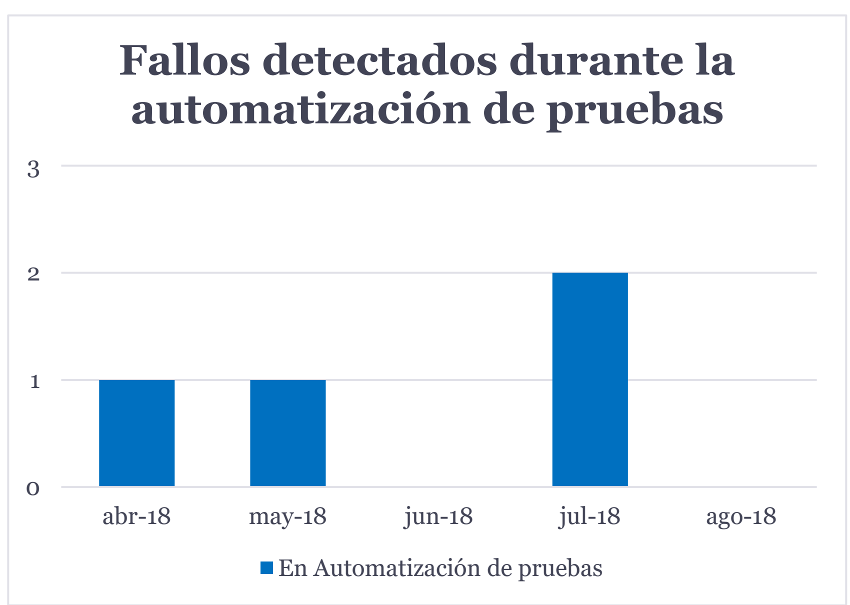

Figura 41: Gráfica que representa el número de fallos detectados durante la automatización de pruebas según el mes.

Uno de ellos se detectó durante la automatización de la prueba de ejemplo presentada en el capítulo 5: Al introducir una fecha posterior a la actual durante el registro o la modificación de un control, ocurría un error interno, del que no aparecía ningún aviso, y no se guardaban los datos. La corrección de fallo consistió en tratar estas excepciones y añadir un mensaje de alerta. La prueba de aceptación asociada a esta funcionalidad se modificó y, por tanto, también el caso de prueba desarrollado.

La siguiente gráfica describe el número de fallos agrupados por su severidad. La severidad es una métrica interna de la empresa en la que se tiene en cuenta la visibilidad del fallo, el número de clientes que resultarían afectados, si ocurre perdida de datos, entre otras. En el apartado *6.4 - Gestión de los fallos* se describieron los distintos niveles de severidad representados.

Como podemos ver en la figura, el fallo de mayor severidad detectado fue de severidad media, debido principalmente al número potencial de clientes afectado. Los otros tres fallos detectados fueron simplemente fallos de campos que no se mostraban correctamente.

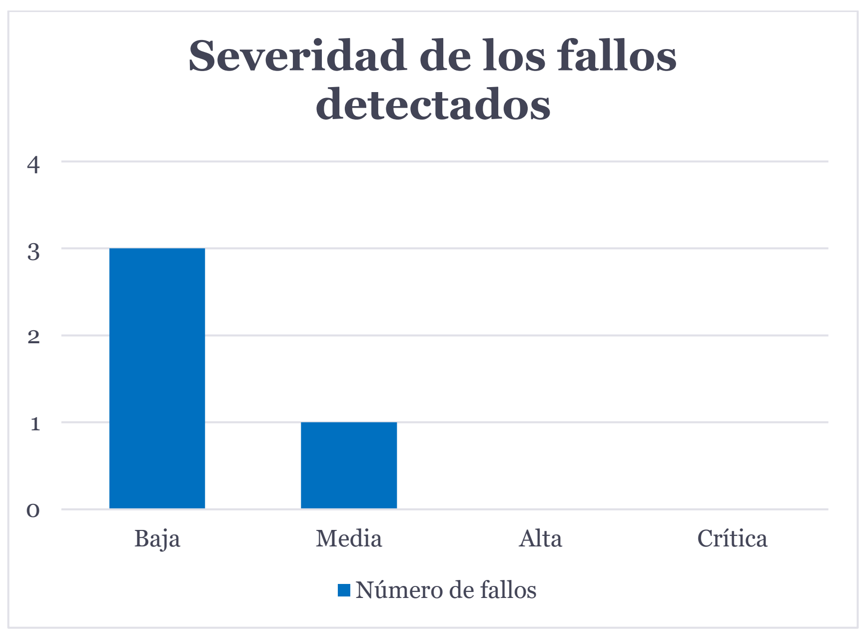

Figura 42: Gráfica que representa el número de fallos detectados agrupados por la severidad.

8 Resumen y cronograma del desarrollo

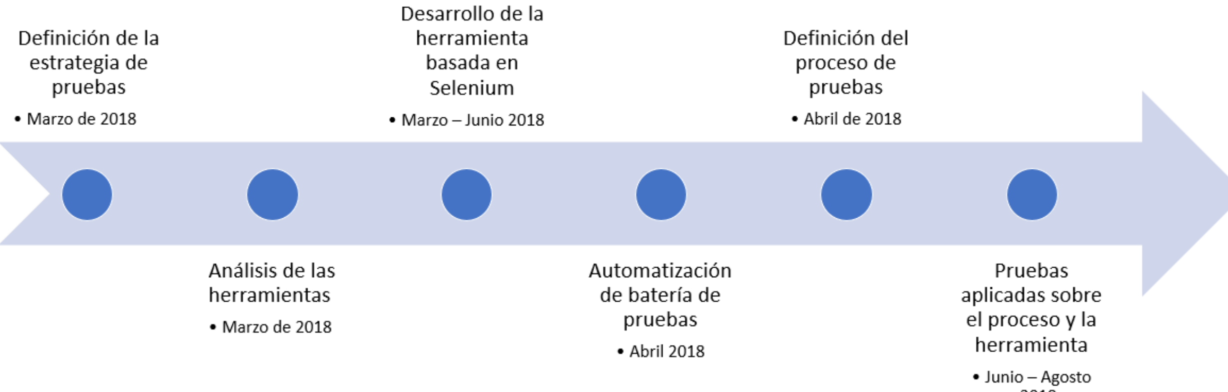

Figura 43: Cronograma que representa las fases del trabajo desarrollado.

El desarrollo de este proyecto comenzó con el análisis del problema. Se deseaba automatizar pruebas de aceptación funcionales de las aplicaciones web desarrolladas por la empresa, de manera similar a como se estaba haciendo con las aplicaciones de escritorio. Debía por tanto establecerse un proceso para el desarrollo de estas pruebas, y contar con herramientas que lo soporten.

Tomando como base una de las aplicaciones, se diseñó una estrategia de pruebas. A partir de esta se detallaron los requisitos que debía cumplir la herramienta elegida. Basándonos en estos requisitos, tuvo lugar una fase de investigación previa sobre las tecnologías disponibles para las pruebas de aplicaciones web. Se llevó a cabo mediante una comparativa basada en la implementación de una prueba de ejemplo.

Se evaluaron las características de tres herramientas distintas: Pupeteer, Katalon Studio y Selenium. Finalmente, se eligió el desarrollo de un proyecto *ad hoc* basado en Selenium. Principalmente por su simplicidad, ya que no eran necesarias herramientas muy sofisticadas, y porque facilitaría la integración con la infraestructura de gestión de las pruebas ya existente.

A continuación, se procedió al diseño de la herramienta y su posterior implementación. Para su diseño, se tomaron como referencia las herramientas de pruebas de escritorio y se tuvieron en cuenta las recomendaciones ofrecidas en el ámbito de las pruebas web. El resultado fue el desarrollo de una herramienta con un diseño modular, que permitía la ejecución de las pruebas en múltiples navegadores a distintas resoluciones.

A la vez que se desarrollaba la herramienta, se implementó una pequeña batería de pruebas para la aplicación web. Así se pudo verificar el funcionamiento de la herramienta, al mismo tiempo que se detectaban fallos y nuevos requisitos.

Una vez el desarrollo se encontró en fases más avanzadas, fue definido el ciclo de vida de las pruebas. Las fases descritas cubrían desde la definición de la prueba de aceptación por parte de un analista; pasando por la selección a automatizar por un *tester*; y acabando con la definición e implementación del caso de prueba por parte de los encargados de la automatización.

A continuación, se analizó como integrar en el ciclo de desarrollo de la empresa la ejecución de esta batería de pruebas y su posterior revisión. Así se terminó de definir el proceso de pruebas automatizadas. Puede descomponerse en tres actividades principales: la actividad de desarrollo de la aplicación, la actividad de automatización y mantenimiento de la batería pruebas, y la actividad de lanzamiento y revisión de los resultados.

Finalmente, para verificar el proceso y el correcto funcionamiento de la herramienta, se aplicó el proceso completo durante el desarrollo de tres versiones distintas, durante los meses de junio, julio y agosto de 2018 respectivamente. También fue instruido un trabajador externo a este proyecto, que continuará con el desarrollo de pruebas para otras aplicaciones.

9 Conclusiones y trabajos futuros

Para concluir la memoria, en este capítulo se presentan a modo de resumen algunas conclusiones sobre el trabajo realizado, inconvenientes encontrados, entre otros. También se detallarán algunas propuestas para trabajos futuros, en caso de que se deseara continuar evolucionando el proceso y la herramienta.

9.1 Conclusiones

Recordemos que el principal objetivo de este trabajo era la implantación de un proceso de pruebas automatizadas para aplicaciones web dentro del ciclo de desarrollo de una empresa. Debía cubrir la automatización, su posterior ejecución y revisión de los resultados. Para ello también se hubo de desarrollar una herramienta basada en Selenium, y una pequeña batería de pruebas para una aplicación web.

Comenzaremos exponiendo las conclusiones respecto del desarrollo de la herramienta y la batería de pruebas. Inicialmente, se estimó que el desarrollo de la herramienta sería corto, con una duración aproximada de dos o tres meses, dedicando el resto del tiempo a la automatización de pruebas. Sin embargo, su desarrollo se alargó por la necesidad de pulir la gestión del ciclo de ejecución de las pruebas. Concretamente, en lo referente al tratamiento de errores durante la ejecución de una prueba, o la gestión de las instancias del navegador. Con los sucesivos lanzamientos de la batería se fueron detectando y arreglando estas irregularidades, hasta alcanzar cierta estabilidad.

Pese a estos inconvenientes, se logró cumplir los requisitos establecidos. El primero de ellos, la ejecución *cross-browser* de las pruebas: se logró dar soporte a los navegadores Chrome, Edge, Internet Explorer y Firefox; aunque estos dos últimos no se utilizaran para pruebas de la batería desarrollada.

Otro de estos requisitos era la ejecución de las pruebas *responsive* o multiresolución, de forma que cada prueba pudiera ejecutarse simulando las características de distintos tamaños de pantalla. Actualmente solo soporta las resoluciones 1920x1080, para escritorio, y 768x1024 para simular la resolución de una *tablet*. De ser necesario, será sencillo añadir más resoluciones si lo requiriera otra aplicación web.

También, gracias a la aplicación del patrón *Page Object* para representar los formularios de la aplicación web, se logró independizar las pruebas del navegador en el que se ejecutaría. En caso de requerir alguna acción especial, se encontraría ubicada en la clase del formulario. Así, no será necesario adaptar secciones de una prueba a un navegador concreto, sirviendo el mismo código para cualquiera de ellos.

Respecto a la batería o *suite* de pruebas desarrollada, en el capítulo *7 - Validación del proceso y la herramienta* se presentaron con más de detalle estos datos, junto con estadísticas la ejecución del proceso, por lo que no haremos mucho hincapié. El objetivo inicial era alcanzar la cifra de cincuenta pruebas desarrolladas, y así contar con una batería robusta para probar la aplicación web. A fecha de julio de 2018 la batería cuenta con apenas veintitrés. Es decir, se desarrollaron menos pruebas de las esperadas a causa de los contratiempos en el desarrollo de la herramienta ya explicados anteriormente.

Pese a ser pocas pruebas, gracias a la automatización de estas se detectaron hasta cuatro errores distintos en la aplicación. Todos ellos ya han sido corregidos, y sus correcciones ya han sido desplegadas a los clientes. Además, se han automatizado pruebas para la batería, que comprobarán que estos fallos no ocurran de nuevo en un futuro.

Para finalizar, cabe destacar que el proceso completo ha llegado a aplicarse en tres *sprints* o versiones distintas de la aplicación. En las dos últimas, fue aplicado *testers* encargados de automatización externos al desarrollo de este proyecto. El objetivo de estas pruebas era comprobar que el proceso estuviera bien definido y pudieran seguirlo paso a paso.

Estas ejecuciones ayudaron a detectar errores o ambigüedades en la definición del proceso, que fueron solventadas. Por ejemplo, el apartado de despliegue y ejecución de la batería de pruebas tuvo que ser ampliado con instrucciones más detalladas, debido a que los usuarios se quedaban bloqueados y no sabían cómo continuar.

Se ha comprobado por tanto que el proceso de pruebas automatizadas y la herramienta que le da soporte se han implantado satisfactoriamente dentro del ciclo de desarrollo de la empresa. Su aplicación continuará en futuros *sprints,* por parte de los *tester* encargados de automatización. Aun así, deberá mantenerse una continua monitorización sobre las futuras ejecuciones de este, para detectar posibles optimizaciones y correcciones que se le puedan aplicar.

En julio de 2018 comenzó la formación de un *tester* en el uso de la herramienta, y se prevé que en septiembre de 2018 comience a desarrollar pruebas de otros productos, siguiendo las guías descritas en este trabajo. Así, paulatinamente se incorporarán todos los encargados de automatización, y conocerán el funcionamiento de la herramienta.

9.2 Relación con asignaturas cursadas en la carrera

Este trabajo presenta una relación con varias de las asignaturas cursadas a lo largo de la carrera. Nos centraremos en la relación con tres de ellas: las asignaturas *Proceso de Software*, *Diseño de Software* y *Mantenimiento y Evolución del Software*.

En primer lugar, la relación con *Proceso de Software* es evidente. Se trata de la mejora de un proceso de desarrollo existente mediante la implantación de un proceso de pruebas automatizadas para productos de una empresa. Así, se buscaba ganar confianza en los lanzamientos de las versiones, intentado reducir los errores de aplicación que llegaban al cliente.

Con *Diseño de Software*, la relación radica en que la necesidad de diseñar y planificar el desarrollo de la herramienta. Se siguió un diseño modular con funcionalidad claramente diferenciada. También se aplicaron diversos patrones de diseño, como el ya mencionado *Page Object* para la automatización de las acciones de los formularios.

Finalmente, presenta relación con *Mantenimiento y Evolución del Software*. Principalmente debido a que la integración de este proceso de pruebas formaba parte del mantenimiento y evolución de un producto existente. En esta asignatura también se estudió el *Modelo en V*, presentado en el capítulo 2, junto con los distintos tipos de pruebas que se pueden realizar sobre un producto.

9.3 Conclusiones personales

En cuanto a las conclusiones personales, realizar este proyecto me ha servido para profundizar mis conocimientos en cuanto a la gestión de procesos: analizar el proceso de desarrollo existente dentro de una empresa, y buscar maneras de optimizarlo. En este caso, mediante la introducción de un proceso de pruebas automatizadas para aplicaciones web.

A su vez, he aprendido sobre la planificación de una estrategia de pruebas de una aplicación web. Por ejemplo, a identificar los factores que dictan los tipos de pruebas a realizar, a elegir los navegadores sobre los que ejecutar las pruebas o a emular el funcionamiento de la aplicación en distintos dispositivos usando distintas resoluciones.

También me ha servido para aprender más sobre la automatización del manejo de navegadores usando Selenium. Aunque ya contaba con unas nociones sobre su uso gracias a la asignatura *Integración e Interoperabilidad*, fue necesario profundizar más en su funcionamiento. Principalmente, para llevar a cabo la gestión del ciclo de vida de los navegadores, el tratamiento de errores, entre otros.

9.4 Trabajos futuros

9.4.1 Integración con la infraestructura existente

La primera de ellas es la integración las pruebas de aplicaciones web con la infraestructura existente de la empresa. Dicha infraestructura, está compuesta por diversas herramientas que gestionan una gran cantidad de información relacionada con los procesos de desarrollo de la empresa: Las incidencias, la documentación, las pruebas, etc.

La integración consistiría en la adaptación de la herramienta basada en Selenium a la aplicación *Lanzador*. Esta aplicación es usada actualmente para el lanzamiento de las baterías de pruebas de escritorio. Durante el lanzamiento, distribuye las pruebas a ejecutar entre distintas máquinas virtuales, según sus capacidades y los requisitos de la prueba.

También gestiona y almacena los resultados y *logs* de las baterías, a modo de histórico. Dentro de esta aplicación se lleva a cabo la revisión de las pruebas de escritorio: pueden marcarse las pruebas fallidas con el motivo de su fallo, asociar fallos de aplicación a las pruebas pertinentes, entre otros.

Esta adaptación serviría para homogeneizar los procesos de las pruebas de escritorio y de navegador. Los encargados de automatización tendrían más facilidades para el lanzamiento de las pruebas, pudiendo lanzar ambas baterías de forma conjunta. La revisión de ambas baterías se realizaría con las mismas herramientas y será más sencillo establecer la trazabilidad entre los fallos y el momento de su detección; etc.

9.4.2 Ejecución de las pruebas en entornos móviles

La otra propuesta sería la creación de entornos de ejecución basados en Android y iOS. Ejecutar las pruebas en un entorno móvil ayudaría a detectar fallos o incompatibilidades que no se produzcan en las versiones de escritorio. Actualmente, se dispone únicamente de máquinas virtuales basadas en Windows 7 y Windows 10, por lo que los resultados podrían resultar sesgados.

El entorno podría crearse haciendo uso de emuladores, tanto de Android como de iOS, y añadiendo soporte para la biblioteca Appium en la herramienta desarrollada. Esta biblioteca permite la ejecución de las pruebas de manera nativa en Android y iOS. Así, se lograría simular con mayor confianza la ejecución de las pruebas en dispositivos móviles y *tablets*.

10 Referencias

[1] A. Ndegwa, «What is a Web Application?» [En línea]. Disponible en: https://www.maxcdn.com/one/visual-glossary/web-application/? [Accedido: 18-abr-2018].
[2] «Global Ecommerce: Statistics and International Growth Trends [Infographic]», *Shopify Plus Blog*, 01-sep-2017. [En línea]. Disponible en: https://www.shopify.com/enterprise/global-ecommerce-statistics. [Accedido: 15-abr-2018].
[3] S. Fernández Rodríguez, «Un marco de trabajo para el desarrollo y ejecución de pruebas automatizadas aplicadas al front-end de una aplicación web», *Universidad Politécnica de Valencia*, sep. 2016.
[4] D. A. Esteve Ambrosio, «Implantación de un proceso de automatización de pruebas para una aplicación software», *Universidad Politécnica de Valencia*, sep. 2015.
[5] «"Fundamentos de la Calidad del Software" - Apuntes de la asignatura Calidad de Software (CSO)». ETSINF, 2016.
[6] R. Gopalaswamy y S. Desikan, *Software Testing: Principles and Practices*. Pearson, 2007. Ch. 2, 3, 5.
[7] P. Letelier, «Gestión de requisitos dirigida por las pruebas de aceptación - Apuntes de la asignatura Proyecto de Ingeniería de Software». 2016.
[8] J. Humble y D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010. Ch 8.
[9] «Types of Integration Testing in Software Testing», *Online QA*, 12-mar-2015. [En línea]. Disponible en: http://onlineqa.com/integration-testing-and-types/. [Accedido: 07-jun-2018].
[10] M. Wacker, «Just Say No to More End-to-End Tests», *Google Testing Blog*. [En línea]. Disponible en: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html. [Accedido: 12-may-2018].
[11] G. D. Everett y R. McLeod Jr., *Software Testing: Testing Across the Entire Software Development Life Cycle*. John Wiley & Sons, 2007. Ch 7.
[12] C. Mills et al., «Introduction to cross browser testing», *MDN Web Docs*. [En línea]. Disponible en: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Cross_browser_testing/Introduction. [Accedido: 01-may-2018].
[13] «Browser Market Share Worldwide», *StatCounter Global Stats*. [En línea]. Disponible en: http://gs.statcounter.com/browser-market-share. [Accedido: 01-may-2018].
[14] J. Cao, «Responsive vs. Adaptive Design: What's the Best Choice?», *Studio by UXPin*, 16-abr-2015. [En línea]. Disponible en: https://www.uxpin.com/studio/blog/responsive-vs-adaptive-design-whats-best-choice-designers/. [Accedido: 15-ago-2018].
[15] D. Park, «UI testing with Puppeteer and Mocha», *Medium*, 16-mar-2018. [En línea]. Disponible en: https://medium.com/@dpark/ui-testing-with-puppeteer-and-mocha-8a5c6feb3407. [Accedido: 16-ago-2018].
[16] Y. Eluwande, «Introduction to Headless Browser Testing», *LogRocket*, 11-sep-2017. [En línea]. Disponible en: https://blog.logrocket.com/introduction-to-headless-browser-testing-44b82310b27c. [Accedido: 13-may-2018].
[17] Brian, «Best Automation Testing Tools for 2018 (Top 10 reviews)», *Medium*, 26-oct-2017. [En línea]. Disponible en: https://medium.com/@briananderson2209/best-automation-testing-tools-for-2018-top-10-reviews-8a4a19f664d2. [Accedido: 21-abr-2018].
[18] S. Avasarala, «Chapter 1: Introducing WebDriver and WebElements», en *Selenium WebDriver Practical Guide*, 2014.
[19] A. Chemakin et al., «WebDriver, W3C Editor's Draft 21 August 2018». [En línea]. Disponible en: https://w3c.github.io/webdriver/. [Accedido: 23-ago-2018].
[20] D. Kovalenko, *Selenium Design Patterns and Best Practices*. Packt Publishing, 2014. Ch. 7.
[21] Selenium Project, «Test Design Considerations», *Selenium Documentation*. [En línea]. Disponible en: https://www.seleniumhq.org/docs/06_test_design_considerations.jsp#. [Accedido: 23-ago-2018].
[22] P. Letelier, «Apuntes de la asignatura Proyecto de Ingeniería de Software (PIN) y Proceso de Software (PSW)». 2016.

Anexos

I. Ejemplo del código de una prueba no estructurada

```
public class PS006257 : Test
{
    protected override void SetUp()
    {
        restoreScript($@"{ Workspace.WORKSPACE_DIRECTORY }
          \ID29426\ID29426_DatosResidencia_01.sql", "DatosResidencia_001_01");
        restoreScript($@"{ Workspace.WORKSPACE_DIRECTORY }
          \ID29426\ID29426_DatosUsuarios.sql", "DatosUsuarios");
    }

    protected override void TestSteps()
    {
        // Navegar a la página de la aplicación.
        string baseURL = "https://test_tablet:7778/tablet/views/LogIn.aspx";
        webDriver.Navigate().GoToUrl(baseURL);

        // 1. Login con 'TestUser02'.
        IWebElement userField = webDriver.FindElement(
            By.XPath("//input[contains(@data-bind, 'user')]")
        );
        IWebElement passwordField = webDriver.FindElement(
            By.XPath("//input[contains(@data-bind, 'password')]")
        );
        IWebElement loginButton = webDriver.FindElement(By.Id("logInButton"));

        userField.SendKeys("TestUser02");
        passwordField.SendKeys("Selenium");
        loginButton.Click();

        info("Iniciando sesión como TestUser02");

        // 2. Seleccionar al residente 'Nuevo Residente 00001'.
        IWebElement selectResidentHeader =
            webDriver.FindElement(By.Id("selectResidentHeader"));
        selectResidentHeader.Click();

        string residentName = "Nuevo Residente 00001";
        IReadOnlyCollection<IWebElement> residents =
            webDriver.FindElement(By.Id("residentsList"))
                     .FindElements(By.XPath(".//span"));

        foreach (IWebElement resident in residents)
        {
            if (resident.Text == residentName)
            {
                resident.Click();
                break;
            }
        }
```

```
// 3. Acceder a Controles -> Diuresis del 'Nuevo Residente 00001'.
string optionName = "Controles";
IReadOnlyCollection<IWebElement> listaOpcionesMenuPrincipal =
    webDriver.FindElementById("operationsList")
             .FindElements(By.XPath(".//H2"));

IWebElement homeOption = null;
foreach (IWebElement button in listaOpcionesMenuPrincipal)
{
    if (button.Text == optionName)
    {
        homeOption = button;
        break;
    }
}
homeOption.Click();

string controlName = "Diuresis";
IWebElement controlsList =
    webDriver.FindElement(By.Id("controlsDiv"));
IReadOnlyCollection<IWebElement> controls =
    controlsList.FindElements(
       By.XPath(".//div[contains(@class, 'controlsList')]/p")
    );
IWebElement controlsOption = null;
foreach (IWebElement button in controls)
{
    if (button.Text == controlName)
    {
        controlsOption = button;
        break;
    }
}
controlsOption.Click();

// 4. Crear un nuevo registro de diuresis con fecha un año mayor a la
//    actual.

// Datos de prueba:
DateTime date = DateTime.Now.AddYears(1);
double quantity = 2.0;
int diapers = 2;
string observations = "Observation";
string dateFormat = "dd/MM/yyyy HH:mm";

IWebElement addButton = webDriver.FindElement(
    By.ClassName("k-grid-add")
);
addButton.Click();

IWebElement dateInput = webDriver.FindElement(By.Name("DateTime"));
dateInput.Clear();
dateInput.SendKeys(date.ToString(dateFormat));
```

```
webDriver.FindElement(
    By.XPath($"//div[@data-container-for='Quantity']/span")
).Click();

IWebElement cantidadInput =
    webDriver.FindElement(By.Name("Quantity"));
cantidadInput.Clear();
cantidadInput.SendKeys(quantity.ToString());

webDriver.FindElement(
    By.XPath($"//div[@data-container-for='IncontinenceNumber']/span")
).Click();

IWebElement panalesInput =
    webDriver.FindElement(By.Name("IncontinenceNumber"));
panalesInput.Clear();
panalesInput.SendKeys(diapers.ToString());

IWebElement observacionesInput =
    webDriver.FindElement(By.Name("Observations"));
observacionesInput.Clear();
observacionesInput.SendKeys(observations);

IWebElement acceptButton =
    webDriver.FindElement(By.ClassName("k-grid-update"));
acceptButton.Click();

// 5. Comprobar que ha aparecido el mensaje de alerta 'No es posible
//    registrar controles a futuro.'.
IWebElement toastMessage =
    webDriver
    .FindElement(By.Id("toast-container"))
    .FindElement(By.XPath(".//div[@class='toast-message']"));
string toastMessageText = toastMessage.Text;

assert("No es posible registrar controles a futuro." ==
       toastMessageText,
       "Comprobamos el texto del mensaje al Toast.");

// 6. Comprobamos que no se ha creado ningún nuevo registro.
List<string> rowFields = new List<string>();
IWebElement tableContent = webDriver.FindElement(
    By.ClassName("k-grid-content")
);
IReadOnlyCollection<IWebElement> tableRows =
    tableContent.FindElements(By.XPath(".//tr"));

IWebElement row = tableRows.ElementAt(1);
foreach (IWebElement cell in row.FindElements(By.TagName("td")))
{
    rowFields.Add(cell.Text.Trim());
}
```

```
            DateTime detectedDate =
               DateUtils.parseFullDate(rowFields.ElementAt(0));
            double detectedQuantity = double.Parse(rowFields.ElementAt(1));
            int detectedDiapers = int.Parse(rowFields.ElementAt(2));
            string detectedObservations = rowFields.ElementAt(3);
            string detectedUser = rowFields.ElementAt(4);

            assert(!(date.ToString(dateFormat) ==
                   detectedDate.ToString(dateFormat)),
                   "Comprobamos que la fecha leída no coincide con la
                    introducida");
            assert(quantity == detectedQuantity,
                   "Comprobamos que la cantidad leída no coincide con la
                    introducida");
            assert(diapers == detectedDiapers,
                   "Comprobamos que el número de pañales no coincide con el
                    introducido");
            assert(observations == detectedObservations,
                   "Comprobamos que las observaciones no coinciden con las
                    introducidas");

            IWebElement logOutButton webDriver.FindElement(
              By.Id("logOutButton"));
            logOutButton.Click();

        }

        protected override void Teardown()
        {
        }
    }
```

II. Ejemplo de código de una prueba *estructurada*

```
public class PS006257 : Test
{
    private LogInPage login;
    private HomePage home;
    private ControlesPage controlesEnfermeria;
    private SelectResidentComponent seleccionarResidente;
    private DiuresisControlPage controlDiuresis;
    private ToastComponent toastComponent;

    protected override void SetUp()
    {
        this.login = new LogInPage(webDriver);
        this.home = new HomePage(webDriver);
        this.controlesEnfermeria = new ControlesPage(webDriver);
        this.seleccionarResidente = new SelectResidentComponent(webDriver);
        this.controlDiuresis = new DiuresisControlPage(webDriver);
        this.toastComponent = new ToastComponent(webDriver);

        restoreScript($@"{ Workspace.WORKSPACE_DIRECTORY }
   \ID29426\ID29426_DatosResidencia_01.sql", "DatosResidencia_001_01");
        restoreScript($@"{ Workspace.WORKSPACE_DIRECTORY }
   \ID29426\ID29426_DatosUsuarios.sql", "DatosUsuarios");
    }

    protected override void TestSteps()
    {
        // 1. Login con 'TestUser02'.
        login.NavigateToPage();
        info("Iniciando sesión como TestUser02");
        login.LogIn("TestUser02", "Selenium");

        // 2. Seleccionar el residente "Nuevo Residente 00001"
        seleccionarResidente.Select("Nuevo Residente 00001");

        // 3. Acceder a Controles -> Diuresis del 'Nuevo Residente 00001'.
        home.SelectOptionHome(HomePage.OpcionMenuPrincipal.Controles);
        controlesEnfermeria.SelectControl(
            ControlesPage.OpcionControles.Diuresis
        );

        // 4. Crear un nuevo registro de diuresis con fecha un año mayor a la
        //    actual.
        DiuresisRecord testDiuresisRecord = new DiuresisRecord(
            DateTime.Now.Add(TimeSpan.FromDays(365)), 23.4, 2, "Asadfasfa"
        );
        controlDiuresis.AddNewDiuresisRecord(testDiuresisRecord);

        // 5. Comprobar que ha aparecido el mensaje de alerta 'No es posible
        //    registrar controles a futuro.'.
        string toastMessage = toastComponent.GetToastMessage();
        assert("No es posible registrar controles a futuro." == toastMessage,
               "Comprobar mensaje de alerta del toast.");
```

```
        // 6. Comprobamos que no se ha creado ningún nuevo registro.
        DiuresisRecord readRecord = controlDiuresis.getDiuresisRecordAt(0);
        string dateFormat = "dd/MM/yyyy HH:mm";
        assert(testDiuresisRecord.date.ToString(dateFormat) !=
               readRecord.date.ToString(dateFormat),
               "Comprobamos que la fecha leída no coincide con la
                introducida");
        assert(testDiuresisRecord.cantidad != readRecord.cantidad,
               "Comprobamos que la cantidad leída no coincide con la
                introducida");
        assert(testDiuresisRecord.numPanales != readRecord.numPanales,
              "Comprobamos que el número de pañales no coincide con el
               introducido");
        assert(testDiuresisRecord.observaciones != readRecord.observaciones,
               "Comprobamos que las observaciones no coinciden con las
                introducidas");

        // 7. Cerrar la sesión.
        info("Cerrando sesión");
        login.LogOut();
    }

    protected override void Teardown()
    {
    }
}
```