UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de arquitectura genérica para pruebas automatizadas en un marco de desarrollo de software dirigido por modelos

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Alejandro Paños Medina

Tutor: Patricio Letelier Torres

2020-2021

Resumen

En el desarrollo de software, un buen sistema de pruebas es tan importante como el desarrollo del propio software. Mantener una batería de pruebas efectiva, fiable y actualizada no es una tarea fácil. Debido a la constante evolución del mundo de la industria del software nos encontramos con el problema que tendremos que elegir una tecnología concreta para llevar a cabo un proyecto y nunca se nos garantiza que esa tecnología posteriormente no cambie. Por multitud de causas nos podemos ver obligados a migrar de tecnología de desarrollo, por ejemplo, porque no se adapta a nuevas necesidades o simplemente porque deja de tener soporte, con todo el trabajo de migración que esto conlleva. La motivación de este TFG surgió en un contexto de desarrollo de software dirigido por modelos, basado en la generación automática de código a partir de modelos, utilizando la tecnología *DSL-Tools* de Microsoft. Esta tecnología permite diseñar un lenguaje específico de dominio y utilizarlo para generar modelos basados en ese lenguaje. Una vez creados esos modelos, pueden utilizarse para generar código automáticamente a partir de ellos. En este contexto había que establecer cómo realizar las pruebas del código generado de una forma que estuviese alineada con el enfoque de generación automática de código y que facilitara un posible cambio de tecnología de desarrollo del producto o la propia herramienta utilizada para dar soporte a las pruebas automatizadas. Así, en este TFG se aborda ese desafío diseñando e implementando una batería de pruebas que utiliza una estructura genérica. Esta puede reutilizarse, permitiendo cambiar de herramienta de pruebas sin la necesidad de reescribir las pruebas del producto desde cero, siendo solo sea necesario adaptar los mecanismos propios de la nueva herramienta a la estructura existente, ahorrando mucho tiempo y recursos.

Palabras clave: Pruebas automatizadas, pruebas de integración, Appium.

Abstract

Related to software development, a reliable test suite is as much important as the software itself. To maintain an effective, reliable and updated software testing suite is not an easy task. Due to the continuous evolution of the software industry, it's important to choose wisely which tool we are going to use in a project, and nothing guarantees us that is not going to change. There are a lot of reasons to force us to migrate the project to another tool because it does not adapt to new needs or simply because it has no longer support, with all the effort that this entails. This paper born in a model driven automated software development context, based in model-based automated code generation using the *Microsoft's DSL-Tools* technology. This technology allows to design a domain specific language and use it to generate models based on it. Once the models are created, they can be used to generate code automatically from them. Using this context, it was necessary to determinate how to perform the generated code tests in a way that was aligned with the automated generation code approach and facilitates a possible change in the development product technology or the tool used to support the automated tests. This paper tries to approach that challenge designing and implementing a tests suite that uses a generic structure. It can be reused allowing to change the testing tool without needing to rewrite every single test from scratch, only being needed to adapt the specific tool mechanisms to the existing structure, saving time and resources.

Keywords: Automated tests, integration tests, Appium.

Tabla de contenidos

Tabla de figuras

1. Introducción

1.1 Motivación

La informática y las tecnologías de la comunicación evolucionan cada vez más rápido para dar soluciones a nuevos problemas. Una de las principales formas de la informática para solucionar esos problemas es mediante aplicaciones software que ofrecen una forma de suplir esas necesidades. Esas necesidades van desde lo más cotidiano, como una aplicación donde subir fotografías, hasta grandes empresas que necesitan, de alguna forma, gestionar todos los aspectos del negocio, como es el caso de los ERP.

En ocasiones, estas aplicaciones llegan a alcanzar un tamaño tan considerable que su mantenimiento puede constituir un gran porcentaje del esfuerzo invertido en su desarrollo. Esto, junto a que el software es un mundo en constante evolución, y los cambios a los que constantemente se ve sometido un producto software durante su ciclo de vida, pone en evidencia la necesidad de utilizar mecanismos avanzados que mejoren la mantenibilidad.

La automatización de tareas asociadas al desarrollo de *software* proporciona una ayuda para aliviar la carga de trabajo que recae sobre los desarrolladores cuando el producto alcanza un tamaño importante. Es especialmente útil en las pruebas de regresión, por ejemplo, que se ejecutan repetidas veces de forma sistemática y no necesiten la intervención de una persona, dejando a los *testers* las pruebas de carácter más visual, subjetivo y enfocadas a la usabilidad.

Por otro lado, los desarrolladores de *software* utilizan herramientas de terceros para implementar funcionalidades ahorrando costes e intentando no reinventar la rueda. Sin embargo, esto provoca en parte cierta dependencia de los creadores por parte del usuario de esos productos, dejando al usuario de esa herramienta a merced de lo que ocurra con ella. Es frecuente que los desarrolladores dejen de dar soporte a ciertos productos cuando deciden que su ciclo de vida útil ha llegado a su fin. Y esto es en el mejor de los casos, ya que, en ocasiones, el proyecto simplemente se abandona o acaba decidiéndose que no resulta de utilidad, por lo que todo el trabajo de integración en nuestro producto se pierde.

Es por esto, que también es interesante planificar el desarrollo del software que utilice librerías o utilidades de terceros de forma que no suponga un impacto demasiado

grande el intercambiar unas herramientas por otras si se diese una de estas situaciones mencionadas en el párrafo anterior.

En este trabajo de final de grado se desarrolla una propuesta para dar solución a los problemas provocados por ese aspecto cambiante del software, proporcionando a una empresa una batería de pruebas automáticas de regresión con un diseño estructural genérico que mejore la mantenibilidad del software, y su adaptabilidad a nuevas herramientas de pruebas.

El problema que plantea el caso de estudio que se trata en este trabajo consta de unos requisitos muy concretos. La solución debe ser capaz de ofrecer una batería de pruebas que puedan ser integradas fácilmente en el ciclo de desarrollo de la aplicación, con una herramienta que ofrezca las mecánicas necesarias para adaptar la batería a los requisitos del equipo de desarrollo, que facilite su funcionamiento en diferentes plataformas y, por último, que en última instancia permita una relativa fácil migración de la batería a otra herramienta.

Dado que son unos requisitos numerosos y muy concretos, tras una búsqueda de soluciones utilizando una serie de palabras clave como pueden ser “batería de pruebas automatizadas genérica”, u otras relacionadas con la fácil migración de una batería, ha sido imposible encontrar alguna solución ya existente. Por ello, se ha decidido prescindir de un capítulo dedicado al Estado del Arte.

Por lo tanto, finalmente se optó por desarrollar la propuesta que se plantea en este trabajo para proporcionar una solución que los cumpliese todos.

La idea de este proyecto nació en la empresa en la que realicé las prácticas del grado donde se está desarrollando una nueva versión de un ERP del sector sociosanitario. La aplicación está desarrollada por componentes y cada vez que se lanzaba una nueva versión del software no tenían forma de probar que la versión de los distintos componentes era la correcta hasta que esta estaba desplegada, y las pruebas se hacían manualmente. Por esto, se propuso una solución basada en una batería automatizada, en la que se realizasen pruebas funcionales de integración que aseguren la interconexión de todos los bloques y evitase ese trabajo manual constante y repetitivo al equipo de desarrollo.

El principal inconveniente a la hora de desarrollar cualquier tipo de software de escritorio es la compatibilidad con los distintos entornos de ejecución. A diferencia de las aplicaciones web, donde cualquier aplicación se ejecuta en el mismo entorno independientemente de qué navegador o sistema operativo se utilice, el software de

escritorio desarrollado para una plataforma concreta sólo puede ejecutarse en esa plataforma.

Para los escenarios en los que se necesite ejecutar una batería de pruebas para distintas plataformas, existen herramientas *cross-platform,* como *Appium* [1] , que permiten, dado el caso, su ejecución en diferentes sistemas operativos.

Por otro lado, otra de las preocupaciones que tenía la empresa era la correcta elección de la herramienta utilizada para las pruebas. Es frecuente que se lleve a cabo el desarrollo de un producto software basado en una herramienta concreta y eventualmente ésta deje de ser una opción inteligente, ya sea porque los creadores o su comunidad deja de ofrecer soporte, o simplemente porque no cubre las necesidades del producto. Llegado el caso, esto plantea un problema, y es la necesidad de reescribir o rehacer todo el proyecto adaptado a una nueva herramienta.

Por ello, se propuso una estructura de las pruebas con un carácter genérico, que permitiese la migración de las pruebas a otras herramientas con el mínimo esfuerzo posible, mejorando aún más esa capacidad adaptativa del proyecto.

En este trabajo se utilizará Windows 10 como entorno de ejecución y Appium + *WinAppDriver* [2] como herramienta principal para las pruebas. Hablaremos de ella y de otras, así como de la estructura utilizada con más detalle más adelante.

1.2 Objetivos

El principal objetivo de este trabajo es la implantación de una batería automática de pruebas de integración que proporcione una solución a los problemas existentes de versionado en el ciclo de desarrollo de un software ERP diseñado para empresas del sector socio-sanitario, cumpliendo con algunos requisitos que se explican a continuación.

Estos requisitos comienzan con la elección de un diseño de la solución que permita su fácil evolución, mantenimiento y adaptación a importantes cambios, como puede ser el cambio de la herramienta de pruebas sin que suponga una inversión de tiempo y recursos prácticamente idéntica a la ya invertida en su desarrollo inicial.

Otro requisito importante es el uso de una herramienta de pruebas de tipo *scripting*, que permitiese en un futuro la generación de las pruebas mediante las herramientas de

[1] https://appium.io/
[2] https://github.com/microsoft/WinAppDriver

lenguaje de dominio específico que utiliza la empresa para generar de forma automática gran parte del código fuente que conforma la aplicación.

Este ERP es multiplataforma, con versión para ordenadores de sobremesa, *tablets* y *smartphones*, por lo que es necesario que la batería de pruebas sea capaz de ejecutarse en distintos entornos con distintos sistemas operativos, evitando tener que reescribir la misma solución con distintas herramientas.

1.3 Estructura de trabajo

En el capítulo 2 – *Pruebas funcionales,* se comentará brevemente sobre las pruebas de software y los tipos en los que se dividen, así como su importancia dentro del ciclo de desarrollo de un producto software.

En el capítulo 3 – *Herramientas para automatización de pruebas,* se presentarán las diferencias entre pruebas manuales y automáticas, así como los criterios a seguir a la hora de decidir qué parte de las pruebas de software deberían automatizarse. También se introducirán algunas de las principales herramientas de pruebas existentes en el mercado tenidas en cuenta para este proyecto.

En el capítulo 4 – *Propuesta de organización de las pruebas,* se presenta el diseño de la estructura genérica para la batería de pruebas automáticas desarrollada, y se analiza su funcionamiento para comprender cómo consigue cumplir los objetivos.

En el capítulo 5 – *Migración de la batería de pruebas,* se explica cómo la estructura diseñada permite la migración de la solución a otra herramienta de tipo *scripting* reduciendo de forma considerable el tiempo y recursos invertidos.

En el capítulo 6 – *Metodología,* se muestra la metodología seguida en el desarrollo del trabajo, así como un cronograma con los principales hitos del proceso.

En el capítulo 7 – *Conclusiones y trabajos futuros,* se comentan las conclusiones finales, así como posibles mejoras o ampliaciones a realizar en la propuesta, y las asignaturas que han aportado conocimientos de mayor peso a la hora de realizar este trabajo.

2. Pruebas funcionales

2.1 Introducción a las pruebas del software

El desarrollo de un producto software y su capacidad para desempeñar la función para la que fue ideado depende de muchos factores, como pueden ser, el factor humano, los constantes cambios a los que es sometido, problemas de comunicación entre el equipo de desarrollo y el cliente que inducen a malentendidos entre ambas partes, requisitos del entorno de ejecución no cubiertos, etc. Tantos factores introducen una gran cantidad de problemas potenciales en el proceso de desarrollo que harían inviable conseguir un producto de calidad. Los distintos tipos de pruebas de software permiten asegurar cierto nivel de calidad y robustez conforme avanza el desarrollo [1].

Existen varios tipos de pruebas para software que se enfocan en probar distintos aspectos del producto y se dividen en funcionales y no funcionales. Las pruebas funcionales se centran en probar que el producto cumple con las especificaciones, y las no funcionales se centran en aspectos alternativos que, si bien no afectan al cumplimiento de las funciones requeridas, sí afectan a aspectos como la seguridad, mantenibilidad, rendimiento o accesibilidad, entre otros. Introduciremos brevemente los principales tipos de pruebas funcionales de software.

Dado que un proyecto software, sobre todo si es de cierta envergadura, es un proyecto en el que participan varios programadores, es de imperiosa necesidad compartimentar el desarrollo en bloques. Las pruebas unitarias permiten verificar el comportamiento y la ausencia de fallos de cada bloque de código sometiéndolo a diferentes situaciones y casos extremos donde es más probable que la lógica falle más allá, preferiblemente, de los típicos *happy paths* [3] o caminos felices.

Es considerada una buena práctica desarrollar las pruebas a la vez que el código fuente de la aplicación. Esto permite una mayor calidad del código y que ésta se mantenga a lo largo de todo el proceso. Existe una técnica llamada *Test Driven Development (TDD)* [2] o desarrollo guiado por pruebas, donde se desarrollan previamente las pruebas a las que será sometido el software. El funcionamiento consiste en desarrollar una prueba en

[3] https://www.rafablanes.com/blog/el-happy-path-o-las-pruebas-felices

base a un requisito y posteriormente se implementa el código que sería capaz de satisfacerla.

Como hemos mencionado antes, el *software* es desarrollado por bloques. Estos bloques, una vez validado su funcionamiento individual, pueden comunicarse entre ellos para formar una única funcionalidad y debe poderse asegurar el su funcionamiento. Las pruebas de integración permiten validar esa interconexión, que es especialmente sensible de fallos en equipos de desarrollo grandes en los que cada componente es implementado por personas distintas.

Las pruebas de integración son naturalmente más complejas de llevar a cabo porque involucran mayor cantidad de código que las unitarias.

A la hora de integrar componentes puede hacerse de manera incremental o no incremental, en función de si se va integrando nuevos componentes con otros ya probados o si, por el contrario, se prueban los componentes por separado y se integran todos cuando se hayan realizado las pruebas individuales.

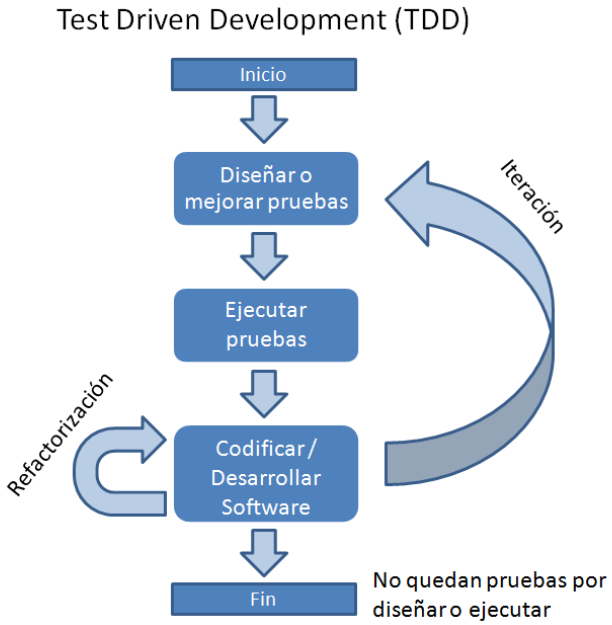

Figura 1: Ciclo de vida del desarrollo de software guiado por pruebas [3].

En la figura 1 puede verse el ciclo de desarrollo seguido en el desarrollo TDD. Se divide en pequeñas iteraciones en las que primero se planifican las pruebas que el producto deberá cumplir al final de esa iteración, y posteriormente se modifica el proyecto para

satisfacerlas. Se realizarán refactorizaciones hasta que las pruebas se ejecuten sin problemas.

Cuando el sistema completo está probado a nivel unitario y de integración, se dice que la integración del sistema está lista y se puede empezar a probar los casos de uso definidos.

Un caso de prueba satisface los requisitos que establece un caso de uso para cumplir un objetivo, y un caso de uso se define como un conjunto de pasos y funciones individuales que forman una única funcionalidad concreta. Una prueba de aceptación define una de esas funcionalidades concretas respecto a unos requisitos establecidos.

En base a esto, primero se definen una serie de Pruebas de Aceptación (PA) que cumplan los requisitos del cliente. Los casos de uso serán los encargados de establecer los pasos a seguir para cumplir una PA y describen distintos caminos que el usuario puede seguir para llegar a un mismo objetivo. Un caso de prueba define una configuración concreta que pruebe los requisitos de un caso de uso. Es lógico pensar, que un caso de uso puede tener varios casos de prueba asociados que verifiquen esa funcionalidad con diferentes configuraciones de los datos de entrada.

Las pruebas de aceptación pueden ser funcionales o no funcionales. En este trabajo nos centraremos en la automatización de pruebas funcionales para aplicaciones de escritorio.

2.2 ¿Pruebas manuales o pruebas automatizadas?

La automatización de pruebas de software surge como un apoyo al proceso de desarrollo de un software para complementar las pruebas manuales y no para sustituirlas. Las pruebas automatizadas tienen muchas ventajas y son de utilidad, pero no proporcionan un método de prueba fiable por sí solas. De hecho, un caso de prueba primeramente es manual, y una vez verificado pasa a automatizarse.

En la práctica, las pruebas manuales están enfocadas a unos tipos de pruebas, y las automáticas a otros. Las manuales, son adecuadas para pruebas de tipo exploratoria o de usabilidad, debido al factor humano que no tienen las pruebas automáticas. Estas últimas, están más aconsejadas las pruebas de regresión, carga, rendimiento, etc. que no requieren de improvisación a la hora de ejecutarlas. Sin embargo, existen pruebas que pueden realizarse tanto de forma automática como manual, como son las pruebas de integración o de sistema.

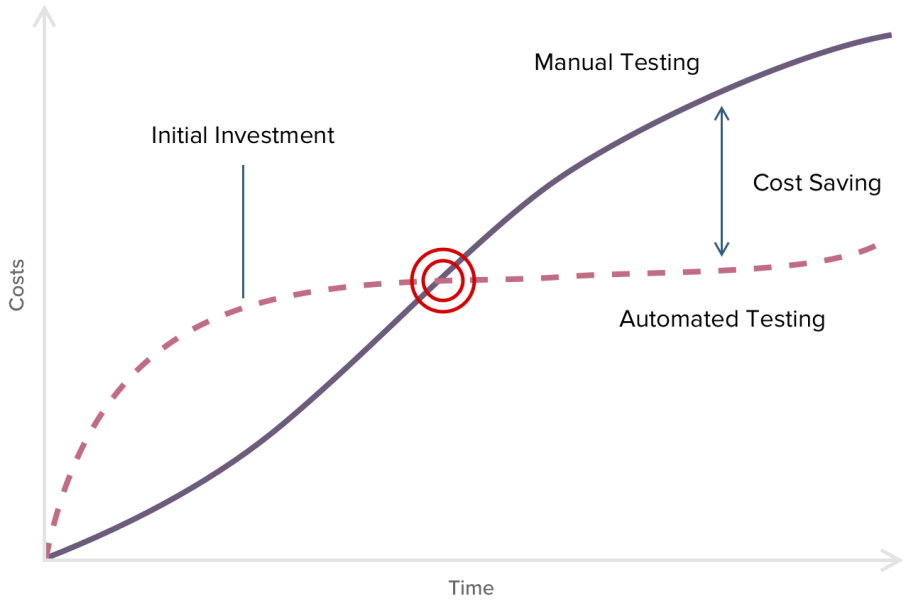

Figura 2: Relación coste-tiempo de las pruebas manuales respecto a las automáticas [3].

Las pruebas automatizadas son frecuentemente utilizadas para Pruebas de Regresión al final de cada ciclo de desarrollo, o versión, y minimizar el riesgo de que errores introducidos en él salgan al entorno de producción. Esto alivia de forma considerable la carga que soportan los *testers*, evitando que dediquen tiempo a probar partes del producto ya probadas anteriormente.

También ayudan a detectar errores colaterales por cambios en el código que a priori no han sido modificados y que por lo tanto pueden no haberse incluido en la lista de funciones que los *testers* han de revisar. Con una batería de pruebas automatizadas que se lancen al final de cada *sprint* se asegura que, aquello ya probado, sigue funcionando correctamente.

Las pruebas automatizadas permiten también aumentar la complejidad y la cobertura de las pruebas, ya que pueden ser ejecutadas infinidad de veces con distintos datos de entrada, permitiendo cubrir un mayor abanico de posibilidades y logrando mayor fiabilidad. Esto no sería posible si tuviesen que realizarlas el equipo de *testers* por el obvio coste de esfuerzo y tiempo. Además, elimina el factor error humano de la ejecución de las pruebas [4].

Son algunas de las ventajas que tienen las pruebas automatizadas, entre otras muchas, pero, como todo, no sólo tiene ventajas.

Mantener una batería de pruebas automatizadas exige también tiempo y esfuerzo. Cada cambio en el producto que afecte a alguno de los componentes probados de forma automática se traduce en la obligación de adaptar la prueba al cambio o esa prueba quedará obsoleta e inútil.

Implica también que el personal necesite conocimientos más avanzados para el desarrollo y mantenimiento de los casos de prueba que se automaticen, lo que se traduce en mayor costo en formación. Este problema puede ser precursor de otro que consiste en el mal diseño de un plan de pruebas automatizadas, lo que supondría un pozo de recursos dedicados al mantenimiento y desarrollo del proyecto que podrían dedicarse a otros aspectos del desarrollo, lastrando así todo el proceso en vez de acelerarlo [3].

En la figura 2 se muestra una gráfica comparativa coste-tiempo entre las pruebas manuales y las automatizadas. Se puede observar el mayor coste inicial de automatizar las pruebas respecto a realizarlas de forma manual, sin embargo, conforme crece la complejidad, llega un punto en el que las pruebas automatizadas ganan rentabilidad respecto a las manuales.

En la siguiente tabla se comparan algunas de las principales ventajas y desventajas de las pruebas manuales respecto a las automatizadas. En términos generales, es evidente que la automatización de pruebas proporciona clara ventaja en aspectos como la fiabilidad y escalabilidad de la batería, y, por el contrario, las pruebas manuales son adecuadas para proyectos pequeños y con pocos casos de prueba, o casos en los que la subjetividad humana sea necesaria.

	Manual	Automatizado
PROS	• Cualquiera puede hacer pruebas. • Más fácil mejorar la calidad. • Adecuado para personas sin formación o experiencia. • Ideal en situaciones en las que las pruebas se ejecutan una sola vez y no de forma repetida. • Adecuado para pruebas enfocadas en la experiencia de usuario o interfaces amigables.	• Mayor fiabilidad y precisión debido a que se realiza mediante código. • Significativamente más rápido. • Ideal para casos en los que las pruebas se ejecutan repetidas veces. • Ahorro económico para las empresas una vez implementado. • Identifica mayor cantidad de errores en menos tiempo.
CONS	• Es posible que no se logre identificar todos los casos de prueba. • Peor calidad de las pruebas debido al alto nivel de errores. • Consume más tiempo. • Se requiere inversión en personal con cualificación. • Mayor número de errores debido al factor humano.	• Se requiere inversión inicial en una herramienta de pruebas. • Se requiere personal con habilidades de programación.

Tabla 1: Tabla comparativa entre testing manual y automatizado [3].

2.3 ¿Cómo elegir qué pruebas automatizar?

Cuando hablamos de cómo elegir qué automatizar y qué no, lo mejor es referenciar la *Pirámide de Cohn*, de la cual se habla en el libro *"Succeeding with Agile" [5].* En él, su autor, *Mike Cohn,* expone la estrategia para realizar pruebas software en metodologías ágiles más utilizada en la actualidad.

Según la *Pirámide de Cohn,* ilustrada en la siguiente figura, la mayor parte de la automatización debe enfocarse en las pruebas de menor nivel, como son, por orden creciente, las pruebas unitarias, de componentes, de integración y de API, y en las que se debería poner menor esfuerzo es en las pruebas de mayor nivel como son las de interfaz.

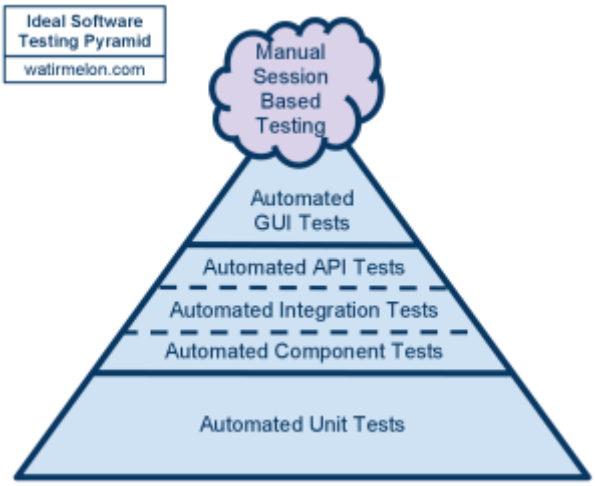

Figura 3: Pirámide de Cohn. Estructuración ideal de las pruebas de software [5].

En un caso ideal, la estructura sería:

- Muchos *test* unitarios automáticos. Esto detecta fallos a nivel de código, y evita que errores a nivel desarrollador se propaguen hacia la parte superior de la pirámide.
- Bastantes *tests* a nivel de componente, integración y de API. Son las mejores candidatas a automatizar, debido a que son las que menos varían durante el proceso de desarrollo.
- Pocos tests de interfaz. Debido a la naturaleza cambiante de la interfaz del producto, resulta más costoso automatizar y mantener pruebas en este apartado que probarlas de forma manual.

Como hemos comentado, esta estructura evita que errores de bajo nivel, afecten a áreas de mayor nivel. Además, automatizar *tests* unitarios documenta el producto, y nos ayuda a detectar qué es lo que falla y dónde se encuentra el fallo [6].

Precisamente, en el caso de estudio de este trabajo, la empresa dispone de una gran cobertura de *testing* automatizado a nivel unitario, y la batería de pruebas automatizadas a nivel de integración e interfaz es muy reducida, limitándose a los casos de prueba mínimos esenciales que no pueden ser cubiertos con las unitarias.

3. Herramientas para automatización de pruebas

En este capítulo se expondrán las principales herramientas que se tuvieron en cuenta para automatizar pruebas de aceptación, así como algunas otras opciones y el por qué se descartaron.

3.1.1 TestArchitect

TestArchitect [4]es una herramienta de tipo *Action-Based-Keyword-Driven* para el diseño, ejecución y seguimiento de pruebas automatizadas. Las pruebas son definidas mediante acciones (*Action-Based*) que, junto a una interfaz amigable, como se muestra en la siguiente figura, y un sistema integrado de *record-playback*, ofrece un mayor nivel de abstracción que un *framework* de tipo *scripting* [8].

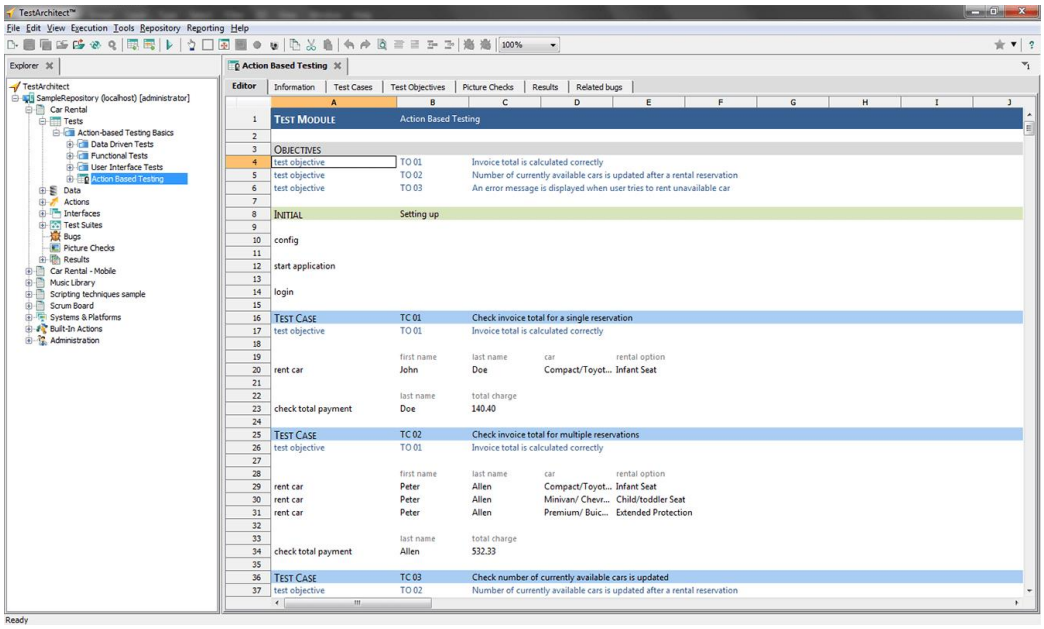

Figura 4: Interfaz de TestArchitect [7].

[4] https://www.testarchitect.com/

Con herramientas como *TestArchitect* podemos diseñar pruebas mediante comandos o acciones cercanas al lenguaje humano, permitiendo el desarrollo de las mismas a personas sin excesivos conocimientos de programación. Incluso algunas, como es el caso de *TestArchitect,* incluyen la posibilidad de implementar las pruebas grabando acciones que realicemos nosotros mismos. De este modo, cuando ejecutemos las pruebas, la herramienta reproducirá las acciones que hayamos grabado. Es una forma más dinámica y con un mayor nivel de usabilidad para hacer las pruebas.

A la hora de desarrollar las pruebas de interfaz, primero se deberá crear un nuevo proyecto y un módulo de pruebas en el que organizaremos las distintas pruebas que implementemos. Una vez la estructura esté lista, tendremos que crear el primer caso de prueba en el apartado *Tests* como puede verse en la figura 4.

Como se muestra en la figura 5, dentro del apartado de la nueva prueba, existen las secciones *Objectives, Initial, Test Case* y *Final* en las que podremos configurar la estructura general de una prueba de software, que se compone de los objetivos de la prueba, la preparación inicial para la ejecución, el caso de prueba concreto, y la preparación o *reset* final, respectivamente.

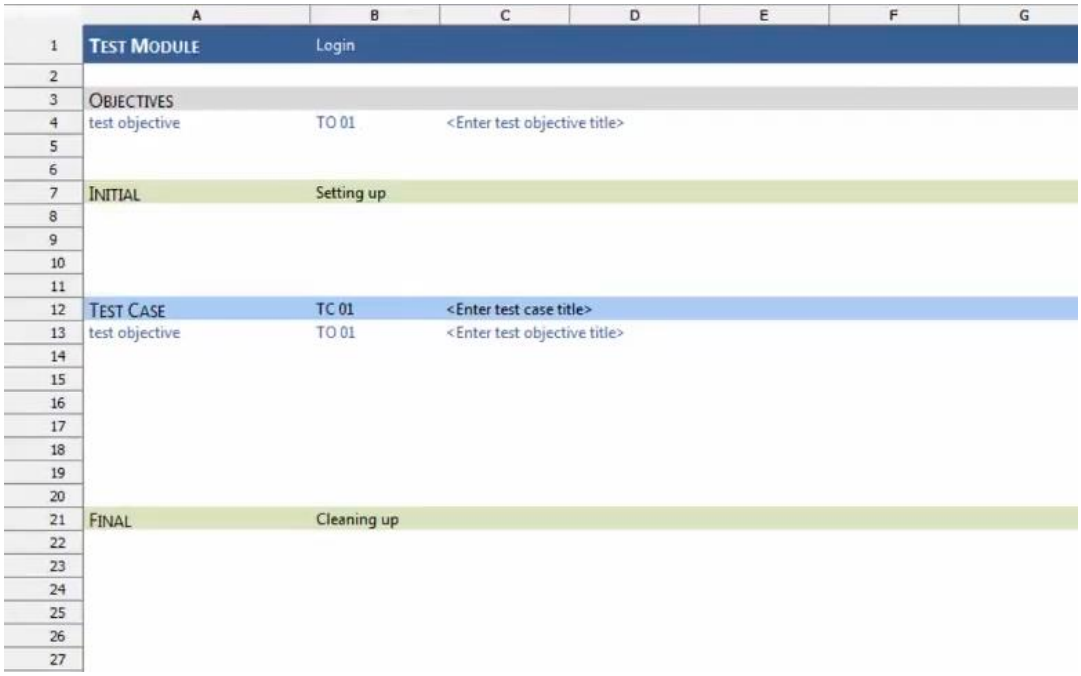

Figura 5: Secciones del caso de prueba en TestArchitect. [7].

Para llevar a cabo las acciones necesarias en cada prueba, deberemos mapear aquellos elementos de interfaz que se utilicen en el caso de prueba. Para ello, *TestArchitect* permite la búsqueda de elementos mediante *XPaths*, etiquetas *CSS*, *ID*, Nombre, etc, entre otros. *TestArchitect* incluye un explorador de interfaz, en el que podemos navegar por los distintos elementos y ver toda la información que permite describir y localizar esos elementos. De esta forma, podemos crear un *Interface Entity* en el que agrupar los elementos de interfaz que vayamos a usar en una prueba determinada, y utilizarla para crear las acciones que vayan a formarla, como se muestra en la figura 6.

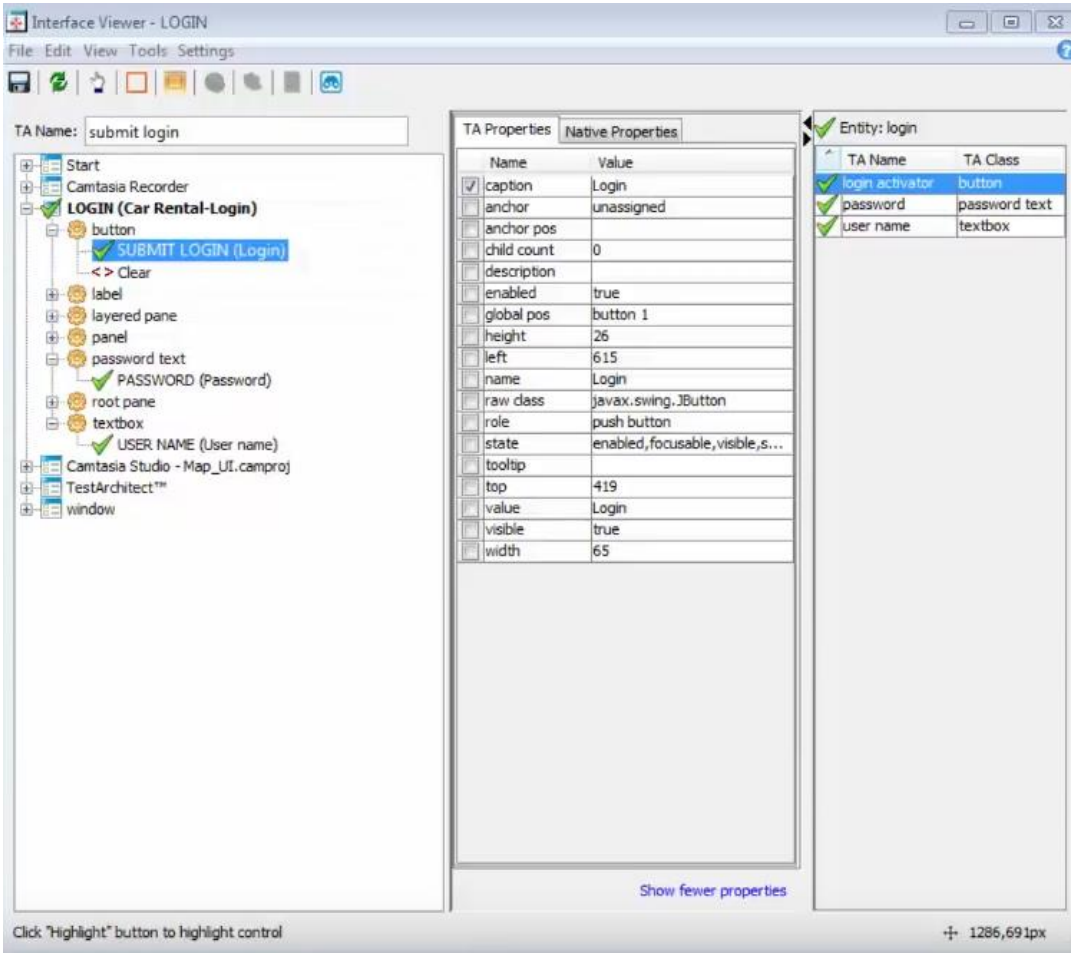

Figura 6: Explorador de interfaz de TestArchitect [7].

En el explorador de interfaz podremos marcar las características que se utilizarán para localizar cada elemento. En el ejemplo mostrado en la figura 7, podemos ver como el cuadro de texto en el que se introduce el nombre de usuario para el *login*, por ejemplo, será de tipo *textbox* y un *label* con el valor *User name*.

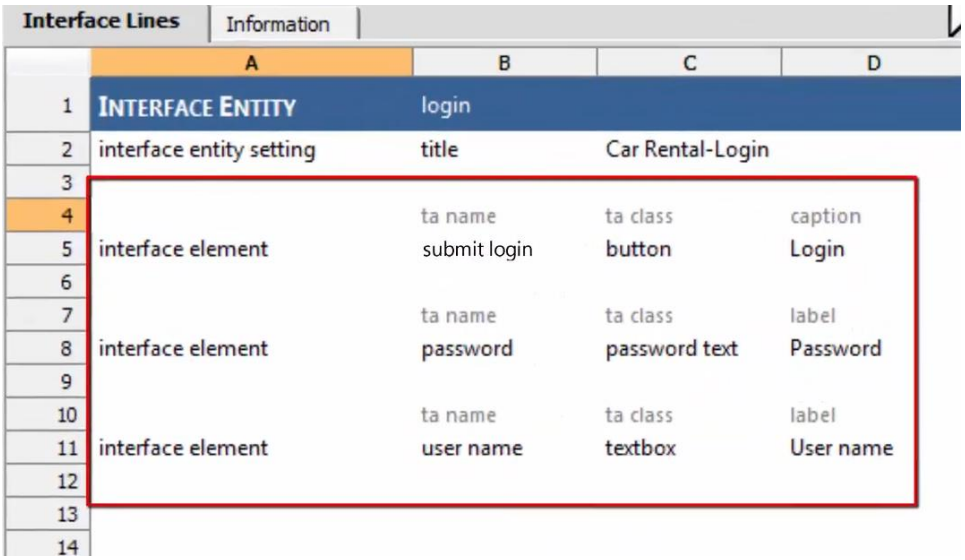

Figura 7: Elementos de interfaz mapeados en TestArchitect [7].

Una vez se ejecutan las pruebas, la herramienta ofrece un visor de resultados en el que se detallan los pasos realizados, junto con capturas de pantalla de cada paso para facilitar el análisis de los resultados, y, si fuese el caso, el motivo de que la prueba haya fallado.

TestArchitect ofrece más herramientas adicionales como la depuración de pruebas, una herramienta *record-playback*, un sistema de importación y exportación de pruebas, un sistema propio integrado de control de versiones o herramientas para utilizar *Data-Driven Testing*[5], que ayudan a completar el proceso de probar un producto software, pero que no entraremos a detallar en este trabajo.

Sí es interesante comentar los puntos negativos y/o los motivos por los que no se utilizó finalmente *TestArchitect* como herramienta para nuestro caso de estudio.

La imposibilidad de integrar las pruebas en el ciclo de desarrollo principal del producto a probar, debido a que no es un *framework* de tipo scripting, fue uno de los principales motivos para descartarlo como herramienta. La idea era poder generar de forma automática las pruebas usando *DSL-Tools* que utiliza la empresa para generar la mayoría de código, y que éste cambiase y se adaptase a los cambios de interfaz de forma automática. Además de la poca capacidad de personalización de las pruebas respecto a una herramienta de tipo *scripting* que ya hemos comentado anteriormente.

Por otro lado, no se puede utilizar Git como sistema de control de versiones ya que *TestArchitect* tiene el suyo propio integrado (*built-in)*.

[5] https://www.softwaretestinghelp.com/data-driven-testing/

Aparte del *ABT (Action-Based-Testing*[6]*),* solo se puede utilizar *Java, Python y C#* como lenguajes para escribir las acciones, lo que podría ser un problema si la aplicación multiplataforma tuviese alguna versión escrita en otros lenguajes [7].

En la figura 8 se muestra un ejemplo de una prueba de *login* en *TestArchitect.*

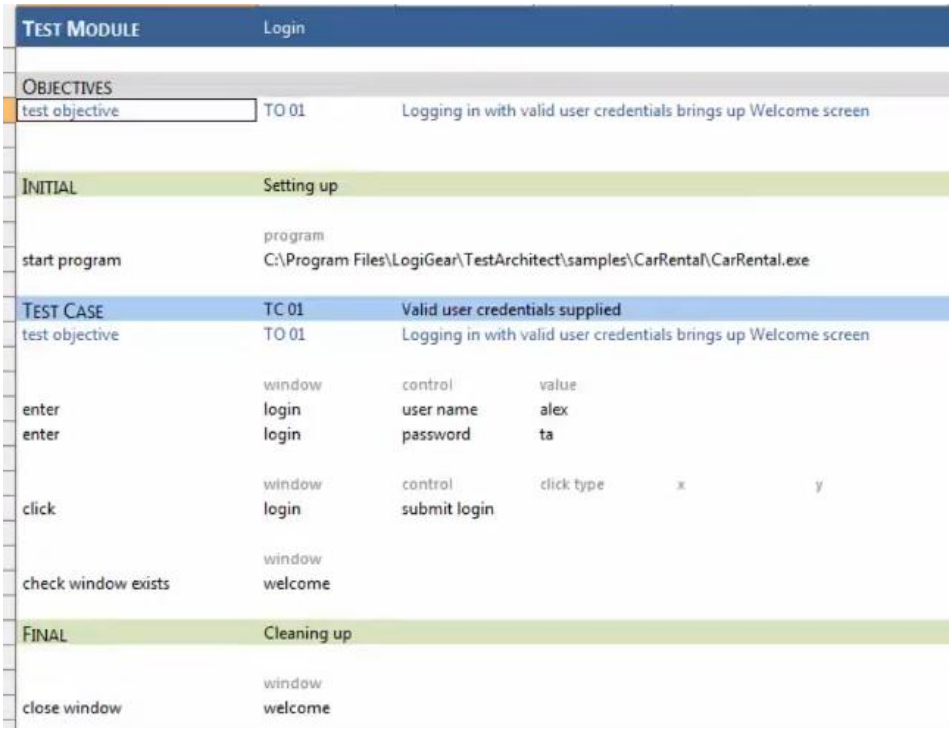

Figura 8: Prueba de login de una aplicación ejemplo en TestArchitect [7].

3.1.2 Winium

Winium [7] es un *framework* gratuito de código abierto basado en Selenium para probar aplicaciones nativas de Windows. Funciona con aplicaciones de Windows (WPF y WinForms), aplicaciones de Windows Store para Windows Phone y con aplicaciones *Windows Phone Silverlight*. Winium ofrece distintas APIs para utilizar según se quiera probar un tipo de aplicación u otro [8].

Para aplicaciones de escritorio de *Windows*, por ejemplo, se deberá utilizar *Winium* Desktop Driver, junto a Winium Cruciatus, que es un *wrapper* de la librería Microsoft UI

[6] https://www.testarchitect.com/support/action-based-testing/abt-overview
[7] https://github.com/2gis/Winium

Automation para *Visual Studio* y permite la implementación de las pruebas en cualquier *framework* de pruebas, como por ejemplo NUnit.

Entre las principales ventajas de Winium, está la posibilidad de escribir las pruebas en casi cualquier lenguaje (*Java, JavaScript, PHP, Python, C#, Ruby, etc*), el uso de cualquier entorno de pruebas, como Junit, *TestNG*[8] o *PyUnit*[9], y que al estar basado en *WebDriver* puede ser familiar para usuarios que ya tengan experiencia con *Selenium* y *Appium*.

Al igual que *TestArchitect* o *Selenium*, utiliza *XPath* [10]para localizar los elementos de la interfaz de las aplicaciones de Windows. No tiene un explorador de elementos como *TestArchitect,* por lo que se deberá utilizar otras opciones, como el Inspector de Windows, para conocer la información necesaria de cada elemento para formar los *XPaths*.

En la figura 9 se muestra un esquema del flujo de trabajo que realiza *Winnium* a la hora de ejecutar las pruebas y el uso del protocolo *JsonWireProtocol*.

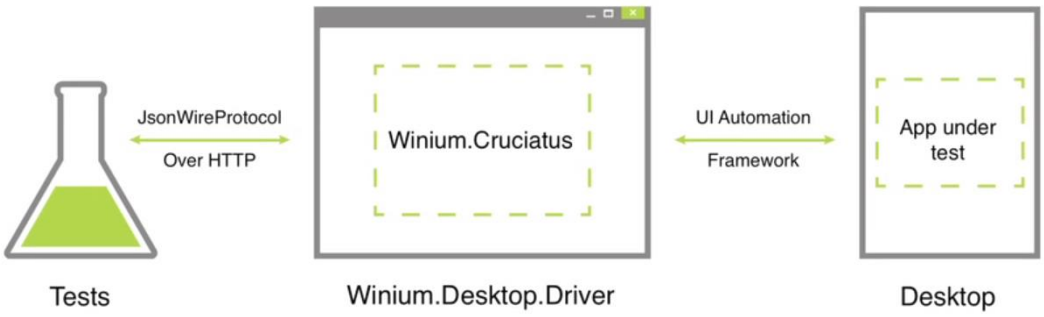

Figura 9: Flujo de trabajo de Winnium para aplicaciones de escritorio de Windows [9].

Winium Desktop Driver implementa el Selenium Remote WebDriver, ya que, como su sucesor WinAppDriver, está basado en WebDriver, y recibe los comandos JsonWireProtocol. Es el responsable de la automatización de la aplicación utilizando Winium Cruciatus.

[8] https://testng.org/doc/
[9] https://wiki.python.org/moin/PyUnit
[10] https://www.w3schools.com/xml/xpath_intro.asp

Al igual que su sucesor, WinAppDriver, Winium sólo puede automatizar aplicaciones basadas en Windows y sólo funciona en Windows 10. Además, al ser una herramienta exclusivamente de tipo *scripting*, no dispone de utilidades como el *record-playback,* gestión de las pruebas y resultados integrado, o control de versiones propio.

Winium dejó de tener soporte en el año 2016, lo que supuso un motivo clave para descartarlo como herramienta para el caso de estudio de este trabajo.

Para empezar a crear pruebas con Winium se configuran las *desired capabilities* para la aplicación a probar en el bloque *SetUp*. En ellas se configuran algunos aspectos como el ejecutable de la aplicación, el puerto de escucha (ya que está basado en WebDriver) o el tiempo de espera en el lanzamiento de la aplicación.

En la figura 10 se muestra un ejemplo de cómo configurar *Winnium.*

```
[SetUp]
public void SetUp()
{
    var dc = new DesiredCapabilities();
    dc.SetCapability("app", Path.Combine(Environment.CurrentDirectory, "WpfTestApplication.exe"));
    dc.SetCapability("launchDelay", 2);
    this.Driver = Activator.CreateInstance(typeof(TDriver), new Uri("http://localhost:9999"), dc) as TDriver;
}
```

Figura 10: Ejemplo de configuración inicial de Winium [10].

Para las pruebas, el procedimiento de trabajo sería encadenar búsqueda de elementos, mediante la obtención de información de sus metadatos, como hablamos anteriormente, para realizar acciones sobre ellos y luego comprobar un resultado esperado determinado. También pueden probarse otros aspectos como el rendimiento, contabilizando el tiempo que tarda la aplicación en arrancar o realizar la carga de algún dato, por ejemplo.

En el fragmento de código de la figura 11 se muestra un ejemplo en el que se controla la aplicación de calculadora de Windows. Pueden verse funciones como *FindElement* o *FindElementByUid* que sirven para localizar elementos de interfaz como en este caso la barra de menú de la aplicación. Con funciones como *Click* o *SelectItem* podemos ejecutar acciones sobre los elementos, en este caso se utilizan para pulsar los botones de los números de la calculadora y el método *Close* para cerrar la aplicación.

```
var calc = new Winium.Cruciatus.Application("C:/windows/system32/calc.exe");
calc.Start();

var winFinder = By.Name("Calculator").AndType(ControlType.Window);
var win = Winium.Cruciatus.CruciatusFactory.Root.FindElement(winFinder);
var menu = win.FindElementByUid("MenuBar").ToMenu();

menu.SelectItem("View$Scientific");
menu.SelectItem("View$History");

win.FindElementByUid("132").Click(); // 2
win.FindElementByUid("93").Click(); // +
win.FindElementByUid("134").Click(); // 4
win.FindElementByUid("97").Click(); // ^
win.FindElementByUid("138").Click(); // 8
win.FindElementByUid("121").Click(); // =

calc.Close();
```

Figura 11: Prueba de ejemplo con Winium sobre la calculadora de Windows [10].

Winium es muy parecido a WinAppDriver en cuanto a cómo interactuar con la aplicación, sin embargo, como veremos más adelante, WinAppDriver es mucho más completo e intuitivo que Winium.

3.1.3 Appium y WinAppDriver

Appium [11] es una herramienta de automatización de pruebas multiplataforma de código abierto para aplicaciones web y para aplicaciones de escritorio nativas, híbridas y móviles. Al ser *cross-platform,* Appium permite crear pruebas en diversas plataformas utilizando la misma API, lo que se traduce en la reutilización de código entre conjuntos de casos de prueba definidos para una aplicación desarrollada para varias plataformas [8].

WinAppDriver [12], framework de código abierto desarrollado por Microsoft, es una implementación de Appium y permite la automatización de pruebas para aplicaciones de escritorio de Windows.

[11] http://appium.io/
[12] https://github.com/microsoft/WinAppDriver

Tanto Appium como WinAppDriver utilizan el *WebDriver* de Selenium, que es un entorno de pruebas para aplicaciones web y que usa un protocolo clientes-servidor conocido como JSON Wire Protocol. Por lo tanto, WinAppDriver es un framework de automatización similar a Selenium. Al igual que Selenium, WinAppDriver es un conjunto de bibliotecas que se pueden integrar en cualquier Test Runner que admita Appium, como por ejemplo en MSTest[13] de Visual Studio.

Según la filosofía de Appium, este está diseñado para satisfacer las necesidades de automatización de acuerdo a los siguientes principios [11]:

- No se debería tener que volver a compilar la aplicación en pruebas para automatizarla.
- No se debería limitar el desarrollo a un lenguaje o marco específico para escribir y ejecutar las pruebas.
- Un framework de automatización no debería reinventar la rueda cuando se trata de una API de automatización.
- Un framework de automatización debe ser de código abierto, tanto en espíritu, en práctica y en nombre.

Appium cumple con el primer requisito utilizando frameworks de automatización proporcionados por el proveedor bajo el capó. De esa forma, no se necesita compilar ningún framework específico de Appium o de terceros en la aplicación. Esto significa que se está probando la misma aplicación sin modificaciones. Los frameworks propietarios que utiliza Appium son XCUITest[14] y UIAutomation[15] para aplicaciones en IOS, UIAutomator y UIAutomator2 para aplicaciones en Android, y el que interesa para el caso de estudio de este trabajo, WinAppDriver para aplicaciones de Windows.

El segundo requisito se cumple envolviendo el framework propietario en una API, la API WebDriver. WebDriver especifica el protocolo cliente-servidor mencionado anteriormente. Dada esta arquitectura cliente-servidor, un cliente escrito en cualquier lenguaje puede ser usado para enviar las peticiones HTTP pertinentes al servidor. Esto también significa que se puede utilizar cualquier *test runner* y *test framework* que se quiera. Las librerías cliente son simplemente clientes HTTP que pueden ser mezclados en el propio código de la forma que se necesite. En otras palabras, los clientes Appium y WebDriver no son técnicamente frameworks de pruebas, sino más bien librerías de

[13] https://docs.microsoft.com/es-es/dotnet/core/testing/unit-testing-with-mstest
[14] https://appium.io/docs/en/drivers/ios-xcuitest/
[15] https://docs.microsoft.com/es-es/dotnet/framework/ui-automation/ui-automation-overview

automatización, de forma que se puede gestionar el entorno de pruebas de la forma que mejor se adapte a nuestros requisitos.

El tercer requisito se cumple mediante WebDriver. Este se ha convertido en el estándar para la automatización de navegadores web. No tiene sentido rehacer algo que ya existe y funciona bien. En su lugar, ampliaron el protocolo con métodos API adicionales útiles para la automatización móvil.

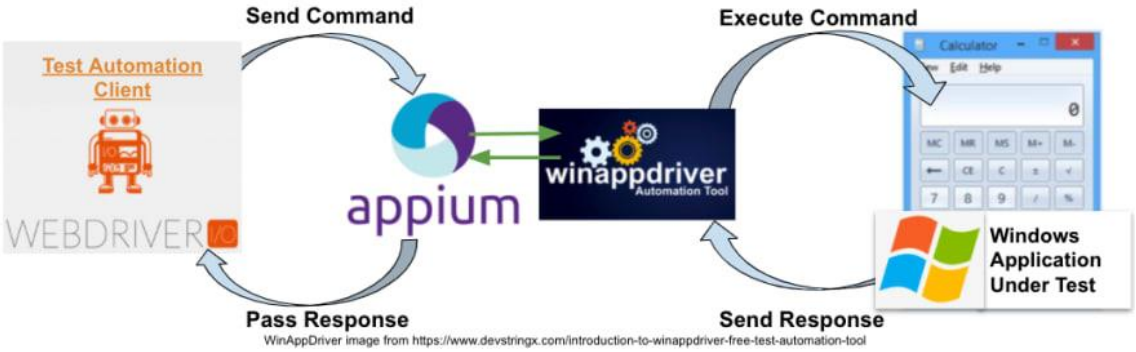

Figura 12: Esquema de comunicaciones entre WebDriver, Appium y WinAppDriver [12].

En la figura 12 se esquematiza el flujo de comunicaciones que hay entre *WebDriver*, *Appium* y *WinAppDriver.*

Finalmente, el cuarto requisito se cumple por razones obvias, Appium es código abierto.

Appium, en esencia, es un servidor web que expone una API REST. Recibe conexiones desde un cliente, lee los comandos, los ejecuta en un dispositivo y responde con una respuesta HTTP representando los resultados de la ejecución de los comandos. Esta estructura abre un mundo de posibilidades ya que permite escribir las pruebas en cualquier lenguaje que tenga una API cliente HTTP, aunque resulta más fácil utilizar una de las librerías cliente de Appium.

En el caso de las aplicaciones de Windows, durante la ejecución de una prueba, Appium recibe comandos procedentes de WebDriver[16] y los envía a WinAppDriver, y este es el encargado de ejecutar los comandos a la aplicación y de recoger los resultados que serán recibidos por Appium para ser enviados al cliente de automatización.

La automatización siempre se realiza en el contexto de una sesión. Los clientes inician una sesión en un servidor enviando una petición con un objeto JSON que contiene las

[16] https://www.selenium.dev/documentation/en/webdriver/

dessired capabilities. En este punto, el servidor inicializa la sesión de automatización y responde con un identificador de sesión que será utilizado para enviar más comandos.

Las *desired capabilities* son un conjunto de valores enviados al servidor de Appium para comunicar al servidor que tipo de sesión de automatización queremos iniciar. Hay varios aspectos que pueden modificar el comportamiento del servidor durante la automatización, como pueden ser la plataforma a la que pertenece la aplicación que queremos probar, el nombre del dispositivo o la ruta del ejecutable.

Existen muchas librerías cliente que admiten las extensiones de Appium para el protocolo WebDriver. Al utilizar Appium, es preferible utilizar estas librerías en lugar del cliente WebDriver común. Esto es debido a que el servidor Appium define extensiones personalizadas para los protocolos oficiales, lo que proporciona a los usuarios de Appium un acceso útil a varios comportamientos del dispositivo, como pueden ser la instalación o desinstalación de aplicaciones durante la ejecución de una prueba. Por eso se necesitan los clientes específicos de Appium, y no solo los clientes Selenium estándar [11].

En la figura 13 puede verse una tabla con los repositorios de *GitHub* para cada cliente específico de Appium.

Language/Framework	Github Repo and Installation Instructions
Ruby	https://github.com/appium/ruby_lib, https://github.com/appium/ruby_lib_core
Python	https://github.com/appium/python-client
Java	https://github.com/appium/java-client
JavaScript (Node.js)	https://github.com/admc/wd
JavaScript (Node.js)	https://github.com/webdriverio/webdriverio
JavaScript (Browser)	https://github.com/projectxyzio/web2driver
Objective C	https://github.com/appium/selenium-objective-c
PHP	https://github.com/appium/php-client
C# (.NET)	https://github.com/appium/appium-dotnet-driver
RobotFramework	https://github.com/serhatbolsu/robotframework-appiumlibrary

Figura 13: Distintos clientes específicos de Appium y su correspondiente repositorio en GitHub [13].

Appium puede instalarse mediante NPM[17] o directamente instalando Appium Desktop, que tiene una interfaz gráfica y es una forma más visual de lanzar el servidor de Appium. En el primer caso se debe instalar primero Node.js, y entonces podremos instalar Appium con el comando “npm install -g Appium” desde la consola de comandos. Si decidimos instalar Appium con el segundo método, será suficiente con ejecutar la aplicación y el servidor de Appium iniciará.

Cada plataforma tiene sus propios requisitos de instalación del entorno para poder automatizar con Appium. En el caso de Windows, necesitamos WinAppDriver, que va ya incluido en la instalación de Appium Desktop, y no hace falta instalarlo por separado. Para iniciar una sesión utilizando el driver de Windows tendremos que incluir las *capabilities* *“platformName”* con el valor “Windows”, *“deviceName”* con el valor “WindowsPC” y “app” con la ruta adecuada del ejecutable de la aplicación a probar. Dependiendo de si se va a probar una aplicación de Universal Windows Platform o una aplicación clásica de Windows, utilizaremos el Application ID o la ruta del ejecutable. Además, hay que recordar que hay que activar en Windows el modo desarrollador para poder automatizar aplicaciones de Windows.

El fragmento de código de la figura 14 pertenece a un ejemplo en el que se prueba la aplicación de calculadora de Windows. En él, puede verse cómo se configuran las *capabilities* de la sesión.

```
public static void Setup(TestContext context)
{
    // Launch Calculator application if it is not yet launched
    if (session == null)
    {
        // Create a new session to bring up an instance of the Calculator application
        // Note: Multiple calculator windows (instances) share the same process Id
        DesiredCapabilities appCapabilities = new DesiredCapabilities();
        appCapabilities.SetCapability("app", CalculatorAppId);
        appCapabilities.SetCapability("deviceName", "WindowsPC");
        session = new WindowsDriver<WindowsElement>(new Uri(WindowsApplicationDriverUrl), appCapabilities);
        Assert.IsNotNull(session);

        // Set implicit timeout to 1.5 seconds to make element search to retry every 500 ms for at most three times
        session.Manage().Timeouts().ImplicitWait = TimeSpan.FromSeconds(1.5);
    }
}
```

Figura 14: Configuración de las capabilities de una sesión de automatización de Appium con WinAppDriver [14].

[17] https://www.npmjs.com/

Es evidente y lógico, que la configuración sea muy parecida, si no igual, que en el caso de Winnium, y es debido a que ambos están basados en WebDriver. De igual forma, también es similar la forma de localizar e interactuar con los elementos de interfaz.

Windows SDK proporciona una buena herramienta para inspeccionar la aplicación que se está probando ya que permite ver todos los elementos o nodos de la interfaz con los que se puede interactuar mediante Windows Application Driver. La herramienta, llamada "inspector.exe", se encuentra en la ruta "C:\Program Files (x86)\Windows Kits\10\bin\x86".

La figura 15 muestra qué estrategia de búsqueda se debería utilizar para buscar elementos en la interfaz.

Client API	Locator Strategy	Matched Attribute in inspect.exe	Example
FindElementByAccessibilityId	accessibility id	AutomationId	AppNameTitle
FindElementByClassName	class name	ClassName	TextBlock
FindElementById	id	RuntimeId (decimal)	42.333896.3.1
FindElementByName	name	Name	Calculator
FindElementByTagName	tag name	LocalizedControlType (upper camel case)	Text
FindElementByXPath	xpath	Any	//Button[0]

Figura 15: Las estrategias de búsqueda de elementos de interfaz con su correspondiente atributo [13].

La mecánica es la misma que con Winnium. Se localizan los elementos de interfaz con los que se quiera interactuar utilizando un método de búsqueda en función de la estrategia de búsqueda que se crea adecuada, y se ejecutan comandos sobre él. En el caso del ejemplo de la calculadora, extraído del repositorio de WinAppDriver[18], se puede ver cómo localiza los botones de los números mediante diferentes estrategias de búsqueda, y utilizando el método "Click" los pulsa para realizar distintas operaciones. Finalmente comprueba el resultado utilizando métodos de la clase "Assert".

```
namespace CalculatorTest
{
    [TestClass]
    public class ScenarioStandard : CalculatorSession
    {
        private static WindowsElement header;
        private static WindowsElement calculatorResult;
```

[18] https://github.com/microsoft/WinAppDriver

```
        [TestMethod]
        public void Addition()
        {
            // Find the buttons by their names and click them in
sequence to perform 1 + 7 = 8
            session.FindElementByName("One").Click();
            session.FindElementByName("Plus").Click();
            session.FindElementByName("Seven").Click();
            session.FindElementByName("Equals").Click();
            Assert.AreEqual("8", GetCalculatorResultText());
        }

        [TestMethod]
        public void Division()
        {
            // Find the buttons by their accessibility ids and click
            them in sequence to perform 88 / 11 = 8

            session.FindElementByAccessibilityId("num8Button").Click();
            session.FindElementByAccessibilityId("num8Button").Click();
            session.FindElementByAccessibilityId("divideButton").Click(
            );
            session.FindElementByAccessibilityId("num1Button").Click();
            session.FindElementByAccessibilityId("num1Button").Click();
            session.FindElementByAccessibilityId("equalButton").Click()
            ;
            Assert.AreEqual("8", GetCalculatorResultText());
        }

        [TestMethod]
        public void Multiplication()
        {
            // Find the buttons by their names using XPath and click
            them in sequence to perform 9 x 9 = 81

            session.FindElementByXPath("//Button[@Name='Nine']").Click(
            );

            session.FindElementByXPath("//Button[@Name='Multiply
            by']").Click();
            session.FindElementByXPath("//Button[@Name='Nine']").Click(
            );
            session.FindElementByXPath("//Button[@Name='Equals']").Clic
            k();
            Assert.AreEqual("81", GetCalculatorResultText());
        }

        [TestMethod]
        public void Subtraction()
        {
            // Find the buttons by their accessibility ids using XPath
            and click them in sequence to perform 9 - 1 = 8

            session.FindElementByXPath("//Button[@AutomationId=\"num9Bu
            tton\"]").Click();
            session.FindElementByXPath("//Button[@AutomationId=\"minusB
            utton\"]").Click();
            session.FindElementByXPath("//Button[@AutomationId=\"num1Bu
            tton\"]").Click();
            session.FindElementByXPath("//Button[@AutomationId=\"equalB
            utton\"]").Click();
```

```
        Assert.AreEqual("8", GetCalculatorResultText());
    }

    [TestMethod]
    [DataRow("One",   "Plus",      "Seven", "8")]
    [DataRow("Nine",  "Minus",     "One",   "8")]
    [DataRow("Eight", "Divide by", "Eight", "1")]
    public void Templatized(string input1, string operation,
        string input2, string expectedResult)
    {
        // Run sequence of button presses specified above and
        validate the results
        session.FindElementByName(input1).Click();
        session.FindElementByName(operation).Click();
        session.FindElementByName(input2).Click();
        session.FindElementByName("Equals").Click();
        Assert.AreEqual(expectedResult,
              GetCalculatorResultText());
    }
```

3.1 Elección de la herramienta para la propuesta

Para decidir qué herramienta es la más adecuada para nuestro proyecto, hay aspectos a tener en cuenta como son el tipo de herramienta y la forma en la que se desarrollan las pruebas, la plataforma en la que deben ejecutarse y su compatibilidad, sistema operativo, la forma en la que la herramienta reporta los errores y/o resultados, la capacidad de personalización, rendimiento, etc.

Dentro de las distintas herramientas que existen, se pueden diferenciar las que tienen una interfaz desde la que se diseñan las pruebas (como las herramientas de tipo *record-playback*, en las que no hay que escribir código, sino que las pruebas están construidas grabando y reproduciendo las propias acciones a realizar) y las que se basan en librerías de código que pueden ser incluidas en un proyecto de pruebas propio (como *Selenium*[19]).

La elección de un tipo u otro dependerá de factores como la complejidad de las pruebas que se necesite implementar, el soporte por parte del propietario o la comunidad, o las habilidades de programación que se posea. Las herramientas basadas en *scripting* exigen tener cierto nivel de conocimientos para diseñar y desarrollar un proyecto con ciertas garantías de calidad, mientras que las de tipo *record-playback*, por ejemplo, suele bastar con tener un conocimiento nivel usuario de la herramienta y los principios del *testing*.

[19] https://www.selenium.dev/

En este caso se optó por un *framework* que se pudiese incorporar a un proyecto de *testing* desarrollado por la propia empresa, dada la flexibilidad y capacidad de adaptación que proporciona el hecho de poder escribir las pruebas desde código, y de poder incluir el *framework* de pruebas en su herramienta de integración continua. La empresa desarrolla mecánicas de generación automática de código, utilizando la *DSL-Tools* de *Microsoft*[20], por lo que había especial interés en dejar abierta la posibilidad de generar en algún momento estas pruebas de forma automática. Esto sólo sería posible si la herramienta permite la escritura de pruebas desde código fuente.

Existen entonces diferentes opciones que pueden cubrir parcialmente las necesidades del proyecto. Hay herramientas que podrían proporcionar un sistema de creación y ejecución de pruebas automáticas perfectamente válido. Sin embargo, el caso de estudio tiene una serie de requisitos, que se expondrán en los siguientes capítulos, en cuanto a la flexibilidad y control total sobre el código fuente del proyecto que sólo podía proporcionar una herramienta de tipo *scripting*.

Por lo tanto, la mejor opción es diseñar un proyecto desde cero con una herramienta que permita cumplir estos requisitos, así como facilitar su integración en el ciclo de desarrollo. Tener el completo control del código fuente del proyecto de pruebas permite una mejor gestión y libertad respecto a otras herramientas. Esto último es un gran punto a tener en cuenta si se planea generar el código de las pruebas de forma automática. También permite un gran control a la hora de decidir cómo gestionar los resultados de las pruebas.

De entre las herramientas de tipo *scripting* existentes, se opta por Appium dado su cada vez mayor uso y su continuo soporte por parte del propietario, y por las capacidades multiplataforma que se han comentado.

[20] https://docs.microsoft.com/es-es/visualstudio/modeling/overview-of-domain-specific-language-tools?view=vs-2019

4. Propuesta de organización de pruebas automatizadas

En este capítulo se presentará la estructura genérica de la solución para las pruebas diseñada para el caso de estudio. Se describirán los proyectos y clases, así como la función de cada una, su papel en la automatización de las pruebas y cómo se relacionan entre ellas para proporcionar esa genericidad.

4.1 Estructura de la solución

La solución .NET se divide en dos proyectos siguientes:

- **Proyecto *Forms*:** Este proyecto consta de una carpeta de clases llamada "Generic", y una carpeta por cada *framework* de pruebas que se vaya a utilizar, que en este caso será Appium. En la carpeta *Generic* estarán las interfaces de cada formulario que definen los casos de uso de ese formulario. Estas interfaces deberán ser implementadas por las clases propias de cada *framework*. En la carpeta Appium, estarán las clases en las que se implementarán los casos de uso definidos en las interfaces utilizando los mecanismos propios del *framework* que estemos utilizando. Se verá con más detalle más adelante. En este proyecto también tendremos la clase *Utils,* donde tendremos clases auxiliares que también veremos más adelante.
- **Proyecto *Tests:*** En este proyecto están las pruebas como tal. Contiene también una carpeta *Generic* y otra específica para cada *framework* que se utilice. En la carpeta *Generic* estarán las clases que definen los pasos de una prueba concreta de forma desacoplada del framework concreto. En la carpeta concreta del *framework* estarán las clases de tipo *TestClass* que contienen los *TestMethods* donde se instancian las clases específicas de la herramienta que se utilice.

En el diagrama de la figura 16 puede verse de forma más visual la organización de la solución.

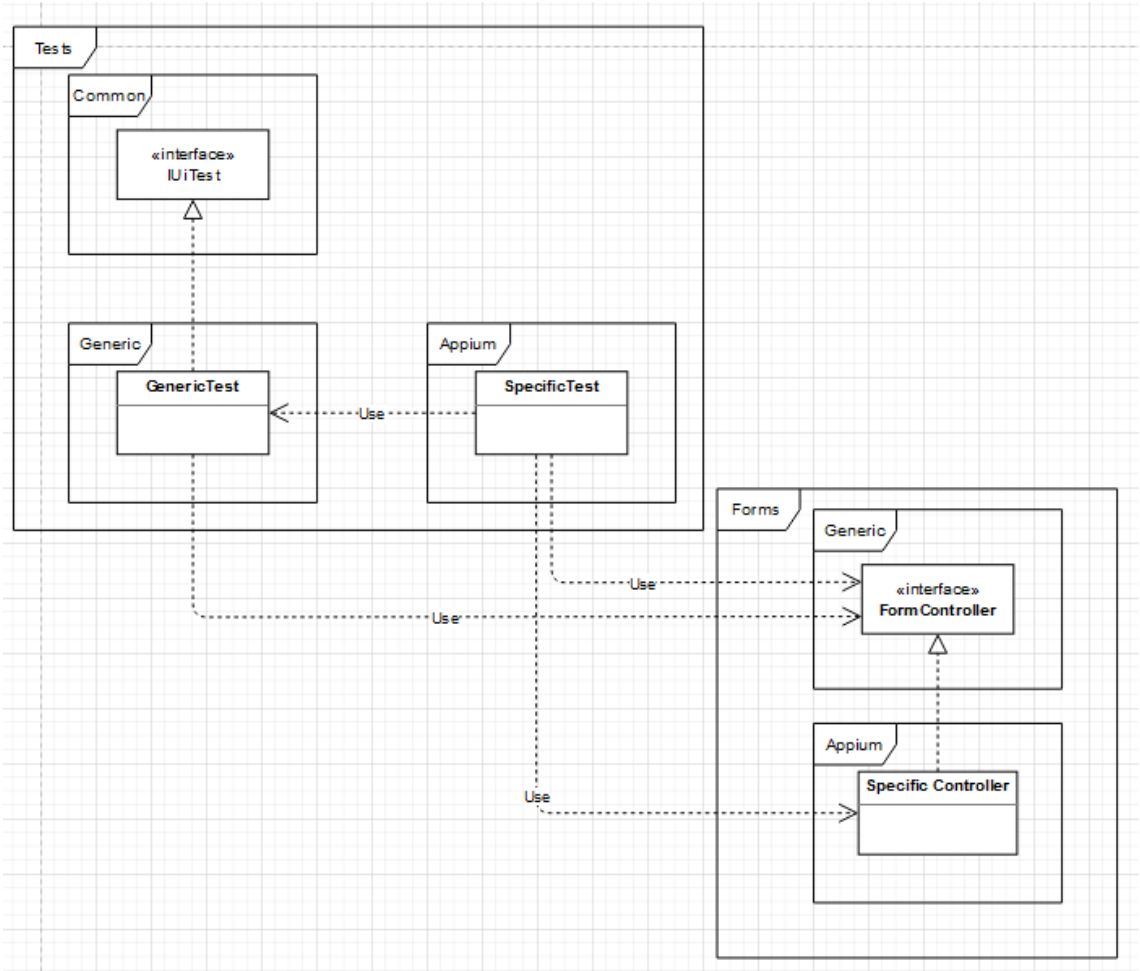

Figura 16: Diagrama de componentes de la solución.

4.1.1 El proyecto *Forms*.

El proyecto *Forms* contiene todas las interfaces que especifican las funciones que deben implementar cada formulario de la aplicación que se quiera automatizar, así como las clases que implementan esas interfaces por cada herramienta de pruebas utilizada. Esas funciones se organizan típicamente como casos de uso como tal, como pueden ser *logear* en la aplicación, elegir una opción de un menú o abrir un nuevo documento.

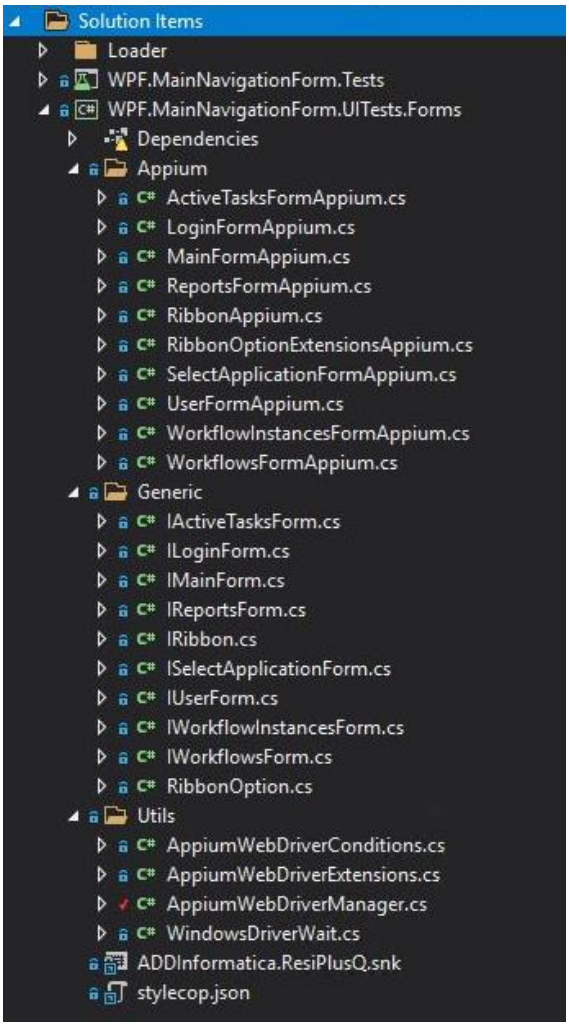

Figura 17: Organización del proyecto Forms en la solución.

En la figura 17 se presenta la organización del proyecto *Forms* dentro de la solución.

A. La carpeta Generic

Dentro del proyecto *Forms* está la carpeta *Generic*. En esta carpeta se encuentran las interfaces de todos los formularios que se vayan a automatizar.

Algunos ejemplos de las clases interfaz son:

La clase *ILoginForm* representa el formulario de *login* de la aplicación, y define que toda implementación para automatizar esta aplicación, sea con el *framework* que sea, debe contener definiciones específicas para esos métodos. En este caso, encontramos, lógicamente, la definición del método *LoginAs* que, como veremos más adelante, tendrán que implementar las clases propias de cada herramienta con sus mecanismos propios.

```
public interface ILoginForm
{
    /// <summary> Method for log-in using the given credentials.
    /// </summary>
    /// <param name="userName"> Name of the user. </param>
    /// <param name="password"> The password of the user. </param>
    void LoginAs(string userName, string password);
}
```

La clase *IRibbon* representa el menú superior de la aplicación y define un método para la selección de opciones del menú.

```
public interface IRibbon
{
    /// <summary> Method for selecting a Ribbon option. </summary>
    /// <param name="ribbonOption"> The Ribbon option. </param>
    void SelectRibbonOption(RibbonOption ribbonOption);
}
```

En otros formularios, se necesitaba comprobar que estos cargaban correctamente y se definió un método para comprobar que el formulario había cargado sin problemas. En las pruebas, se combinaría la funcionalidad del *Ribbon* para seleccionar una opción del menú, con los definidos en los formularios donde se comprobase la carga.

Uno de esos formularios, por ejemplo, es el *UserForm*, y su método para comprobar la carga de ese formulario es *CheckUserFormIsLoaded.*

```
public interface IUserForm
{
    /// <summary> Method that checks if User Form is loaded.
    /// </summary>
    /// <returns> True if it's loaded, false if not. </returns>
    bool CheckUserFormIsLoaded();
}
```

Estas clases interfaz permiten desacoplar las clases de prueba del proyecto *Tests* de las implementaciones concretas de cada herramienta, pero se explicará con más detalle en el apartado del proyecto *Tests.*

B. La carpeta Appium

En la carpeta Appium, se encuentran las clases que implementan las interfaces genéricas y que ofrecen los casos de uso definidos en las interfaces implementadas con los mecanismos propios de cada herramienta. Estas clases actúan como controlador de la aplicación y son las encargadas de realizar las operaciones para cada prueba.

Así, por ejemplo, la clase *LoginFormAppium* es un controlador del formulario de *login* de la aplicación, que implementa el método público *LoginAs* definido en la interfaz

ILoginForm, y otros métodos privados que utiliza la propia clase para llevar a cabo el caso de uso asociado al intento de inicio de sesión. En estos métodos, utilizando la sesión de WinDriver que, como se vio anteriormente, utiliza *Appium* para manejar la aplicación y que se le pasa como parámetro en el constructor, controla el formulario para introducir los datos de acceso y controlar los botones de *Login* y *Cancel.*

```
/// <summary> Form for viewing the login. </summary>
public class LoginFormAppium : ILoginForm
{
    private readonly WindowsDriver<WindowsElement> session;

    /// <summary> Initializes a new instance of the <see
    /// cref="LoginFormAppium"/> class. </summary>
    /// <param name="session"> The session. </param>
    public LoginFormAppium(WindowsDriver<WindowsElement> session)
    {
        this.session = session;
    }

    /// <inheritdoc/>
    public void LoginAs(string userName, string password)
    {
        this.InsertUserName(userName);
        this.InsertPassword(password);
        this.ClickLoginButton();
        this.session.WaitUntilWindowFocusChanges(20);
    }

    /// <summary> Inserts an user name in the user name field.
    /// </summary>
    /// <param name="userName"> The user name. </param>
    private void InsertUserName(string userName)
    {
        this.session.FindElementByAccessibilityId(LoginFormTestCons
        tants.UserNameTextBox).SendKeys(userName);
    }

    /// <summary> Inserts a password in the password field.
    /// </summary>
    /// <param name="password"> The password. </param>
    private void InsertPassword(string password)
    {
        this.session.FindElementByAccessibilityId(LoginFormTestCons
        tants.PasswordPasswordBox).SendKeys(password);
    }

    /// <summary> Click login button. </summary>
    private void ClickLoginButton()
    {
        this.session.FindElementByAccessibilityId(LoginFormTestCons
        tants.LoginCommandButton).Click();
    }

    /// <summary> Click cancel button. </summary>
    private void ClickCancelButton()
    {
```

```
            this.session.FindElementByAccessibilityId(LoginFormTestCons
            tants.CancelCommandButton).Click();
        }
    }
```

El hecho de organizar los métodos como casos de uso, evita que desde las clases *Test* se haga referencia a pasos concretos como hacer *click* o introducir el usuario y la contraseña, lo que hace que las clases de prueba sean mucho más legibles y limpias.

C. La carpeta Utils

En la carpeta *Utils* se encuentran algunas clases en las que se ha desarrollado código de apoyo que permite realizar funciones especiales como las esperas mientras carga un formulario o la monitorización del cambio de un formulario a otro.

La clase *AppiumWebDriverExtensions* es una clase extensión de Windows driver en la que se han implementado unos métodos *wait* que se utilizan para realizar las esperas necesarias debido al retardo producido en la aplicación debido a tiempos de carga o cambios de ventana, por ejemplo. En este caso, se han implementado los métodos *WaitUntilWindowFocusChanges* y *WaitUntilFormLoads* que se utilizan en las pruebas de carga de formularios para provocar que la aplicación se detenga mientras la aplicación carga el formulario nuevo, o cambie el foco de ventana en el momento de iniciar sesión.

```
/// <summary> Extension methods for Appium Windows driver.
/// </summary>
public static class AppiumWebDriverExtensions
{
    private const int DefaultTimeout = 20;

    /// <summary>
    ///     Extension method for Windows driver that implements a
    /// wait for window focus change.
    /// </summary>
    /// <param name="session"> The Windows driver session.
    /// </param>
    /// <param name="timeoutInSeconds"> The timeout in seconds.
    /// </param>
    public static void WaitUntilWindowFocusChanges(this
        WindowsDriver<WindowsElement> session, int timeoutInSeconds
        = DefaultTimeout)
    {
        WindowsDriverWait windowsDriverWait = new
        WindowsDriverWait(session,
        TimeSpan.FromSeconds(timeoutInSeconds));

        try
        {
              windowsDriverWait.Until(AppiumWebDriverConditions.Win
              dowsFocusHasChanged(session));
               session.SwitchTo().Window(session.WindowHandles[0]);
        }
```

```
            catch (WebDriverTimeoutException)
            {
                        throw;
            }
        }

        /// <summary>
        ///     Extension method for Windows driver that implements a
        /// wait for the given form to be loaded.
        /// </summary>
        /// <param name="session"> The Windows driver session.
        /// </param>
        /// <param name="formAutomationId"> The form automation
        /// identifier. </param>
        /// <param name="timeoutInSeconds"> The timeout in seconds.
        /// </param>
        public static void WaitUntilFormLoads(this
            WindowsDriver<WindowsElement> session, string
            formAutomationId, int timeoutInSeconds = DefaultTimeout)
        {
            WindowsDriverWait windowsDriverWait = new
            WindowsDriverWait(session,
            TimeSpan.FromSeconds(timeoutInSeconds));

            try
            {

                  windowsDriverWait.Until(AppiumWebDriverConditions.For
                  mIsLoaded(session, formAutomationId));
            }
            catch (WebDriverTimeoutException)
            {
                throw;
            }
        }
    }
```

Como puede verse en el fragmento de código anterior, los métodos de espera reciben como parámetro una sesión de *WinDriver* y un entero que representa el periodo de tiempo que ha de esperar la aplicación. Después se construye un nuevo objeto de tipo *WindowsDriverWait* que es una implementación propia de *WinDriver* que proporciona métodos de espera como "*Until*".

Llegado un punto se necesitaron unas condiciones especiales que había que definir por separado de alguna forma. Es por eso que no se utilizaron los métodos de espera proporcionados por *WinDriver.*

El método *"Until"* pausa la ejecución del programa hasta que se cumpla una condición, y es ahí donde entra en juego la clase *AppiumWebDriverConditions.* En esta clase extensión se implementaron las funciones anónimas *WindowsFocusHasChanged* y *FormIsLoaded* que se le pasan como parámetro al método *Until* de *WinDriver* y funcionan como una variable de tipo booleana. De esta forma, el método *Until* pausará

la ejecución hasta que la función anónima *WindowsFocusHasChanged* detecte que el identificador del formulario con el foco es distinto, momento en el que devolverá *true* y la ejecución continúe.

```
public static class AppiumWebDriverConditions
    {
        /// <summary> Method that checks if the windows focus has
        /// changed. </summary>
        /// <param name="session"> The Windows driver. </param>
        /// <returns> A Func that checks if the windows focus has
        /// changed. </returns>
        public static Func<WindowsDriver<WindowsElement>, bool>
            WindowsFocusHasChanged(WindowsDriver<WindowsElement>
            session)
        {
            return (driver) =>
            {
                try
                {
                    if (!string.Equals(session.WindowHandles[0],
                        session.CurrentWindowHandle,
                        StringComparison.Ordinal))
                    {
                        return true;
                    }
                }
                catch (Exception)
                {
                    return false;
                }

                return false;
            };
        }
```

```
/// <summary> Method that checks if a given form is already
        /// loaded. </summary>
        /// <param name="session"> The Windows driver. </param>
        /// <param name="formAutomationId"> The form automation
        /// identifier. </param>
        /// <returns> A Func that checks if the given form is already
        /// loaded. </returns>
        public static Func<WindowsDriver<WindowsElement>, bool>
            FormIsLoaded(WindowsDriver<WindowsElement> session, string
            formAutomationId)
        {
            return (driver) =>
            {
                try
                {
                    IReadOnlyCollection<WindowsElement> elements =
                        session.FindElementsByAccessibilityId(formAutom
                        ationId);

                    return elements.Count > 0;
                }
                catch (Exception)
                {
                    return false;
                }
            };
        }
    }
```

La clase *AppiumWebDriverManager* se creó como clase extensión de WinDriver para implementar el patrón *Disposable* o “desechable”, de forma que, en lugar de instanciar directamente una sesión *WebDriver,* se instanciará la clase *AppiumWebDriverManager* y, cuando ya no sea usada, se lanzará el método *Dispose* y terminará completamente la sesión. Esto se consigue encapsulando cada prueba dentro de una cláusula *using*, y se verá con más detalle en el apartado del proyecto *Tests*.

Esto, aparte de mejorar la eficiencia de recursos, evita que se queden sesiones abiertas entre las ejecuciones de las pruebas y provoque errores producidos por el bloqueo del puerto que se utiliza *WebDriver*, que es común para todas las pruebas.

```
public class AppiumWebDriverManager : IDisposable
    {
        private const int DefaultImplicitWaitTime = 20;

        private bool disposed = false;

        /// <summary>
        /// Initializes a new instance of the <see
        /// cref="AppiumWebDriverManager"/> class.
        /// </summary>
        /// <param name="appCapabilities"> The app capabilities.
        /// </param>
        /// <param name="uri"> The URI. </param>
```

```
        /// <param name="implicitWaitTime"> The implicit wait time.
        /// </param>
        public AppiumWebDriverManager(AppiumOptions appCapabilities,
            Uri uri, int implicitWaitTime = DefaultImplicitWaitTime)
        {
            this.Driver = new WindowsDriver<WindowsElement>(uri,
                  appCapabilities);

            if (implicitWaitTime != DefaultImplicitWaitTime)
            {
                this.Driver.Manage().Timeouts().ImplicitWait =
                  TimeSpan.FromSeconds(implicitWaitTime);
            }
        }

        /// <summary> Gets the web driver. </summary>
        public WindowsDriver<WindowsElement> Driver { get; }

        /// <summary> Public implementation of Dispose pattern.
        /// </summary>
        public void Dispose()
        {
            this.Dispose(true);
            GC.SuppressFinalize(this);
        }

        /// <summary> Protected implementation of Dispose pattern.
        /// </summary>
        /// <param name="disposing"> The disposing value. </param>
        protected virtual void Dispose(bool disposing)
        {
            if (this.disposed)
            {
                return;
            }

            if (disposing)
            {
                this.Driver.Quit();
                this.Driver.Dispose();
            }

            this.disposed = true;
        }
    }
```

4.1.2 El proyecto *Tests*.

En el proyecto *Tests* se encuentran las clases de prueba como tal, además de las clases que implementan cada caso de uso de forma genérica haciendo las pruebas independientes de la herramienta que se utilice. Estas clases además implementan la interfaz *IUiTest*, donde se define el método *ExecuteTest* que contiene todos los pasos necesarios para llevar a cabo la prueba y que deberá estar presente en todas las clases caso de uso. Este devolverá el resultado para ser utilizado en los asertos de las clases de prueba.

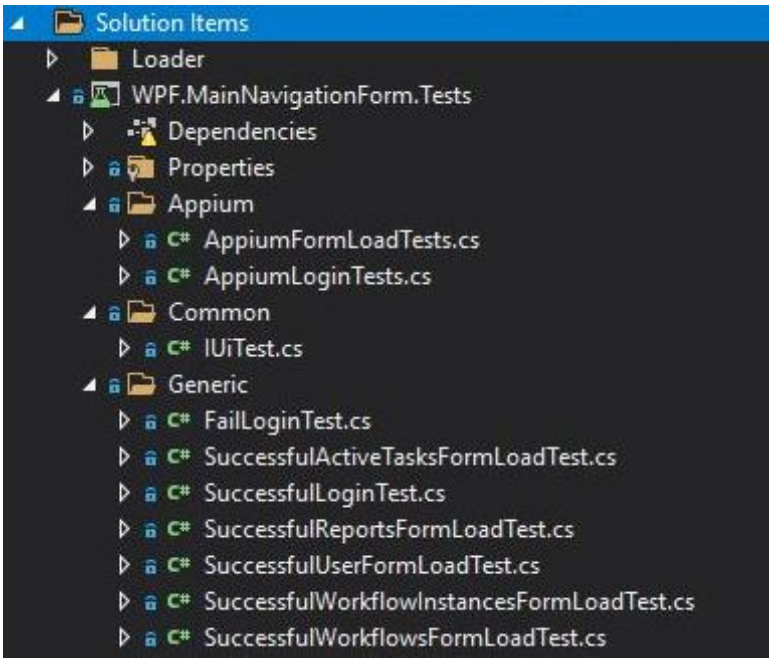

Figura 18: Estructura del proyecto Tests.

La figura 18 muestra la organización del proyecto *Tests* dentro de la solución.

Estas clases se instanciarán en las clases de prueba específicas de la herramienta, que se encuentran en la carpeta Appium en este caso, utilizando los controladores propios definidos en el proyecto *Forms*.

Vamos a explicar de formas más detallada las clases que hay en este proyecto para entender mejor cómo se relacionan unas con otras y los beneficios de estructurar de esta forma la solución y cómo esa genericidad permite ahorrar costes de tiempo a la hora de adaptar la solución a otra herramienta.

A. La carpeta Generic.

En la carpeta *Generic* están las clases que representan una prueba concreta y que implementan toda la funcionalidad necesaria para llevarla a cabo y devolver un único resultado booleano en función de si la prueba ha resultado exitosa o no.

En el constructor de estas clases se pasan como parámetros los controladores genéricos de todos los formularios que estén involucrados en la ejecución de la prueba. Los parámetros han de tener el tipo de las interfaces genéricas de los controladores, de forma que, en las clases de prueba, se instancian estas clases genéricas con el tipo específico del controlador de la herramienta que se quiera utilizar en ese momento, quedando totalmente desacopladas de las clases o funciones específicas de la herramienta utilizada.

Para verlo más claro lo ilustraremos utilizando como ejemplo la clase *SuccessfulLoginTest.*

Esta clase representa la prueba en la que el usuario de la aplicación realiza un inicio de sesión de forma exitosa. A su vez, también hay otra clase similar, *FailLoginTest*, para el caso en el que el inicio de sesión es incorrecto.

En la clase pueden verse el constructor y el método público *ExecuteTest*, definido en la interfaz que implementa esta clase y que todas las clases de prueba genéricas deben implementar.

En el caso de uso de un inicio de sesión con éxito en la aplicación que se está probando, primeramente, hay que lanzar la aplicación, después introducir los datos del usuario y, si los datos son correctos, seleccionar la aplicación llamada *Prototype.*

```
/// <summary> Generic login test. </summary>
internal class SuccessfulLoginTest : IUiTest
{
    private readonly ILoginForm loginForm;
    private readonly ISelectApplicationForm selectApplicationForm;
    private readonly IMainForm mainForm;

    private readonly string userName = "Alejandro@ResiPlusQ.com";
    private readonly string password = "123456aA.";

    /// <summary>
    ///     Initializes a new instance of the <see
    /// cref="SuccessfulLoginTest"/> class.
    /// </summary>
    /// <param name="loginForm"> The log-in form. </param>
    /// <param name="selectApplicationForm"> The select
    /// application form. </param>
    /// <param name="mainForm"> The main form. </param>
    public SuccessfulLoginTest(ILoginForm loginForm,
        IMainForm mainForm, ISelectApplicationForm
        selectApplicationForm)
    {
        this.loginForm = loginForm;
        this.selectApplicationForm = selectApplicationForm;
        this.mainForm = mainForm;
    }

    /// <summary> Executes the test operation. </summary>
    /// <returns> True if the execution is success, false
    /// otherwise. </returns>
    public bool ExecuteTest()
    {
        this.loginForm.LoginAs(this.userName, this.password);

        this.selectApplicationForm.SelectApplication("Prototype");

        return this.mainForm.CheckMainFormIsLoaded();
    }
}
```

Vemos entonces que, en esta prueba, se ven involucrados tres formularios: el formulario de inicio de sesión para introducir los datos, el formulario de selección de aplicación y el formulario principal de la aplicación para comprobar que ha cargado, lo que significaría que el inicio de sesión es correcto.

Por tanto, como puede verse en el fragmento de código de la clase, en el constructor se le pasa como parámetro los tres controladores genéricos de esos formularios, que son, *ILoginForm, IMainForm* y *ISelectApplicationForm.*

En el método *Execute*, estarán los pasos a seguir para la prueba utilizando los mecanismos necesarios que están implementados en las clases controlador específicas de cada herramienta.

En este ejemplo, primero se llama al método *LoginAs*, definido en las clases que implementan la interfaz *ILoginForm*, utilizando las credenciales definidas para este caso de prueba. Después, se lanza el método *SelectApplication* del formulario de selección de aplicación, y finalmente se devuelve el resultado de tipo booleano del método *CheckMainFormIsLoaded* del formulario *Main.* Si este formulario ha cargado, significa que el intento de inicio de sesión ha sido correcto.

Es fácil observar, que esta clase no está acoplada a ninguna clase específica de una herramienta porque sus mecanismos propios están en las clases controlador específicas, y que, gracias a esto, las clases genéricas que definen los mecanismos de una prueba no necesitan ser modificadas cuando se quiere introducir una nueva herramienta de pruebas.

B. La carpeta Appium

En la carpeta *Appium* están las clases de prueba o *TestClass*. En ellas se instancian los casos de prueba utilizando los controladores específicos de la herramienta que se use.

Estas clases tienen dos secciones diferenciables. Por un lado, el bloque *TestInitialize* donde se preparan los requisitos previos a la ejecución de la prueba, como por ejemplo la configuración de las *Capabilities* del *WebDriver* como se explicó en el punto donde se habla de la herramienta. El fragmento de código siguiente pertenece a la clase *AppiumLoginTests*, en la que se agrupan las pruebas relacionadas con el inicio de sesión para la implementación con la herramienta *Appium,* y contiene el bloque *TestInitialize*.

```
/// <summary> Test class set up. </summary>
[TestInitialize]
 public void TestSetUp()
 {
      this.appCapabilities = new AppiumOptions();

      this.appCapabilities.AddAdditionalCapability("deviceName",
            @"WindowsPC");

      this.appCapabilities.AddAdditionalCapability("platformName",
            @"Windows");

      this.appCapabilities.AddAdditionalCapability("app",
            @"Loader.exe");
  }
```

Los *TestMethods* son los métodos donde se instancian las clases de prueba genéricas para un caso de prueba concreto. Una vez instanciados los controladores necesarios para una prueba determinada, y utilizados para construir la instancia de esta, se llama al método *Execute*.

En el caso de la clase *AppiumLoginTests*, a la hora de crear las instancias de los controladores utilizados en las clases de prueba genéricas, en el tipo estático del objeto construido se indica la clase interfaz genérica, como por ejemplo *ILoginForm*. Esto se debe a que la clase de pruebas genérica espera recibir un objeto de tipo *ILoginForm*. Sin embargo, en el tipo dinámico, se indicará el tipo del controlador de formulario específico, en este caso sería *LoginFormAppium*.

De esta forma, si fuese necesario cambiar la herramienta utilizada, sólo sería necesario cambiar los tipos dinámicos de esta clase, y crear las clases controlador específicas de la nueva herramienta.

En los ejemplos de inicio de sesión, el valor devuelto por el método Execute se utiliza en el método *IsTrue* de la clase *Assert* para verificar que el resultado es el esperado.

```
/// <summary>
///     The method: LoginSucces_Appium, when the log-in is success
/// using the given
///     credentials, it validates without errors.
/// </summary>
[TestMethod]
public void LoginSucces_Appium()
{
      using (AppiumWebDriverManager session =
            new AppiumWebDriverManager(
            this.appCapabilities,
            new Uri("http://127.0.0.1:4723/wd/hub")))
      {
            // Arrange.
            ILoginForm loginFormAppium =
                  new LoginFormAppium(session.Driver);

            ISelectApplicationForm selectApplicationForm =
                  new SelectApplicationFormAppium(session.Driver);

            IMainForm mainForm =
                  new MainFormAppium(session.Driver);

            IUiTest loginTest =
                  new SuccessfulLoginTest(loginFormAppium, mainForm,
                  selectApplicationForm);

            // Act.
            bool result = loginTest.ExecuteTest();

            // Assert.
            Assert.IsTrue(result, "Login has failed.");
      }
}
```

También se puede observar el uso del patrón *disposable* a la hora de instanciar la sesión de *Appium* utilizada en los controladores para evitar que quede cualquier rastro de ella una vez finalizada la prueba. La sesión se instancia dentro de una cláusula *using*, y, una vez se haya ejecutado todo el bloque que contiene, la sesión será destruida completamente.

4.2 Lanzamiento y validación de las pruebas

Este proyecto se ha desarrollado junto al equipo de desarrollo del departamento de I+D+i de la empresa interesada en la propuesta.

El equipo de I+D+i decidió hacerse cargo de la parte de integración en el ciclo de desarrollo, así como el lanzamiento, validación y gestión de los resultados de las pruebas de la batería, por lo que el detallado de esa parte queda fuera del objetivo de este trabajo, ya que no se conocen los detalles del proceso. Sin embargo, a continuación, se comentan algunos aspectos a grandes rasgos del uso actual de la batería de pruebas.

La solución de la batería está integrada en la rama master del proceso de desarrollo de la aplicación principal. Existe un paso en el *pipeline* de la aplicación, que se ejecuta en el lanzamiento de cada nueva versión, previo al despliegue final, en el que se ejecutan las pruebas de la batería y se valida que las versiones de cada componente de la aplicación son los que corresponden a la versión a desplegar.

Actualmente, no se ha desarrollado ningún método específico para la gestión de los errores. Si la ejecución de las pruebas notificase algún error, el despliegue fallaría y se procedería manualmente a revisar las versiones de los componentes. Esto es debido en parte al temprano estado de desarrollo en el que se encuentra el software, dado que por ahora no supone un gran esfuerzo realizar esa revisión de forma manual, se ha decidido posponer la inversión de tiempo necesaria para más adelante.

5. Migración de la batería de pruebas

Durante todo el trabajo se ha remarcado que uno de los principales objetivos era poder migrar la batería de pruebas a otra herramienta, llegado el caso, sin tener que rehacer toda la solución y con el menor costo de tiempo y recursos posible.

A continuación, se presentará una comparativa en la que se pueda ver más fácilmente el ahorro de tiempo y recursos que supone la migración a otra herramienta de pruebas de una batería de pruebas dentro de una solución sin estructurar respecto a la solución genérica propuesta.

La comparativa se realizará contando el número de líneas de código que sería necesario reescribir en caso de migrar la batería a otra herramienta. Se utilizará como ejemplo la solución de la propuesta presentada en este trabajo y se extrapolará el resultado a una batería con un mayor número de pruebas.

Para el cálculo de líneas a reescribir en una solución sin ninguna estructura genérica, supondremos que las pruebas son escritas de forma desestructurada. Esto es, que las pruebas son implementadas paso a paso utilizando directamente los mecanismos de la herramienta en las propias clases de prueba. Para readaptar la solución, sería necesario revisar todas las pruebas e implementar las mismas con los mecanismos de la nueva herramienta, por lo que, a efectos prácticos, habría que revisar prácticamente todas las líneas de las pruebas lo que supone un coste de tiempo muy alto, además de la reducción de fiabilidad en las pruebas que son sometidas a importantes modificaciones.

Utilizando la propuesta presentada, sólo sería necesario revisar las clases de las carpetas específicas, en este caso las carpetas llamadas *Appium,* tanto del proyecto *Forms* como la del proyecto *Tests*. Por lo tanto, sólo habría que implementar las mismas funciones en las clases controlador específicas de la carpeta *Appium* del proyecto *Forms,* y cambiar el tipo dinámico de los controladores instanciados en las clases de prueba de la carpeta *Appium* del proyecto *Tests.* Incluso, el cambio del tipo dinámico, al tener localizados los tipos dinámicos anteriores y conocer el nombre de los nuevos, puede reducirse a la sustitución de forma sistemática de unos nombres por otros, lo que reduce el tiempo empleado a prácticamente nulo.

En la siguiente tabla se muestra el número de líneas a revisar en los dos casos expuestos, suponiendo que la nueva herramienta tenga los mismos mecanismos que *Appium*. De esta forma, excluimos de la ecuación la diferencia de coste de implementar cada función en una herramienta respecto a otra, y aislamos la mejora que proporciona la estructura de la propuesta.

	Genérica	No genérica
Líneas totales implementadas originalmente	1459	1157
Líneas susceptibles de ser modificadas	496	1157
Porcentaje	33%	100%

Tabla 2: Comparativa de líneas a revisar en la migración de la solución.

En la tabla 2 se comparan las líneas de código totales escritas en cada uno de los diseños respecto a las líneas susceptibles de ser examinadas, para comprobar si es necesario modificarlas, en una supuesta migración. Se puede observar que, aunque en una estructura genérica el esfuerzo inicial es mayor debido a la complejidad añadida a la solución, el coste posterior potencial invertido en el mantenimiento y/o una posible migración se ve reducido en un 77%, compensando con creces el mayor coste inicial del proyecto.

También hay que considerar que en soluciones de mayor envergadura el número de líneas escritas en una solución no genérica puede dispararse, y acabar siendo incluso mayor que si se usase una estructura genérica, además de hacerlo muy difícil de manejar de cara al mantenimiento.

6. Metodología

Inicialmente se llevaron a cabo una serie de reuniones con el equipo de I+D+i en la que se pusieron en común tanto los requisitos que el equipo tenía para este proyecto, como las sugerencias de cada miembro que pudiesen ayudar a elaborar una propuesta. Partiendo de esa puesta en común se propuso una solución inicial, que era una versión básica de la propuesta que finalmente se desarrollaría.

Tras un tiempo analizando la propuesta e investigando posibles alternativas ya existentes similares a lo que se buscaba. Sin mucho éxito, en posteriores reuniones se decidió desarrollar la solución desde cero. En esas reuniones, el equipo de I+D+i indicaron qué pruebas se necesitaban y se inició la búsqueda de una herramienta adecuada para la propuesta.

Se inició la planificación y desarrollo de la solución, empezando por plantear una estructura genérica en la que desacoplar las pruebas de la herramienta utilizada. Tras decidir separar el código en dos proyectos, se inició la implementación de los proyectos *Forms* y *Tests* para las pruebas indicadas por el equipo. El desarrollo de ambos proyectos se realizaba de forma paralela implementando los controladores genéricos de un formulario junto con los controladores específicos de *Appium* para poder implementar los *tests* de ese formulario.

Durante la implementación, el código se sometía a revisiones de código por parte del equipo de I+D+i y se discutían posibles mejoras. Una vez la solución estaba terminada, se realizó la integración a la rama *master* de la aplicación que se está probando para poder lanzarla en cada *release*.

En la figura 19 se muestra un cronograma con los principales hitos del desarrollo del trabajo.

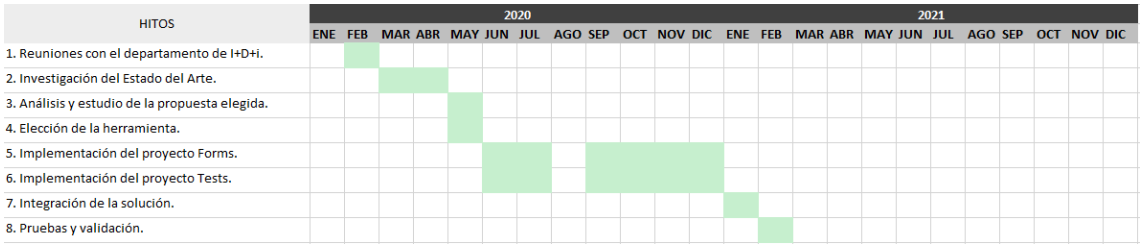

Figura 19: Cronograma del desarrollo del trabajo.

7. Conclusiones y trabajos futuros

En este capítulo se expondrán las conclusiones finales, así como la relación con asignaturas cursadas en el grado y posibles mejoras futuras del proyecto que puedan ser de utilidad en el entorno de trabajo de la empresa.

7.1 Conclusiones.

El principal objetivo de este proyecto era conseguir una batería de pruebas que solucionasen ese problema de versionado en cada lanzamiento, y que fuese suficientemente flexible como para ampliarla con más pruebas, ejecutarlo en otras plataformas, o migrar a otra herramienta de pruebas si fuese necesario. Esto se ha conseguido mediante tres factores principales que son:

- La elección de *Appium* como herramienta de pruebas, proporcionando la capacidad multiplataforma que permitirá producir versiones de la batería para distintas plataformas.
- Uso de una herramienta de tipo *scripting,* lo que permite la integración del desarrollo de pruebas en el marco de desarrollo dirigido por modelos, y autogenerar el código fuente de las pruebas con *DSL-Tools*.
- Implementación de una estructura genérica que permita la migración a otra herramienta de pruebas con el menor coste de tiempo y recursos posible.

Actualmente, el proyecto está integrado en la rama principal de desarrollo y se ejecuta periódicamente en los despliegues antes del lanzamiento de cada versión de la aplicación, y los errores de versionado ya no suponen un problema, puesto que se detectan antes de su despliegue en producción.

El grueso de este trabajo se ha enfocado en el diseño de la solución genérica, en su correcta automatización de la aplicación con la herramienta de pruebas y en su integración en el ciclo de desarrollo, y pese a que la batería consta apenas de unas pocas pruebas, han sido suficientes para cumplir el objetivo de este proyecto, reuniendo las características necesarias que se plantearon en un principio.

Después de este estudio, la empresa cuenta con un proyecto de pruebas para su nueva aplicación que servirá como apoyo a la batería principal de pruebas unitarias y que deja abierto un abanico de posibilidades y mejoras.

A nivel personal, participar en este proyecto me ha proporcionado una visión general de todo el proceso de desarrollo, que es algo que uno no llega a visualizar del todo hasta que te involucras en uno, por mucho que estudies el proceso de forma docente. He podido participar en reuniones muy interesantes donde todos los miembros del equipo ponían en común sus propias ideas, y he podido discutirlas de forma dinámica, cosa que personalmente me resulta muy interesante. He sido un eslabón más en la cadena que es un equipo de desarrollo de software, participando en investigaciones, estudios de diferentes tecnologías, revisiones de código y muchos aspectos más propios de un proceso de este tipo. He podido mejorar mi capacidad de estimación de las tareas que componen un proyecto, ayudándome a ser más preciso a la hora de planificarlas y determinar costes en términos de tiempo y recursos, y he aprendido a sincronizarme con otras personas, lo que se traduce en una mejora de la capacidad de trabajo en equipo.

7.2 Relación con los estudios realizados en el grado

En el desarrollo de este proyecto han sido realmente útiles los conocimientos adquiridos en varias de las asignaturas cursadas en el grado.

Primeramente, *Proceso de Software (PSW)* y el *Proyecto de ingeniería de software (PIN)* han ayudado a comprender la dinámica de desarrollo de un proyecto de software dentro del proceso que sigue una empresa que utiliza metodologías ágiles en su ritmo diario de trabajo y a ejecutar e integrar el desarrollo de este proyecto dentro de ese proceso.

En *Diseño de software (DDS)* se estudió cómo estructurar un proyecto correctamente y a aplicar estructuras y patrones de diseño que se han utilizado en el desarrollo de la propuesta presentada y, junto a los conocimientos obtenidos en *Mantenimiento y evolución del software (MES)*, ha permitido producir código de calidad y con un alto nivel de mantenibilidad.

7.3 Trabajos futuros

Algunas de las mejoras pendientes pueden ser la ampliación de la batería de pruebas para incluir casos de prueba adicionales que no pudiesen ser cubiertos con pruebas unitarias. El proyecto por ahora cubre mayormente el correcto despliegue y carga de la aplicación en el lanzamiento de cada versión. Pero es probable que, cuando la aplicación crezca en tamaño y complejidad, surjan nuevas funcionalidades que deban ser probadas y no sea posible hacerlo mediante otro tipo de pruebas.

Por otro lado, sería interesante la integración del proyecto en el ciclo de generación automática de código con la que se genera gran parte del código de la aplicación en pruebas, al menos la parte genérica de ellas. Esto reduciría el coste aún más a la hora de generar nuevas pruebas y mantener las ya existentes.

También se utilizará la solución en otras plataformas, puesto que está planeada la implementación de la aplicación en distintas plataformas, entre ellas la móvil, una vez se hayan desarrollado. La elección de *Appium* como herramienta se hizo con vistas que este proceso fuese lo más fácil posible, y hará que esa migración sea mucho más llevadera por la característica *cross-platform* de la que se habló anteriormente.

8. Referencias

[1] Quijano, Juan, «¿Qué pruebas debemos hacerle a nuestro software y por qué?» [En Línea, consultado en https://www.genbeta.com/desarrollo/que-pruebas-debemos-hacerle-a-nuestro-software-y-para-que] [Accedido: 10-11-2020]

[2] Kent Beck, «Test Driven Development: By Example », 2003 [En línea, consultado en https://books.google.es/books?id=CUlsAQAAQBAJ&lpg=PR7&ots=QCdXYa6PQ-&dq=test%20driven%20development%20by%20example&lr&hl=es&pg=PR7#v=onepage&q=test%20driven%20development%20by%20example&f=false] [Accedido: 15-11-2020]

[3] Levison, «M. Adavantages of TDD», [En línea, consultado en http://www.pmoinformatica.com/2012/12/TEST-DRIVEN-DEVELOPMENT-VENTAJAS-Y.HTML] [Accedido: 15-11-2020]

[4] Charles Rodriguez, Alejandro Berardinelli, «When to Automate a Test», [En línea, consultado en https://abstracta.us/blog/test-automation/when-to-automate-a-test/] [Accedido: 13-12-2020]

[5] Cohn, Mike, «Succeeding With Agile», 2009, [En línea, consultado en https://www.oreilly.com/library/view/succeeding-with-agile/9780321660534/] [Accedido: 20-1-2021]

[6] Javier Garzas, «Automatización de pruebas», [En línea, consultado en https://www.javiergarzas.com/2015/01/automatizacion-pruebas.html] [Accedido: 21-3-2021]

[7] LogiGear Corporation, «TestArchitect Overview», [En linea, consultado en https://www.youtube.com/watch?v=E83ahFA4ZlY] [Accedido: 15-1-2021]

[8] Nguyen, Thuc, «12 Best Automation Tools for Desktop Apps in 2021», [En línea, consultado en https://www.logigear.com/blog/test-automation/12-best-automation-tools-for-desktop-apps-in-2021/#section1] [Accedido: 1-2-2021]

[9] Nick Abalov, «Winium – Now for Windows pone», [En línea, consultado en https://tech.badoo.com/en/article/281/winium-now-for-windows-phone/] [Accedido: 10-2-2021]

[10] Winnium Project, «Samples», [En línea, consultado en https://github.com/2gis/Winium] [Accedido:11-3-2021]

[11] Appium Project, «Appium Philosophy», [En línea, consutado en https://appium.io/docs/en/about-appium/intro/] [Accedido: 15-2-2021]

[12] Pauliedoherty, «The painful parts of end-to-end test automation for your Windows application», [En línea, consultado en https://dev.to/beanworks/the-painful-parts-of-end-to-end-test-automation-for-your-windows-application-1dde] [Accedido: 17-12-2020]

[13] Appium Project, «List of client libraries with Appium server support», [En línea, consultado en https://appium.io/docs/en/about-appium/appium-clients/index.html] [Accedido: 16-2-2021]

[14] Appium Github, «Samples», [En línea, consultado en

https://github.com/appium/appium] [Accedido: 15-2-2021]