



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



TFG del Grado en Ingeniería Informática

TestingAWE



Sistema para generar test
automáticos End-to-end de aplicaciones
Angular/Web/Electron

Presentado por Mario Ubierna San Mamés

en la Universidad de Burgos – 3 de julio de 2019

Tutor Universidad de Burgos: Dr. Raúl Marticorena Sánchez

Tutores Observatorio HP: Alicia Gago Pérez,

David Calvo Duarte,

Juan Manuel Zamarreno Pérez

Índice de Contenido

Índice de Contenido	3
Índice de tablas	5
Índice de ilustraciones	6
Apéndice A Manuales	8
A.1. Introducción	8
A.2. Planificación temporal	9
A.3. Estudio de viabilidad	16
Apéndice B Especificación de Requisitos	24
B.1. Introducción	24
B.2. Objetivos generales	25
B.3. Catálogo de requisitos	25
B.4. Especificación de requisitos	27
Apéndice C Especificación de Diseño	35
C.1. Introducción	35
C.2. Diseño de datos	35
C.3. Diseño procedimental	36
C.4. Diseño arquitectónico	38
Apéndice D Documentación técnica de programación	47
D.1. Introducción	47
D.2. Estructura de directorios	47

D.3. Manual del programador	48
D.4. Pruebas del sistema	55
Apéndice E Documentación de usuario.....	57
E.1. Introducción.....	57
E.2. Requisitos de usuarios.....	57
E.3. Instalación.....	57
E.4. Manual de usuario	59
Bibliografía.....	67

Índice de tablas

Tabla 1 - Estimación respecto story points	10
Tabla 2 - Costes de personal.....	16
Tabla 3 - Costes de hardware.....	17
Tabla 4 - Costes de software.....	17
Tabla 5 - Costes varios	18
Tabla 6 - Resumen costes	18
Tabla 7 - Beneficios	19
Tabla 8 - Licencias.....	21
Tabla 9 - Comparativa entre MIT y Apache-2.0.....	22
Tabla 10 - Resumen de las licencias	23
Tabla 11 - CU-01 Soporte de la librería.....	27
Tabla 12 - CU-02 Automatización de la toolkit	29
Tabla 13 - CU-03 Generación de ficheros	29
Tabla 14 - CU-04 Lectura de los ficheros HTML	30
Tabla 15 - CU-05 Extracción de los id	31
Tabla 16 - CU-06 Configuración de la aplicación	33
Tabla 17 - CU-07 Ayuda de la aplicación.....	34

Índice de ilustraciones

Ilustración 1 - Estructura interna para la obtención de los elementos HTML	35
Ilustración 2 - Diagrama de Secuencia Automatización.....	37
Ilustración 3 - Patrón MVC.....	38
Ilustración 4 - Patrón Singleton.....	39
Ilustración 5 - Patrón POM.....	40
Ilustración 6 - Diagrama de Componentes.....	41
Ilustración 7 - Interfaces libraryWIO.....	41
Ilustración 8 - Diagrama de clases de ClickAbstract	42
Ilustración 9 - Diagrama de clases de ValueAbstract	42
Ilustración 10 - Diagrama de clases de ValueClickAbstract	43
Ilustración 11 - Diagrama de clases de ValueSelectedClick	43
Ilustración 12 - Relación ClickAbstract con elementos HTML.....	44
Ilustración 13 - Relación ValueAbstract con elementos HTML.....	44
Ilustración 14 - Relación ValueClickAbstract con elementos HTML	45
Ilustración 15 - Relación ValueSelectedClickAbstract con elementos HTML.	45
Ilustración 16 - Relación directa de Root con elementos HTML.....	46
Ilustración 17 - Apertura de GitKraken.....	50
Ilustración 18 - Apertura ventana de clonación GitKraken.....	51
Ilustración 19 - Clonación del repositorio en GitKraken	51
Ilustración 20 - Ventana del proyecto en GitKraken.....	52
Ilustración 21 - Estructura inicial del proyecto.....	52
Ilustración 22 - Estructura final del proyecto.....	53

Ilustración 23 - Modificación del atributo main del nuevo proyecto	54
Ilustración 24 - Modificación del atributo scripts del nuevo proyecto.....	54
Ilustración 25 - Módulo TestingAWE en NPM	58
Ilustración 26 - Package.json inicial.....	60
Ilustración 27 - Atributo main actualizado.....	60
Ilustración 28 - Atributo test actualizado.....	60
Ilustración 29 - Atributo de automatización.....	61
Ilustración 30 - Resultado final del fichero package.json.....	61
Ilustración 31 - Mensaje inicial TestingAWE	62
Ilustración 32 - Aweconfig.json inicial	62
Ilustración 33 - Aweconfig.json final	63
Ilustración 34 - Proyecto Spectron antes de crear un test.....	64
Ilustración 35 - Proyecto Spectron después de crear un test	64
Ilustración 36 - Ejemplo test TestingAWE.....	65
Ilustración 37 - Ejecución del test.....	66

Apéndice A Manuales

A.1. Introducción

La fase de planificación es una de las más importantes en el desarrollo de cualquier proyecto, es aquí donde se estima el trabajo, el tiempo y el dinero que conlleva la realización del mismo. En este anexo se va a realizar el estudio de todo el proceso, enfocándonos principalmente en dos aspectos:

- La planificación temporal.
- Estudio de viabilidad.

Dentro del primer aspecto, se aplicó la metodología SCRUM para el desarrollo del proyecto, dada una fecha de comienzo se estimó la duración del mismo. Además, a cada *user story* se le asignaba una dificultad siguiendo la serie de Fibonacci, y dentro de éstas se definían las tareas, las cuales también se estimaban en un determinado número de horas.

Respecto al segundo apartado, se ha analizado dos aspectos importantes:

- **Viabilidad económica:** en ella se estima tanto los costes y beneficios que podemos obtener si realizamos el proyecto.
- **Viabilidad legal:** nos permite determinar si nuestro proyecto cumple o no con las leyes, es decir, aquellas leyes que están relacionadas con el software y el desarrollo del mismo.

A.2. Planificación temporal

Como bien he mencionado antes hemos seguido una de las metodologías ágiles más conocidas, SCRUM, para el desarrollo del proyecto.

Aunque no hemos aplicado la metodología SCRUM al cien por cien, hemos tratado de seguirla conceptualmente:

- El desarrollo fue iterativo, es decir, se utilizaron *sprints*.
- Fue incremental, ya que al final de cada *sprint* se entregaba una parte del producto.
- La duración del *sprint* fue de dos semanas.
- Dentro de un mismo *sprint* se realizaban tres reuniones, siendo la primera reunión dónde definíamos los objetivos de dicho *sprint*, en la segunda reunión era como una reunión diaria (explicaba lo que había hecho, los problemas que había tenido, y lo que iba a hacer hasta final del *sprint*), y la tercera reunión servía tanto para cerrar el *sprint*, como para indicar de nuevo los objetivos del siguiente *sprint*.
- Al comienzo del *sprint* definíamos las tareas que íbamos a realizar.
- Las tareas se estimaban dependiendo de la complejidad de las mismas.
- Se utilizó un sistema Kanban para la organización de las tareas dentro del *sprint*.

Aunque el desarrollo fue incremental, es verdad que realmente al principio no podíamos entregar una parte del producto, ya que nuestro proyecto no es algo visual, sin embargo, sí que explicaba y enseñaba lo que había hecho. Una vez pasado el ecuador del proyecto, sí que pudimos entregar una parte funcional del producto al final de cada *sprint*.

Para la parte de la librería, la metodología que seguimos fue la de probar y probar, ya que no podíamos hacer test unitarios sobre los métodos que habíamos hecho, sino que para poder probarlo teníamos que hacer uso de una aplicación Electron.

Por otro lado, para el desarrollo de la lógica de la generación de los ficheros sí que pudimos hacer test unitarios y de integración, es verdad que podíamos haber seguido la metodología TDD, sin embargo, decidimos que lo mejor era hacer primero el código y luego las pruebas, ya que la tecnología con la que estábamos trabajando

la desconocíamos, de esa forma pudimos “jugar” un poco con ella mientras realizábamos el código.

Para la estimación de las *user stories*, hemos usado los *story points* que proporciona Azure DevOps. En la siguiente tabla se muestra la estimación del tiempo respecto a los *story points*:

Story points	Estimación temporal
1	15 minutos
2	45 minutos
3	2 hora
5	5 horas
8	1 día
13	2 días
21	1 semana

Tabla 1 - Estimación respecto story points

En los siguientes puntos voy a explicar los diferentes *sprints* que hemos realizado, junto con la estimación respecto a lo que sucedió.

Sprint 0 (18/01/2019 – 03/02/2019)

Aunque anterior a esta fecha ya se hizo alguna reunión, fue a partir de este momento en el que comenzamos la realización de este proyecto, cabe mencionar que en este punto todavía no estábamos haciendo metodología ágil, sino que fueron dos semanas en las que se establecieron los diferentes objetivos que queríamos alcanzar.

En esta primera fase hay que destacar el hecho de que consistió principalmente en aprender a manejarnos con las nuevas tecnologías con las que íbamos a tener que trabajar, es decir, fue una fase de documentación, investigación, etc.

Como bien he mencionado, en este punto no estábamos haciendo SCRUM, por lo que no se estimó ningún *story point* para las *user stories*.

Sprint 1 (04/02/2019 – 18/02/2019)

En este punto ya comenzó el desarrollo a través de SCRUM, es decir, hacer uso de *story points* para estimar las tareas que se deseaban realizar en este *sprint*.

Dentro de este *sprint*, podemos encontrar dos fases:

- El arranque de una aplicación Angular/Electron, ésta fue Tour of Heroes, la cual está desarrollada en Angular, pero que posteriormente la tuvimos que convertir a una aplicación Electron.
- En la segunda fase, nos pusimos como objetivo realizar algunos test sobre la aplicación anterior, para así aprender a utilizar la librería WebdriverIO.

En este sprint se estimaron 18.5 horas, y finalmente se invirtieron 17.55 horas para completar todos los objetivos.

Sprint 2 (19/02/2019 – 25/02/2019)

Tal y como se puede apreciar en las fechas de este *sprint*, la duración no fue de dos semanas, esto se debe a que realmente fue una extensión del *sprint* anterior.

En este *sprint* también podemos apreciar dos fases:

- En la primera, nos pusimos como objetivo modificar el HTML de la aplicación Angular, ya que una buena práctica en el desarrollo de HTML es que todos los elementos tengan su id, el cual lógicamente tiene que ser único.
- En la segunda fase, el objetivo fue empezar la creación de la librería, es decir, probar que creándonos una clase sencilla llamada Button, a la cual en su constructor le pasábamos el id del elemento, podíamos utilizar los métodos que proporciona WebdriverIO para dicha etiqueta.

En este *sprint* se estimaron 5 horas, y finalmente se invirtieron 5 horas para alcanzar dichos objetivos.

Sprint 3 (26/02/2019 – 11/03/2019)

Este fue el último *sprint* que sirvió para el aprendizaje en el manejo de las diferentes tecnologías, ya que a partir del siguiente *sprint*, comenzó el desarrollo puro de la *toolkit* TestingAWE.

Dentro de este *sprint*, nos propusimos de nuevo dos objetivos (por norma general esto es así, ya que se buscaba alcanzar un objetivo una semana, y el otro en la segunda semana del *sprint*):

- El primer objetivo fue reimplementar el *mock*, es decir, ampliar la librería (crear la estructura de clases para los diferentes elementos HTML) para así poder hacer más pruebas sobre la aplicación Tour Of Heroes.
- El segundo objetivo fue reescribir los test, es decir, aquellos test que estaban escritos usando directamente WebdriverIO fueron reemplazados por la nueva librería, para así poder comprobar que todo funcionaba correctamente.

La estimación de este sprint fue de 14 horas, y realmente se invirtieron 11 horas para alcanzar los diferentes objetivos.

Sprint 4 (12/03/2019 – 25/03/2019)

En este *sprint* comenzó realmente el verdadero desarrollo de la *toolkit*. En este momento propuse la creación de un módulo NPM que contuviera toda la *toolkit*, de tal forma que el uso, y sobre todo la instalación, fuera muy simple.

Por lo tanto, el objetivo de este sprint fue crear inicialmente un módulo NPM, dicho módulo iba a estar de forma local en mi equipo hasta que no tuviera toda la funcionalidad de TestingAWE, ya que a partir de ese momento se publicaría.

Al ser una idea propuesta por mí y no por el grupo de HP, tuve que investigar y documentarme sobre el cómo lo podía hacer.

En este *sprint* se estimaron 9 horas (sin tener en cuenta la fase de investigación), y finalmente se realizaron 12 horas para alcanzar el objetivo.

Sprint 5 (26/03/2019 – 08/04/2019)

Este *sprint* fue uno de los más importantes ya que introdujo la base de la *toolkit*. Al igual que en el *sprint* anterior, fue una fase más de documentación, ya que se probaron distintas tecnologías para parsear los ficheros HTML.

En esta fase, nos propusimos dos objetivos:

- Investigar cuál era el módulo NPM más adecuado para parsear los ficheros HTML de nuestra aplicación Angular/Electron.
- Aplicar de forma correcta el módulo elegido, para así poder extraer los id de cada uno de los elementos que contienen los ficheros HTML.

Ambos objetivos se caracterizaron por ser costosos, es decir, por invertir una gran cantidad de tiempo.

Es por ello que la estimación fue de 45 horas, y el tiempo invertido para alcanzar dichos objetivos fueron también de 45 horas.

Sprint 6 (09/04/2019 – 22/04/2019)

Aunque la fecha final es el 22/04/2019, realmente este *sprint* terminó el 03/05/2019, esto se debe a que fue semana santa, y el tutor que solía manejar la parte de Azure DevOps se marchó de vacaciones.

Este *sprint* fue muy importante, ya que fue un momento crítico en el desarrollo del proyecto. Esto se debe a que en Angular puede haber elementos cuyos id se calculen de forma dinámica, por lo tanto se tuvo que hacer el desarrollo para identificar cuando un elemento tiene un id estático o dinámico.

Este *sprint* también lo podemos dividir en dos fases:

- Identificar y crear el código automático para los id dinámicos, es decir, manejar de forma diferente si el elemento es estático o dinámico.
- Crear la automatización para la generación de los ficheros.

El tiempo estimado en este sprint fue de 47 horas, y finalmente fueron 40 horas las invertidas.

Sprint 7 (06/05/2019 – 19/05/2019)

Este *sprint* se centró principalmente en dos objetivos:

- Crear la configuración necesaria para la automatización, es decir, la creación del fichero “aweconfig.json”.
- Crear la automatización de toda la aplicación, es decir, que ejecutando un solo comando se realice toda la generación de los ficheros.

En este *sprint* se estimaron 15 horas, y finalmente se invirtieron 13 horas para alcanzar los objetivos.

Sprint 8 (21/05/2019 – 01/06/2019)

Aunque la fecha de finalización de este *sprint* fue el 01/06/2019, realmente no terminó hasta la entrega del proyecto, es decir, hasta el 02/07/2019. Esto se debe a que el desarrollo de la *toolkit* ya había finalizado, sin embargo, quedaba de realizar la documentación y mejora del código.

En este *sprint* se estimaron los siguientes objetivos:

- Mejorar el código, es decir, eliminar código duplicado.
- Aumentar el soporte de la librería, permitir que TestingAWE reconociera más elementos HTML.
- Realizar la documentación del proyecto.
- Publicar la *toolkit* en NPM.
- Realizar el diseño gráfico, vídeos, póster, etc.

En este *sprint* se estimaron 88 horas, y finalmente se invirtieron 96 horas para alcanzar todos los objetivos.

Resumen

Como se puede apreciar en los *sprints* comentados en el punto anterior, por norma general sí que hay una buena aproximación entre el tiempo esperado y el tiempo necesario para alcanzar dichos objetivos.

Cabe destacar que inicialmente se estimó terminar el proyecto en la segunda semana de junio, es decir, entre el 03/06/2019 y el 09/06/2019, pero como suele pasar el desarrollo se retrasó dos semanas más, las mejoras tanto del código como de la documentación supuso que el proyecto finalizase la última semana de junio, es decir, entre el 24/06/2019 y el 30/06/2019.

A.3. Estudio de viabilidad

Viabilidad económica

En este apartado se va a estudiar y explicar los costes y beneficios que supondrían si el proyecto en vez de ser de ámbito educativo fuera empresarial.

Costes

Los costes relacionados con un proyecto de este ámbito los podemos clasificar en cuatro grupos: personal, hardware, software y varios.

Costes de personal:

El proyecto ha sido desarrollado por una única persona, aunque el tiempo del proyecto como tal ha sido de 250 horas aproximadamente, entre la fase previa (al comienzo no se definió la metodología SCRUM, no se tuvo en cuenta las horas) y las fases de documentación iniciales (las cuales tampoco están reflejadas en Azure DevOps), estamos hablando de que aproximadamente 350 horas han sido las invertidas para alcanzar los objetivos propuestos.

En resumidas palabras, el desarrollo lo ha hecho un programador a media jornada durante cinco meses. Es por ello que se considera el siguiente salario:

Concepto	Coste
Salario Mensual Neto	914.17 €
IRPF (15%)	161.33 €
Seguridad Social (28.3%)	424.5 €
Salario Mensual Bruto	1500 €
Total 5 meses	7500 €

Tabla 2 - Costes de personal

Costes de hardware:

Respecto a estos costes, son aquellos elementos *hardware* necesarios para la realización del proyecto, se considera que la amortización se realiza en 5 años y han pasado 5 meses.

Concepto	Coste	Amortizado
Ordenador Portátil	999 €	83.25 €
Total	999 €	83.25 €

Tabla 3 - Costes de hardware

Costes de software:

Costes que suponen el uso de licencias *software* para el desarrollo del proyecto. Al igual que sucedía en el coste anterior, se consideran que se realiza una amortización en 5 años y ya han pasado 5 meses.

Concepto	Coste	Amortización
Windows 10 Pro	259 €	21.58 €
Total	259 €	21.58 €

Tabla 4 - Costes de software

Costes varios

En este punto tenemos en cuenta aquellos costes que sean diferentes de los anteriores.

Concepto	Coste
Memoria impresa	30 €
Póster	12 €
Alquiler de oficina (piso)	700 €
Internet	125 €
Total	867 €

Tabla 5 - Costes varios

Resumen Costes

Por lo tanto, el coste total que supone la realización de este proyecto es el siguiente:

Concepto	Coste
Personal	7500 €
Hardware	83.25 €
Software	21.58 €
Varios	867 €
Total	8471.83 €

Tabla 6 - Resumen costes

Beneficios

La *toolkit* será distribuida de forma gratuita y sin publicidad, es por ello que a corto plazo no vamos a obtener ninguna ganancia.

El modelo por el cual obtendremos beneficio es a partir del desarrollo de una nueva versión, en la cual vamos a añadir dos funcionalidades:

- Ampliación de la librería, es decir, proporcionar un mayor soporte, ya que con WebdriverIO no todo se puede realizar.
- La creación de una sintaxis, es decir, que a partir de un lenguaje natural se pueda realizar los test de una forma más sencilla, haciendo uso de la herramienta creada en este proyecto.

Para ello se considera una suscripción como el método más adecuado para ingresar dinero de forma continua. Es por ello, que habrá tres tipos de suscripciones:

Tipo	Ampliación librería	Sintaxis natural	Precio
Rookie	No	No	Gratis
Amateur	Sí	No	1 € por mes
Profesional	Sí	Sí	5 € por mes

Tabla 7 - Beneficios

A la vista de la tabla anterior, si solo hubiera licencias *Amateur* necesitaríamos un total de 8471 licencias. Por otro lado, si todos los usuarios pagaran por una suscripción Profesional, necesitaríamos 1695 licencias.

En resumen, alcanzar ambos objetivos son posibles, ya que las cifras de licencias no son desorbitadas.

Viabilidad Legal

En este apartado se estudiará y analizará las diferentes licencias con las que podemos liberar nuestra *toolkit*. Lo primero que debemos de tener en cuenta es, ¿qué es una licencia?

“*Una licencia de software es un contrato entre el licenciante (autor/titular de los derechos de explotación/distribución) y el licenciatario (usuario consumidor,*

profesional o empresa) del programa informático, para utilizarlo cumpliendo una serie de términos y condiciones establecidas dentro de sus cláusulas, es decir, es un conjunto de permisos que un desarrollador le puede otorgar a un usuario en los que tiene la posibilidad de distribuir, usar o modificar el producto bajo una licencia determinada.” [1]

Software

A la hora de liberar nuestra aplicación bajo una licencia, debemos analizar antes las diferentes posibilidades que nos proporcionan las librerías que hemos utilizado para el desarrollo del proyecto, ya que alguna de éstas puede tener determinadas restricciones.

Por lo tanto, en la siguiente tabla se indica las diferentes licencias que poseen las librerías usadas:

Dependencia	Versión	Descripción	Licencia
Chai	4.2.0	Librería de aserciones para Node.	MIT
Electron	4.0.3	Framework que permite la creación de aplicaciones multiplataforma usando JavaScript/TypeScript, HTML y CSS	MIT
Fs-Extra	7.0.1	Librería que contiene métodos de sistema de archivos, que no están incluidos en el módulo fs.	MIT
Htmlparser2	3.10.1	Librería que permite realizar el parseo de ficheros HTML.	MIT
Mocha	5.2.0	Marco de pruebas para JavaScript/TypeScript.	MIT
Spectron	5.0.0	Framework para escribir tests de integración en aplicaciones Electron.	MIT

Ts-node	8.1.1	Librería que permite hacer uso de MIT ficheros TypeScript en Node.
TypeScript	3.4.5	Librería que permite hacer uso del Apache-2.0 lenguaje de TypeScript.
WebdriverIO	4.13.0	API que nos permite realizar peticiones MIT a un servidor Selenium mediante WebDriver Wire Protocol

Tabla 8 - Licencias

Como podemos ver en la tabla anterior, la gran mayoría de licencias son MIT, excepto TypeScript que tiene licencia Apache-2.0.

Para poder elegir la licencia más adecuada, nos hemos basado en una que nos permita dar mayor libertad, sin embargo, antes de poder seleccionar la licencia, tenemos que hacer un estudio entre MIT y Apache-2.0.

La licencia MIT es perfecta siempre y cuando se quiera llegar al mayor número de desarrolladores, no es una licencia para software libre “total” ya que no nos permite garantizar la libertad de la aplicación, sino que solo exige que los derechos de autor sean incluidos en todas las copias del software [2] [3].

Por otro lado, la licencia Apache-2.0 es bastante similar a la licencia MIT, es una licencia permisiva ya que no obliga a que los trabajos derivados tengan la misma licencia. La principal diferencia entre Apache-2.0 y MIT es que la primera obliga a que los cambios que haya en la versión original tienen que estar reflejados en el código fuente, es decir, los derechos de autor han de estar tanto en el código fuente como en los binarios [3] [4].

Resumen de comparativa entre MIT y Apache-2.0:

Característica	MIT	Apache-2.0
Compatible con DFSG [5]	✓	✓
Compatible con GPL [6]	✓	✓

Aprobado por FSF [7]	✓	✓
Aprobado por OSI [8]	✓	✓
Copyleft [9]	✗	✗
Utilizable con otras licencias	✓	✓
Obligado mantenimiento de cambios	✗	✓

Tabla 9 - Comparativa entre MIT y Apache-2.0

Como nuestro objetivo era dar la mayor visibilidad posible a los desarrolladores, además la librería de TypeScript (la cual tiene licencia Apache-2.0) nos permite hacer uso de la misma y publicar bajo otro tipo de licencia, por otro lado, no hemos hecho ni se harán modificaciones sobre esta librería, hemos decidido que la licencia con la que vamos a publicar el proyecto va a ser una licencia MIT.

Documentación

Aunque hay muchos tipos de licencias, para la parte de la documentación hemos elegido la licencia *Creative Commons*, las cuales suelen ser las más habituales para la documentación de un proyecto. La versión más utilizada a día de hoy es *Creative Commons Attribution 4.0 Internacional (CC-BY-4.0)* [10].

Esta licencia establece los siguientes puntos:

- Compartir: se puede copiar y redistribuir el material en cualquier medio o formato.
- Adaptar: se puede remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.

Resumen

Para concluir con este apartado, las licencias seleccionadas han sido las siguientes:

Recurso	Licencia
Proyecto software	MIT
Documentación	CC-BY-4.0

Tabla 10 - Resumen de las licencias

Apéndice B Especificación de Requisitos

B.1. Introducción

En este apéndice se va a realizar el estudio de los requisitos, los cuales definen el comportamiento de nuestra aplicación. La especificación de requisitos es una parte fundamental en el desarrollo del software, ya que en ella participa tanto el usuario final como el equipo de desarrollo. Una vez finalizada esta fase, en un documento se tiene que describir lo que la aplicación deber hacer [11].

Los principales objetivos que se deben de cumplir en la especificación de requisitos según el estándar IEEE 830-1998 son: [12]

- **Correcto:** un requisito es correcto si éste describe alguna necesidad real.
- **No ambiguo:** un requisito es no ambiguo siempre y cuando éste solo tenga una interpretación.
- **Completo:** la especificación debe de incluir todos los requerimientos de la aplicación.
- **Verificable:** tiene que existir algún proceso no muy costoso, que nos permita determinar si la aplicación satisface el requerimiento.
- **Consistente:** siempre y cuando los requerimientos no sean contradictorios.
- **Clasificable:** los requisitos se tienen que poder clasificar bajo uno o varios criterios.
- **Modificable:** la especificación es modificable siempre y cuando ésta nos permite cambiar cualquier requisito de una forma fácil, garantizando el resto de los objetivos.
- **Explorable:** si se puede determinar la historia, o aplicación de un requisito.

- **Utilizable:** se deben de tener en cuenta las necesidades del mantenimiento de los requisitos.

B.2. Objetivos generales

Los principales objetivos que hemos buscado con el desarrollo de este proyecto son los siguientes:

- Desarrollar una *toolkit* para ordenadores que nos permita realizar test *end-to-end* en aplicaciones Angular/Web/Electron.
- Facilitar la sintaxis de test, a partir de una librería más simple.
- Realizar la automatización de la aplicación, es decir, usar la *toolkit* para facilitar la creación de test.
- Facilitar la instalación de nuestra *toolkit* a través de un módulo NPM.

B.3. Catálogo de requisitos

En este apartado se van a detallar cada uno de los requisitos funcionales y no funcionales que han de cumplir nuestra *toolkit*.

Requisitos Funcionales

- **RF-1 Soporte de la librería:** el usuario tiene que ser capaz de hacer uso de la librería para la sintaxis de los test.
- **RF-2 Automatización de la toolkit:** la aplicación tiene que ser capaz de proporcionar todos los elementos necesarios para la creación de test.
 - **RF-2.1 Generación de ficheros:** la aplicación tiene que ser capaz de generar los ficheros que contienen los elementos HTML sobre los que se van a hacer pruebas.
- **RF-3 Lectura de los ficheros HTML:** la aplicación tiene que ser capaz de poder leer los ficheros HTML de una aplicación Angular/Electron.
 - **RF-3.1 Extracción de los id:** la aplicación tiene que ser capaz de determinar si un elemento HTML es estático o dinámico.
- **RF-4 Configuración de la aplicación:** el usuario debe poder configurar todos los parámetros que están disponibles en el fichero de configuración.

- **RF-5 Ayuda de la aplicación:** el usuario tiene que poder obtener ayuda (información) sobre el mal funcionamiento de la aplicación.

Requisitos No Funcionales

- **RNF-1 Usabilidad:** la aplicación tiene que ser fácil de usar, en otras palabras, ha de ser intuitiva.
- **RNF-2 Escalabilidad:** la aplicación debe permitir que se añadan nuevas funcionalidades.
- **RNF-3 Rendimiento:** la aplicación debe tener unos tiempos de automatización aceptables, nunca se puede permitir que la aplicación deje de funcionar.
- **RNF-4 Disponibilidad:** la aplicación debe estar disponible para su uso en cualquier momento.
- **RNF-5 Mantenibilidad:** la aplicación debe permitir que sea fácil su mantenimiento, la cual nos permita garantizar el correcto funcionamiento de la misma, la escalabilidad, etc.
- **RNF-6 Soporte:** la aplicación debe dar soporte para todos los sistemas operativos Windows que permitan el uso de Node y Angular/Electron.
- **RNF-7 Internacionalización:** la aplicación tiene que poder usarse independientemente de la configuración del idioma del sistema operativo.

B.4. Especificación de requisitos

En este apartado se va a realizar un estudio sobre los diferentes casos de uso, todos estos casos vienen determinados por las *user stories*, las cuales se encuentran en el proyecto de Azure DevOps [13].

Casos de uso

CU-01	Soporte de la librería
Versión	1.0
Autor	Mario Ubierna San Mamés
Requisitos asociados	RF-1
Descripción	El usuario tiene que ser capaz de hacer uso de la librería para la sintaxis de los test.
Precondición	La <i>toolkit</i> se encuentra disponible.
Acciones	<p>1º El usuario ejecuta el comando para la generación de los ficheros.</p> <p>2º Se genera un archivo por cada fichero HTML que se encuentre en el proyecto Angular/Electron.</p> <p>3º Por cada elemento HTML identificado vamos a poder ejecutar una serie de métodos.</p>
Postcondición	Salida esperada depende del método que se vaya a ejecutar.
Excepciones	Errores lanzados por WebdriverIO
Importancia	Alta

Tabla 11 - CU-01 Soporte de la librería

CU-02	Automatización de la toolkit
Versión	1.0
Autor	Mario Ubierna San Mamés
Requisitos asociados	RF-2, RF-2.1
Descripción	La aplicación tiene que ser capaz de proporcionar todos los elementos necesarios para la creación de test.
Precondición	La <i>toolkit</i> se encuentra disponible.
Acciones	<p>1º El usuario ejecuta el comando para la generación de los ficheros.</p> <p>2º Se genera un archivo del tipo “awe.ts” por cada fichero HTML que se encuentre en el proyecto Angular/Electron.</p> <p>3º Si no hay ningún error, la generación de estos ficheros se habrá realizado de forma correcta, y contendrán los elementos HTML sobre los que se pueden hacer pruebas.</p>
Postcondición	Se genera un fichero del tipo “awe.ts” por cada fichero HTML que encontramos en el proyecto Angular/Electron.
Excepciones	<ul style="list-style-type: none">• ERROR: compruebe que es un proyecto Electron válido.• ERROR: compruebe que es un proyecto Spectron válido.• ERROR: compruebe que ambos proyectos están en el mismo directorio.• ERROR: el directorio no existe.• Errores lanzados por los módulos FS y FS-EXTRA.

Importancia	Alta
--------------------	------

Tabla 12 - CU-02 Automatización de la toolkit

CU-03 Generación de ficheros	
Versión	1.0
Autor	Mario Ubierna San Mamés
Requisitos asociados	RF-2.1
Descripción	La aplicación tiene que ser capaz de generar los ficheros que contienen los elementos HTML sobre los que se van a hacer pruebas.
Precondición	La <i>toolkit</i> se encuentra disponible.
Acciones	<p>1º El usuario ejecuta el comando para la generación de los ficheros.</p> <p>2º Se genera un fichero del tipo “awe.ts” por cada fichero HTML que se encuentre en el proyecto Angular/Electron.</p> <p>3º Si no hay ningún error, la generación de estos ficheros se habrá realizado de forma correcta.</p>
Postcondición	Se genera un fichero del tipo awe.ts por cada fichero HTML que encontramos en el proyecto Angular/Electron.
Excepciones	Errores lanzados por los módulos FS y FS-EXTRA.
Importancia	Alta

Tabla 13 - CU-03 Generación de ficheros

CU-04	Lectura de los ficheros HTML
Versión	1.0
Autor	Mario Ubierna San Mamés
Requisitos asociados	RF-3, RF-3.1
Descripción	La aplicación tiene que ser capaz de poder leer los ficheros HTML de una aplicación Angular/Electron.
Precondición	La <i>toolkit</i> se encuentra disponible.
Acciones	<p>1º El usuario ejecuta el comando para la generación de los ficheros.</p> <p>2º Por cada fichero HTML que se encuentra en el proyecto Angular/Electron se lee.</p> <p>3º Si no hay ningún error, el proceso se habrá realizado de forma correcta.</p>
Postcondición	-
Excepciones	Errores lanzados por los módulos FS.
Importancia	Alta

Tabla 14 - CU-04 Lectura de los ficheros HTML

CU-05	Extracción de los id
Versión	1.0
Autor	Mario Ubierna San Mamés
Requisitos asociados	RF-3.1
Descripción	La aplicación tiene que ser capaz de determinar si un elemento HTML es estático o dinámico.
Precondición	La <i>toolkit</i> se encuentra disponible.
Acciones	<p>1º El usuario ejecuta el comando para la generación de los ficheros.</p> <p>2º Por cada fichero HTML que se encuentra en el proyecto Angular/Electron, se lee y a la vez que se está leyendo se determina si el elemento HTML es estático o dinámico.</p> <p>3º Si no hay ningún error, el proceso se habrá realizado de forma correcta.</p>
Postcondición	En la estructura de datos correspondiente, quedan almacenados cada uno de los elementos HTML junto con su id, independientemente de si su id es estático o dinámico. Además, si un elemento HTML no tiene id o ese id es incorrecto no quedará almacenado.
Excepciones	Errores lanzados por los módulos FS.
Importancia	Alta

Tabla 15 - CU-05 Extracción de los id

CU-06	Configuración de la aplicación
Versión	1.0
Autor	Mario Ubierna San Mamés
Requisitos asociados	RF-4
Descripción	El usuario debe poder configurar todos los parámetros que están disponibles en el fichero de configuración.
Precondición	La <i>toolkit</i> se encuentra disponible.
Acciones	<p>1º De forma opcional el usuario ejecuta el comando para la generación de los ficheros si no ha creado el fichero de configuración “aweconfig.json”, ya que de lo contrario no hace falta.</p> <p>2º Si el fichero de configuración “aweconfig.json” no está creado entonces lo genera, de lo contrario comienza con la automatización.</p> <p>3º El usuario tiene que cambiar los parámetros de configuración respecto a las rutas del proyecto Angular/Electron y del proyecto Spectron.</p> <p>5º De forma opcional podrá cambiar el parámetro “ignore”.</p> <p>4º Si no hay ningún error, el proceso se habrá realizado de forma correcta.</p>
Postcondición	-
Excepciones	<ul style="list-style-type: none"> • Errores lanzados por los módulos FS y FS-EXTRA. • ERROR: no está definida la propiedad electronProject.

- ERROR: no está definida la propiedad spectronProject.

Importancia	Alta
--------------------	------

Tabla 16 - CU-06 Configuración de la aplicación

CU-07	Ayuda de la aplicación
Versión	1.0
Autor	Mario Ubierna San Mamés
Requisitos asociados	RF-5
Descripción	El usuario tiene que poder obtener ayuda (información) sobre el mal funcionamiento de la aplicación.
Precondición	La <i>toolkit</i> se encuentra disponible.
Acciones	<p>1º El usuario ejecuta el comando para la generación de los ficheros.</p> <p>2º Si no hay ningún error, el proceso se habrá realizado de forma correcta, de lo contrario se mostrará información al usuario sobre el error.</p> <p>3º De forma opcional, podrá leer el README.md para obtener más información sobre la instalación y funcionamiento de la <i>toolkit</i>.</p>
Postcondición	-
Excepciones	<ul style="list-style-type: none"> • Errores lanzados por los módulos FS y FS-EXTRA.

- ERROR: no está definida la propiedad electronProject.
- ERROR: no está definida la propiedad spectronProject.
- ERROR: compruebe que es un proyecto Electron válido.
- ERROR: compruebe que es un proyecto Spectron válido.
- ERROR: compruebe que ambos proyectos están en el mismo directorio.
- ERROR: el directorio no existe.

Importancia	Alta
--------------------	------

Tabla 17 - CU-07 Ayuda de la aplicación

Apéndice C Especificación de Diseño

C.1. Introducción

En este apéndice se va a estudiar y analizar los distintos diseños de la *toolkit*, es decir, el diseño de datos, el procedimental y el diseño arquitectónico.

C.2. Diseño de datos

Aunque en TestingAWE sí que realizamos un diseño de datos “interno”, es decir, la información que se proporciona a la *toolkit* ha de tener una determina estructura, por ejemplo, la obtención de los elementos HTML que se encuentran en una determina página se realiza a través de un array de arrays, en el cual guardamos el tag del elemento junto con su id, tal y como podemos ver en la siguiente ilustración.

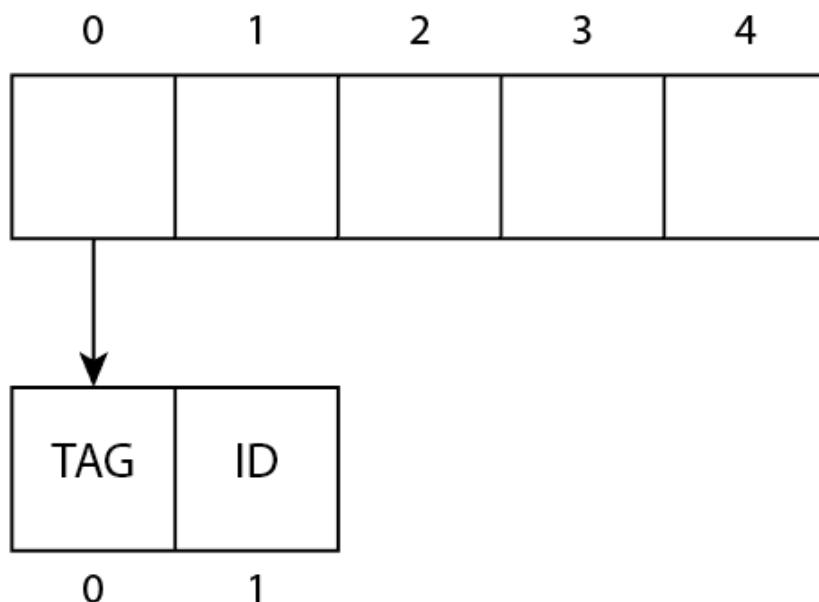


Ilustración 1 - Estructura interna para la obtención de los elementos HTML

Sin embargo, este trabajo de fin de grado presenta la característica que no posee un diseño de base datos, es decir, no es necesario el almacenamiento de información para el funcionamiento de la *toolkit*, ya que todo lo que necesitamos es el fichero de configuración inicial “aweconfig.json” y la aplicación Angular/Electron (dando por supuesto que ya está instalado el módulo TestingAWE en nuestro proyecto).

C.3. Diseño procedimental

En este apartado vamos a realizar el estudio sobre el funcionamiento de nuestra aplicación a través de un diagrama de secuencia.

En este diagrama se ha representado el cómo se comunican las partes que componen la *toolkit* para la generación de los ficheros. Para que este diagrama tenga sentido se da por supuesto que el proyecto en el que está instalado el módulo TestingAWE está configurado correctamente, y además poseemos una aplicación Angular/Electron sobre la que ejecutar nuestra *toolkit*.

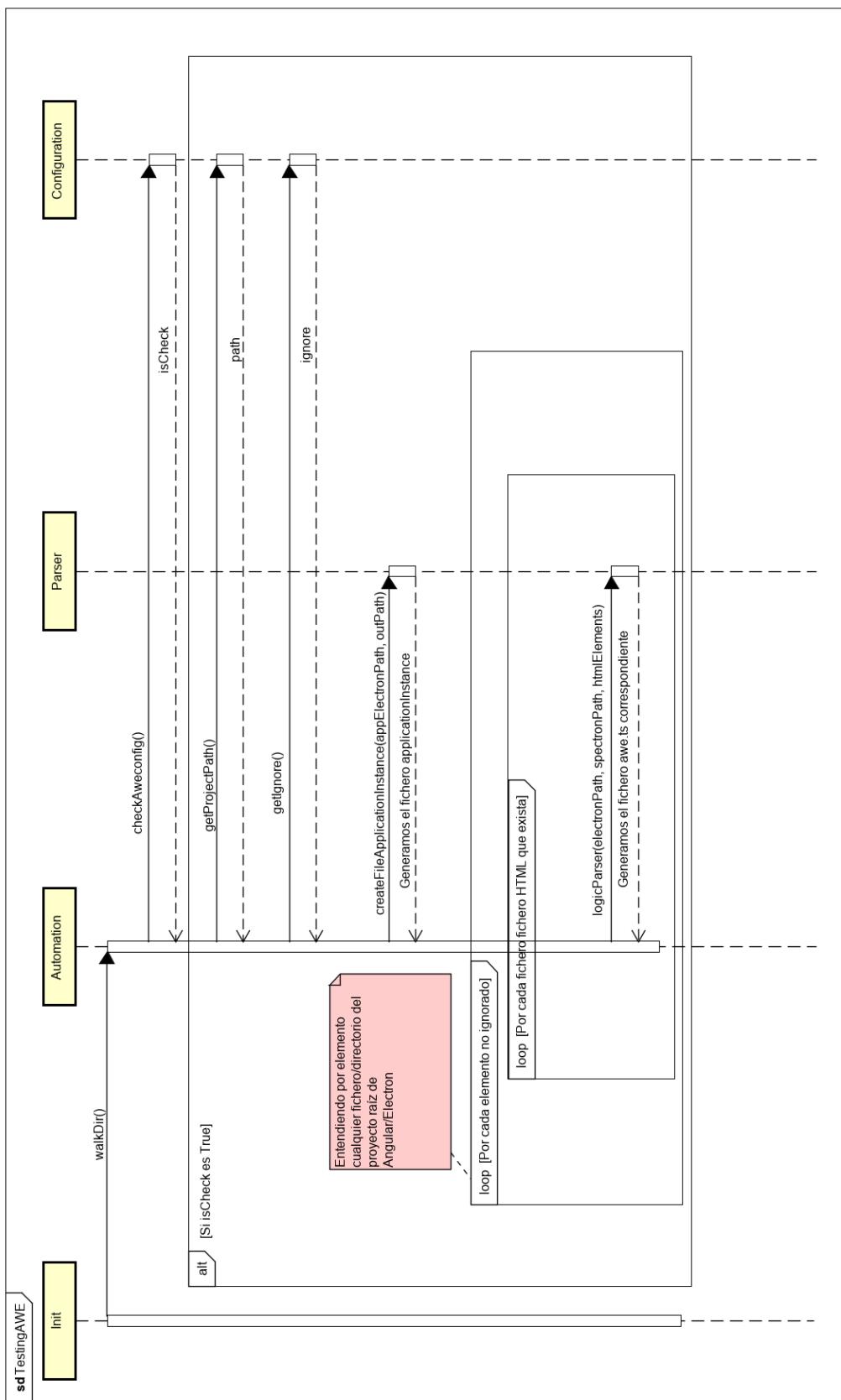


Ilustración 2 - Diagrama de Secuencia Automatización

C.4. Diseño arquitectónico

Para realizar el desarrollo software de esta aplicación hemos tenido que seguir tres patrones de diseño:

- Patrones arquitectónicos: Modelo-Vista-Controlador (MVC) y el patrón Singleton.
- Patrón de automatización: Page-Object-Model (POM)

Modelo-Vista-Controlador (MVC)

El Modelo-Vista-Controlador es un patrón de diseño software, el cual nos permite separar los datos y la lógica de la aplicación de la lógica de la vista [14] [15]. Este patrón tiene tres conceptos:

- Modelo: se encarga del manejo de los datos, aunque su principal uso suele ser acceder a la base de datos, no tiene por qué, como es en nuestro caso.
- Controlador: es el intermediario, ya que se encarga de pedir los datos al modelo y de dárselos a la vista.
- Vista: no es más que la representación de los datos, es decir, la parte gráfica.

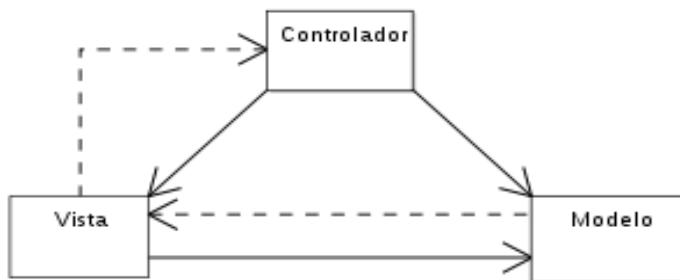


Ilustración 3 - Patrón MVC

Patrón Singleton

El patrón de diseño Singleton es uno de los patrones más sencillos y usados del desarrollo software, tiene como objetivo garantizar que solo hay una única instancia [16]. Para poder aplicar dicho patrón debemos realizar los siguientes puntos:

- El constructor de la clase tiene que ser privado, para asegurarnos que solo puede haber una instancia de la aplicación.
- Tiene que tener un atributo privado y estático, ya que éste va a ser el que vamos a utilizar para tener una única instancia.
- La clase debe tener un método público y estático, en el cual vamos a generar una instancia o a devolver la instancia que ya hay.

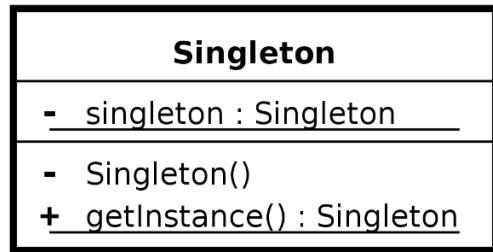


Ilustración 4 - Patrón Singleton

Dicho patrón ha sido utilizado para garantizarnos que solo se creaba una única instancia de Spectron cuando se iban a ejecutar los test, es decir, una vez que han sido generados los ficheros que contienen los elementos HTML, hemos tenido que crear una única instancia de Spectron para poder realizar pruebas sobre esos elementos HTML.

Page-Object-Model (POM)

Aunque este patrón no es arquitectónico en cuanto al desarrollo de la aplicación, sí que ha sido utilizado para la automatización, es decir, para la generación de los ficheros.

El patrón POM nos permite representar cada una de las pantallas que componen el sitio web o la aplicación que nos interesa probar, como un conjunto de objetos que encapsulan las características y funcionalidades representadas en la página [17] [18].

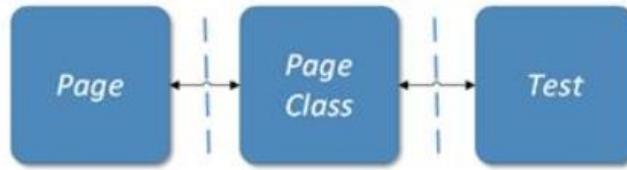


Ilustración 5 - Patrón POM

Este patrón lo que nos permite definir es la arquitectura de los ficheros que genera TestingAWE, ya que ésta genera un fichero por cada HTML que encuentra y dicho fichero contiene todos los elementos que hay dentro del mismo.

Diseño de estructura

Aunque en este proyecto no existen paquetes como tal, es verdad que sí que hay una lógica estructural dentro de TestingAWE.

Podemos dividir la lógica en cuatro componentes:

- *Componente aut*: agrupa toda la lógica respecto a la automatización del proyecto.
- *Componente config*: agrupa toda la lógica respecto a la configuración del proyecto, es decir, todo lo relacionado con el fichero de configuración “aweconfig.json”.
- *Componente libraryWIO*: agrupa la lógica respecto a la librería de WebdriverIO, este componente se puede dividir a su vez en tres grupos.
 - *abstractClass*: en ella se encuentran las clases abstractas de la librería.
 - *class*: en ella se encuentran todos los elementos HTML que da soporte la aplicación.
 - *interfaces*: dentro se encuentran las interfaces de la librería.
- *Componente parser*: agrupa toda la lógica respecto a la lectura de los ficheros HTML.

Los componentes aut, config y parser son los que permiten la automatización, es decir, la generación de los ficheros. Sin embargo, el componente libraryWIO es independiente del resto de componentes.

El siguiente diagrama de componentes muestra las dependencias entre aut, config y parser.

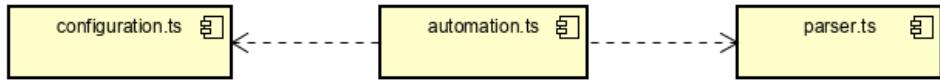


Ilustración 6 - Diagrama de Componentes

Diagrama de clases

Respecto al componente libraryWIO como ya hemos mencionado antes, tenemos las siguientes interfaces:

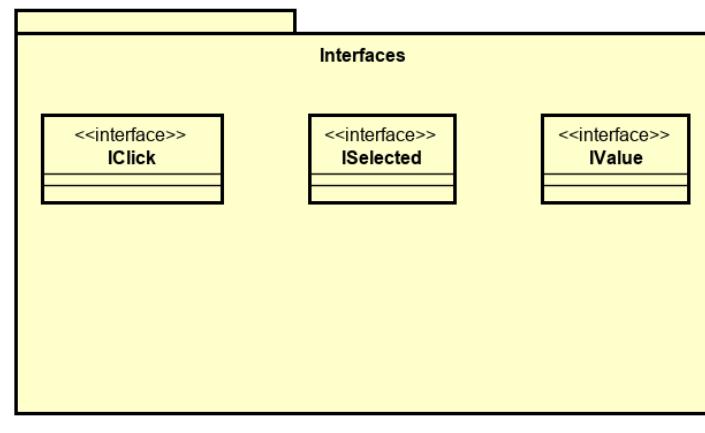


Ilustración 7 - Interfaces libraryWIO

Como bien hemos indicado, aunque no es un paquete como tal, sí que tiene una estructura equivalente. Esto mismo sucede con el resto de los componentes de libraryWIO.

Los siguientes diagramas reflejan la estructura arquitectónica entre las interfaces y las clases abstractas:

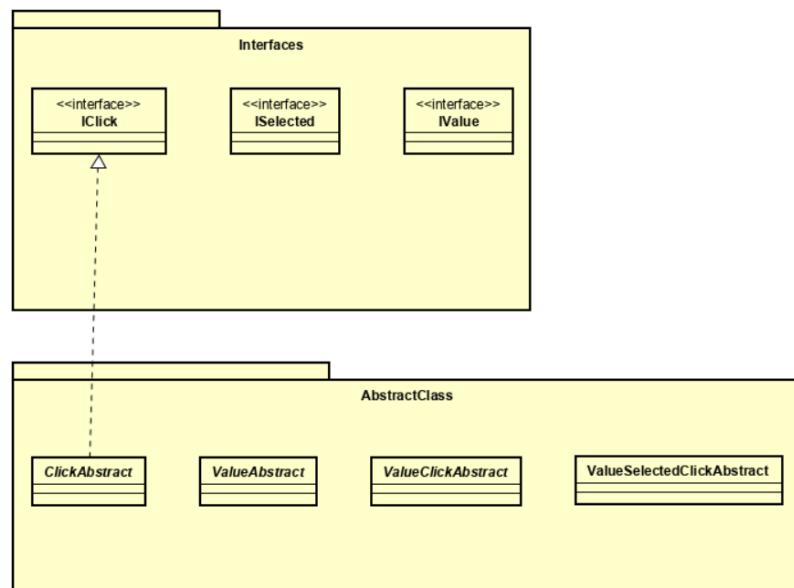


Ilustración 8 - Diagrama de clases de *ClickAbstract*

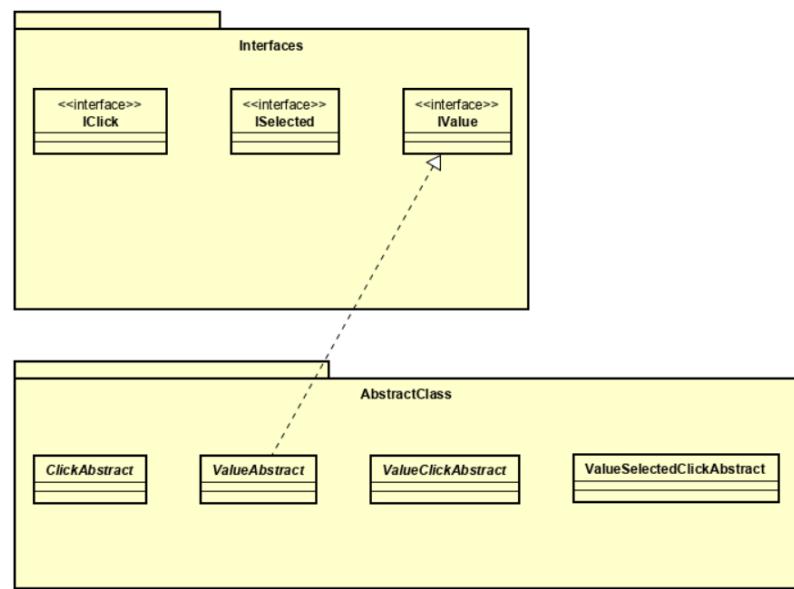


Ilustración 9 - Diagrama de clases de *ValueAbstract*

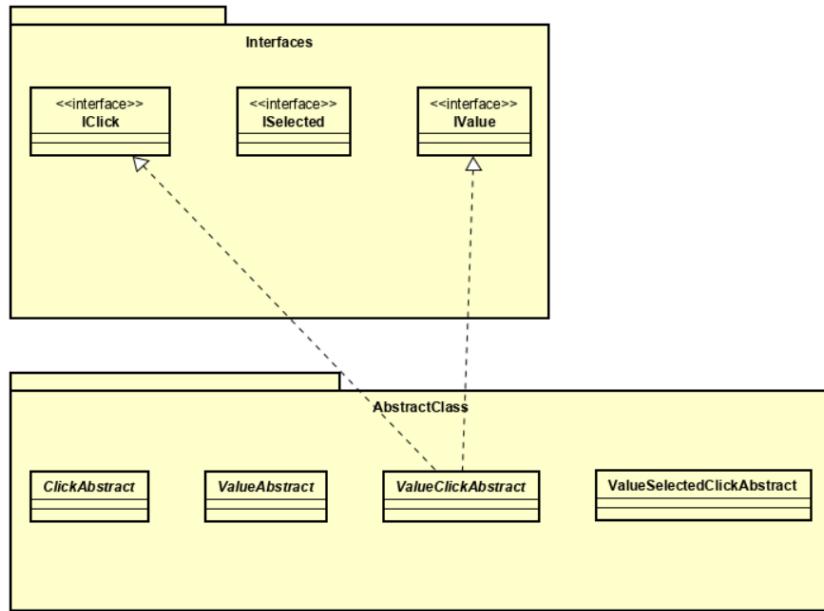


Ilustración 10 - Diagrama de clases de *ValueClickAbstract*

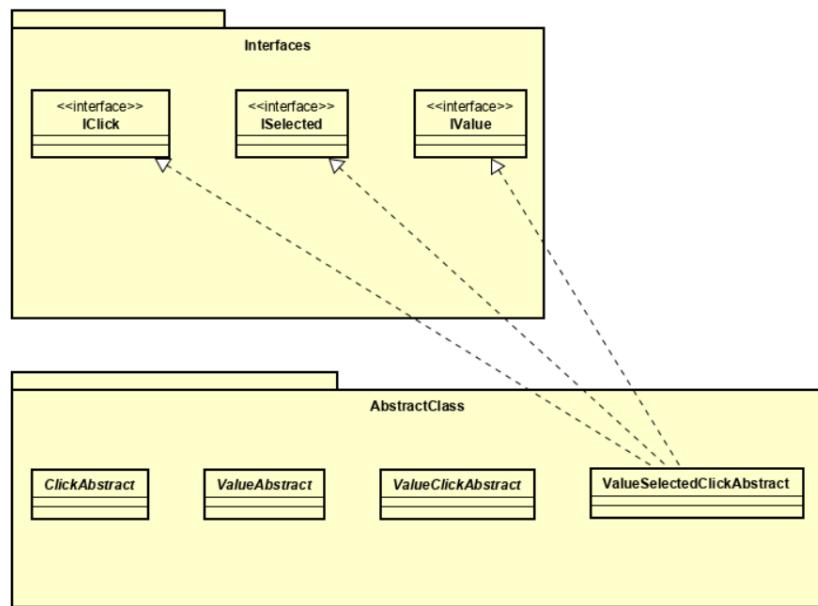


Ilustración 11 - Diagrama de clases de *ValueSelectedClick*

En las siguientes ilustraciones podemos observar el cómo se relacionan las clases abstractas, con las clases HTML que da soporte nuestra librería. Cabe mencionar que solo se muestran los diferentes tipos de relaciones, ya que el resto se pueden representar como alguna de éstas.

Por otro lado, vemos que todas las clases heredan de *Root*, o bien directamente o a través de una clase abstracta. Esto se debe a que *Root* posee todos los métodos genéricos de la librería.

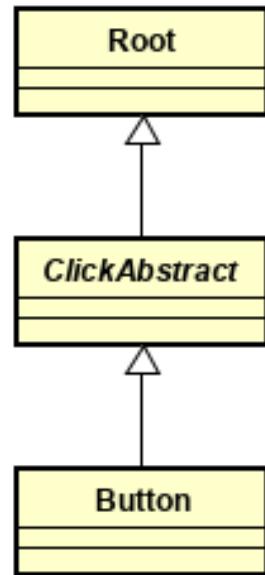


Ilustración 12 - Relación *ClickAbstract* con elementos HTML

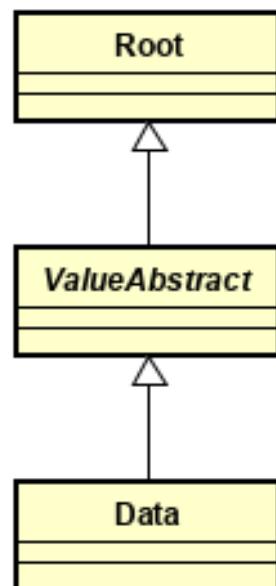


Ilustración 13 - Relación *ValueAbstract* con elementos HTML

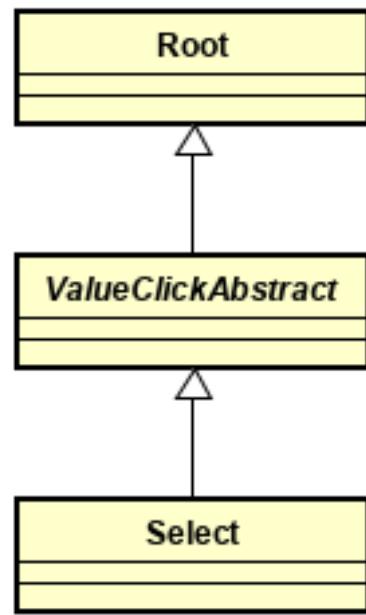


Ilustración 14 - Relación *ValueClickAbstract* con elementos HTML

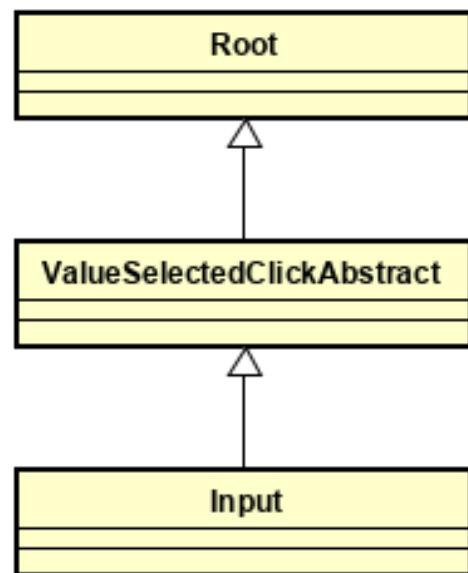


Ilustración 15 - Relación *ValueSelectedClickAbstract* con elementos HTML

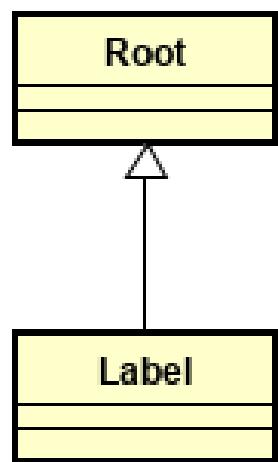


Ilustración 16 - Relación directa de Root con elementos HTML

Apéndice D Documentación técnica de programación

D.1. Introducción

En este apéndice vamos a explicar la documentación técnica de programación, es decir, la estructura de directorios del proyecto, la instalación del entorno de desarrollo y los test realizados.

D.2. Estructura de directorios

El repositorio del proyecto [19] presenta la siguiente estructura:

- `/`: contiene todos los ficheros de configuración, tanto para Git, como para NPM, o también para TypeScript. Además, contiene el fichero de arranque de la *toolkit*, el fichero README y la copia de la licencia.
- `/dist/`: carpeta en la que se encuentran todos los ficheros JavaScript correspondientes a la carpeta lib.
- `/lib/`: carpeta en la que se encuentran todos los ficheros TypeScript correspondientes a la *toolkit*.
- `/lib/aut/`: carpeta en la que se encuentra el fichero de automatización del proyecto.
- `/lib/config/`: carpeta en la que se encuentra el fichero de configuración, utilizado para el arranque de la *toolkit*.
- `/lib/parser/`: carpeta en la que se encuentra el fichero de parseo.
- `/lib/libraryWIO/`: carpeta en la que se encuentra toda la librería.
- `/lib/libraryWIO/abstractClass`: carpeta en la que se encuentran las clases abstractas.
- `/lib/libraryWIO/class`: carpeta en la que se encuentran todas las clases correspondientes a los elementos HTML.

- `/lib/libraryWIO/interfaces`: carpeta en la que se encuentran las interfaces.
- `/test/`: carpeta en la que se encuentran los test unitarios y de integración.

Para más información sobre cómo se tiene que estructurar un módulo NPM consulte la documentación [20].

D.3. Manual del programador

Este apartado tiene como objetivo establecer los pasos necesarios para que los próximos desarrolladores puedan seguir trabajando en la *toolkit*. En este manual se va a explicar el cómo se tiene que instalar el entorno de desarrollo, cómo podemos obtener todo el código fuente del proyecto, cómo lo podemos compilar, ejecutar y probar.

Entorno de desarrollo

Para poder continuar el desarrollo de este proyecto vamos a necesitar la instalación de los siguientes elementos:

- IDE o Editor de Texto.
- Node.
- TypeScript.
- Git.

IDE o Editor de Texto

Para el desarrollo de este proyecto hemos utilizado un editor de texto, en este caso, hemos usado Visual Studio Code [21]. Para mejorar la usabilidad dentro de éste puedes instalarte diferentes plugins.

La instalación de Visual Studio Code es sencilla, accediendo a la documentación [21] podemos descargarlo, una vez que hemos accedido a dicho *link* solo tenemos que seleccionar el sistema operativo y la arquitectura correspondiente para que comience la instalación, una vez finalizada ésta el asistente de instalación nos permitirá instalar dicho editor en nuestro ordenador.

Node

Para poder hacer uso de nuestra aplicación tenemos que instalarnos Node, éste no es más que un entorno de ejecución de JavaScript orientado a eventos asíncronos [22].

La instalación de Node es muy simple, accediendo a la documentación [23] podemos descargarlo, una vez que hemos accedido a dicho *link* tenemos que descárganos la versión 10.15.1 LTS.

TypeScript

TypeScript es un lenguaje de programación de código abierto, está orientado a objetos. TypeScript lo que permite es convertir el código desarrollado en este lenguaje a JavaScript.

Aunque TypeScript introduce una mayor complejidad, ya que éste se instala como una dependencia más en nuestro proyecto, la curva de aprendizaje es muy parecida a JavaScript ya que TypeScript se basa en JavaScript. Ambos lenguajes tienen una sintaxis similar, pero TypeScript añade el tipado como la característica más importante.

Para instalar TypeScript tenemos dos opciones:

- Obtener el código fuente de TestingAWE el cual se explicará más adelante, y situados en el directorio del repositorio ejecutar en la consola “*npm install*”.
- Una vez instalado Node, ejecutar en la consola “*npm install -g typescript*”.

Se recomienda la primera opción, ya que una vez obtenido el código, al ejecutar “*npm install*” vamos a instalar todo lo necesario (TypeScript, Ts-Node, Electron...) para poder probar TestingAWE sobre otra aplicación Angular/Electron.

Git

Para poder obtener todo el código fuente vamos a necesitar tener instalado Git, dicho gestor de versiones nos va a permitir clonar el proyecto de forma local en nuestro equipo, lo cual es necesario para obtener el código.

Realizar la instalación de Git es sencilla, podemos descargarlo desde su página oficial [24], una vez que hemos accedido al *link* tenemos que seleccionar el sistema operativo y la arquitectura correspondiente. Finalmente, el asistente de instalación nos permitirá instalar Git en nuestro equipo.

Obtención del código fuente

Para realizar el desarrollo de TestingAWE hemos utilizado un repositorio Git, el cual se encuentra en GitLab. La gestión de dicho repositorio lo hemos hecho a través del software GitKraken [25], esto es opcional, ya que si sabes manejarte por consola bash no hace falta la instalación de este software. Para instalar GitKraken tenemos que acceder al *link* que se proporciona en la documentación [25], una vez descargado el instalador, lo ejecutamos y siguiendo los pasos podremos obtener GitKraken en nuestro equipo.

Para poder clonar el código fuente del repositorio Git a nuestro equipo, lo primero de todo es abrir el software GitKraken.

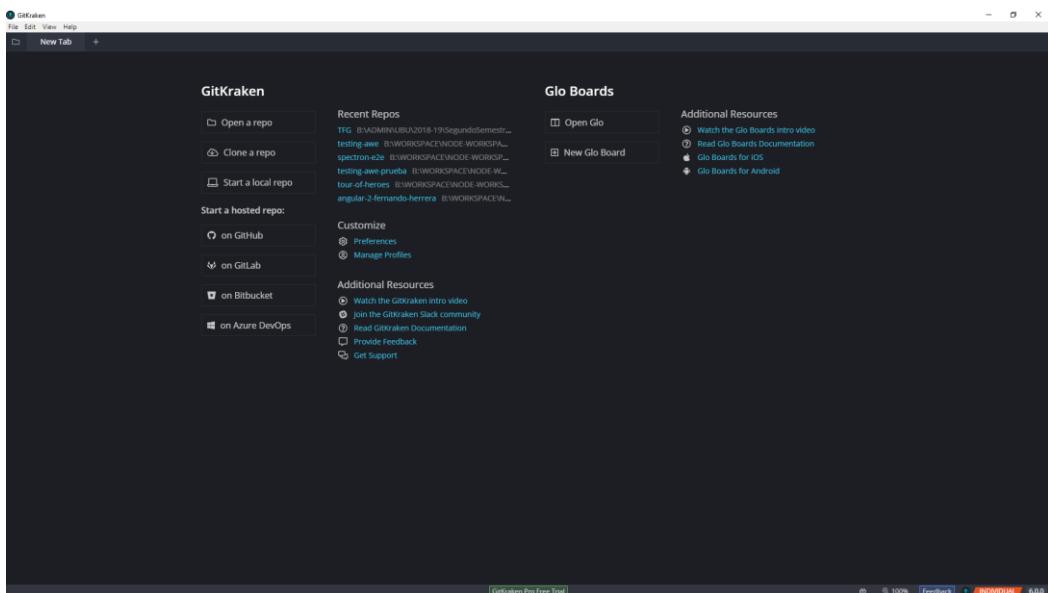


Ilustración 17 - Apertura de GitKraken

Después haremos click con el botón primario del ratón sobre la opción File y luego sobre la opción Clone Repo, accederemos a la siguiente ventana:

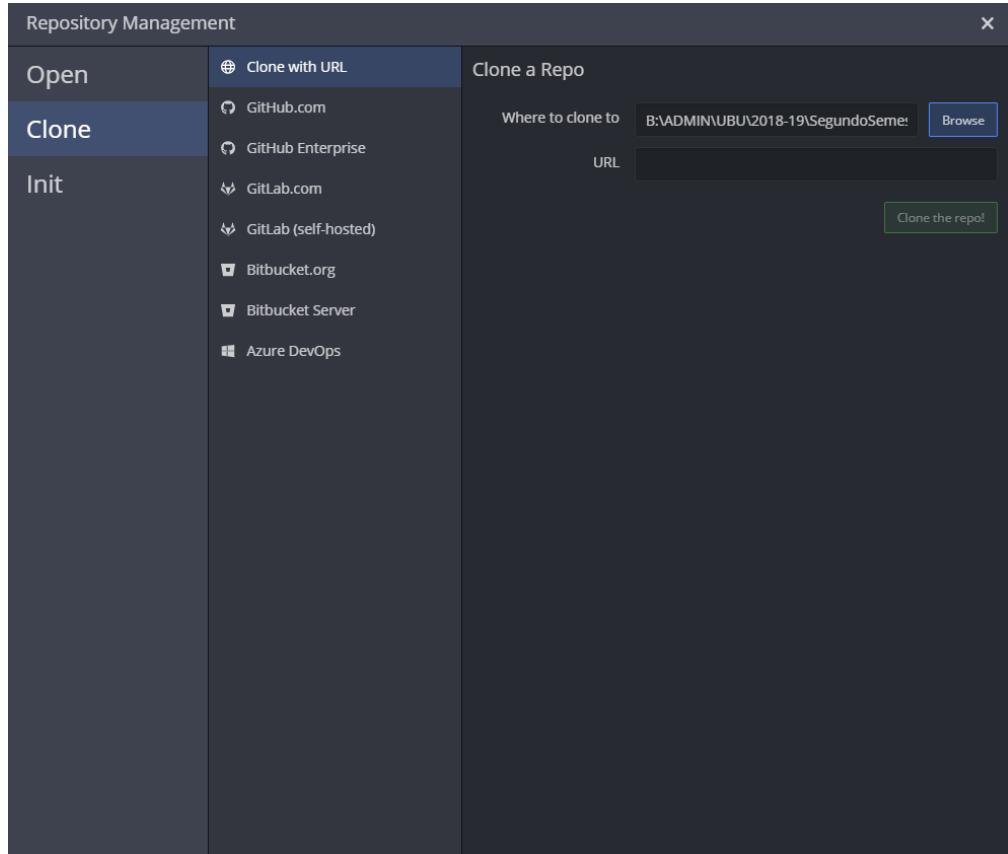


Ilustración 18 - Apertura ventana de clonación GitKraken

En ella introducimos los datos, el dónde queremos clonar el repositorio y cuál es la URL de dicho repositorio. Una vez introducidos los datos nos saldrá habilitado el botón “*Clone the repo!*” tal y como se muestra en la siguiente ilustración:

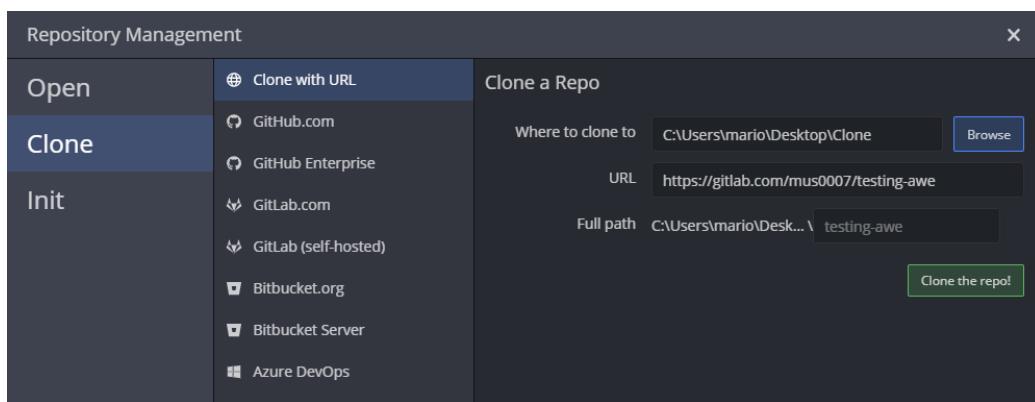


Ilustración 19 - Clonación del repositorio en GitKraken

Finalmente pulsamos al botón y tendremos en la ruta indicada la copia del proyecto TestingAWE. A partir de este momento ya podemos gestionar los cambios que vayamos a realizar en el proyecto.

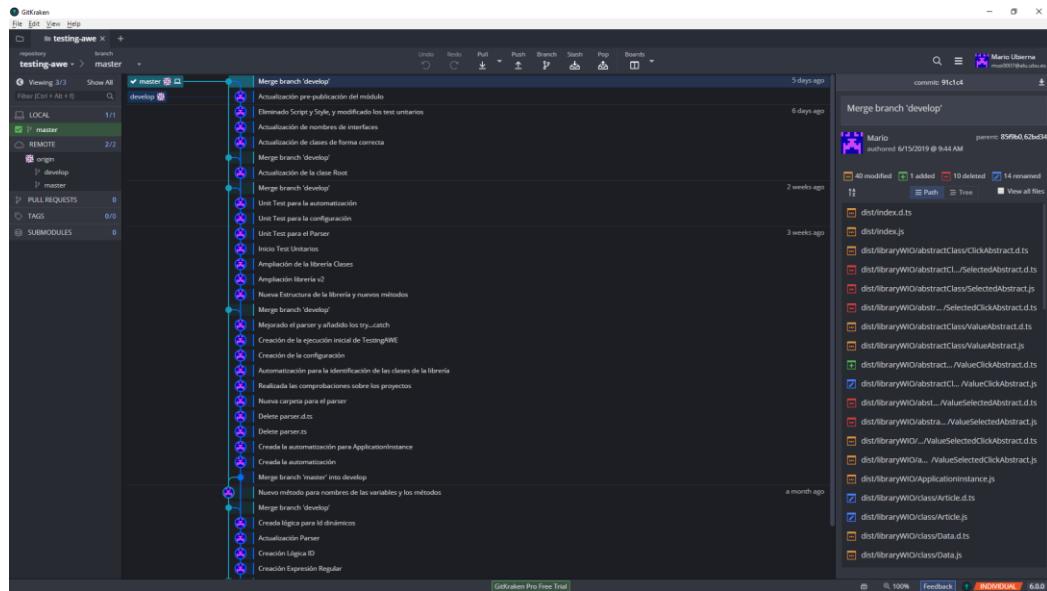


Ilustración 20 - Ventana del proyecto en GitKraken

Compilación código fuente

Una vez que ya tenemos la copia del proyecto en nuestro equipo, lo que debemos hacer es ejecutar en el directorio raíz de ese proyecto el siguiente comando “*npm install*” (en la consola), esto nos permitirá instalar todas las dependencias de nuestro proyecto, las cuales son necesarias, para la compilación y ejecución de los test.

La situación inicial de nuestro proyecto antes de ejecutar ese comando es la siguiente:

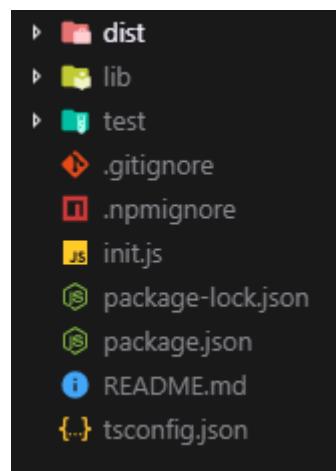


Ilustración 21 - Estructura inicial del proyecto

Una vez ejecutado ese comando, veremos que aparece un nuevo directorio en nuestra estructura, llamado “*node_modules*”, es aquí donde se instalan todas las dependencias de nuestro proyecto.

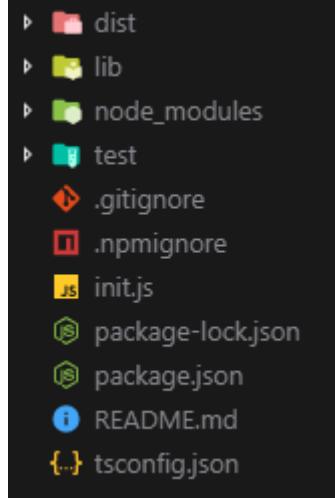


Ilustración 22 - Estructura final del proyecto

Finalmente, para poder realizar la compilación de nuestro proyecto tenemos que ejecutar el siguiente comando en la consola (dentro del directorio del proyecto) “*npm run prepublish*”.

Ese comando se encargará de transcribir “compilar” de fichero TypeScript a ficheros JavaScript, y el resultado de este proceso lo almacena en la carpeta **/dist/** del proyecto.

Se aconseja que cada vez que vayamos a compilar se elimine dicha carpeta. Aunque es verdad que, si un fichero tiene modificaciones y lo compilamos esas modificaciones van a seguir estando en **/dist/**, pero si por ejemplo eliminamos un fichero de **/lib/** al compilar va a seguir existiendo dicho fichero en **/dist/**.

Ejecutar toolkit

TestingAWE es un proyecto que por sí solo no se puede ejecutar, es decir, necesitamos de una aplicación Angular/Electron para probar el funcionamiento de esta *toolkit*.

Esta ejecución se explicará de forma más detallada en el manual del usuario, ya que es allí donde vamos a realizar todo el proceso desde cero (la instalación desde

NPM), pero si queremos probar TestingAWE desde la clonación debemos realizar los siguientes pasos:

1. Crear un nuevo proyecto, ejecutando “*npm init*”.
2. Instalamos de forma manual todas las dependencias del proyecto TestingAWE en el nuevo proyecto.
3. En la carpeta “*node_modules*” del nuevo proyecto pegamos el proyecto TestingAWE, es decir, la carpeta “*testing-awe*”.
4. El nuevo proyecto, y el proyecto Angular/Electron deben estar al mismo nivel, es decir, tienen que estar en el mismo directorio.
5. Cambiamos el fichero “*package.json*” del proyecto nuevo. En el atributo “*main*” establecemos el main del proyecto Angular/Electron.

```
"main": "../libraryTest/main.js"
```

Ilustración 23 - Modificación del atributo main del nuevo proyecto

En el atributo “*scripts*” añadimos las siguientes líneas

```
"scripts": {
  "initTestingAWE": "cd ./node_modules/testing-awe && npm run init",
  "start": "electron ../libraryTest/main.js",
  "test": "mocha -r ts-node/register test/**/*.{ts,js} --timeout 1000000"
},
```

Ilustración 24 - Modificación del atributo scripts del nuevo proyecto

El atributo “*start*” hace referencia al “*main*” del proyecto Angular/Electron. Si hemos realizado todos los pasos de forma correcta, al ejecutar el comando “*npm run initTestingAWE*” en el directorio del proyecto nuevo, nos generará los ficheros por cada HTML encontrado en el proyecto Angular/Electron. Puede ser que no tengamos en el proyecto nuevo el fichero de configuración “*aweconfig.json*”, en ese caso al ejecutar el comando anterior nos generará dicho fichero de configuración, el cual tendremos que modificar.

Se aconseja ejecutar la *toolkit* a partir de la instalación del módulo a través de NPM, dicha instalación y ejecución se encuentra en el manual de usuario. Además, se encuentra disponibles una serie de videos para que sirvan de guía [26].

D.4. Pruebas del sistema

Para poder verificar que la *toolkit* funciona de forma correcta se ha diseñado una batería de pruebas.

Se aconseja que para ejecutar los test se haga desde la instalación de NPM, y que haya una aplicación Angular/Electron sobre la que realizar las pruebas, ya que de lo contrario puede ser que algún test deje de funcionar (debido a que hay test en los que necesitamos que haya una aplicación Angular/Electron y otros en los que el módulo tiene que estar dentro de la carpeta “*node_modules*”).

Test unitarios

Las pruebas unitarias nos permiten comprobar el funcionamiento de un módulo aislado, es decir, que una determinada unidad de código haga lo que tiene que hacer.

Se han testado la parte de la automatización (*automation*, *configuration*, *parser*), ya que estos componentes no tienen la necesidad de lanzar Spectron para poder realizar las pruebas, es decir, no necesitan arrancar la aplicación Angular/Electron.

Para ejecutar estos test tenemos que dirigirnos al proyecto en el que esté instalado TestingAWE, posteriormente nos trasladamos hasta la carpeta “*testing-awe*” y dentro de ella ejecutamos el siguiente comando “*npm run test*”.

Test de integración

Las pruebas de integración nos permiten determinar si los diferentes componentes que forman el software funcionan correctamente cuando los probamos juntos.

Al igual que en el caso anterior se han testado la parte de la automatización, ya que no tienen la necesidad de lanzar Spectron para poder realizar las pruebas, es decir, no necesitan arrancar la aplicación Angular/Electron.

Tanto los test unitarios como los test de integración se pueden encontrar dentro de la carpeta **/test/** del módulo TestingAWE.

Para ejecutar estos test tenemos que dirigirnos al proyecto en el que esté instalado TestingAWE, posteriormente nos trasladamos hasta la carpeta “testing-awe” y dentro de ella ejecutamos el siguiente comando “*npm run test*”.

Test libraryWIO

Respecto a la parte de la librería no se han podido dejar los test dentro del módulo de TestingAWE, ya que en este caso sí que necesitamos lanzar Spectron para poder comprobar que todo funciona correctamente, y para lanzar Spectron necesitamos una aplicación Angular/Electron.

Debido a esto, lo que hicimos fue crearnos un nuevo proyecto Angular/Electron con todos los elementos HTML que soporta la librería TestingAWE, y crear test *end-to-end* para comprobar que para cada elemento HTML los métodos que tiene asociados funcionan correctamente.

Apéndice E Documentación de usuario

E.1. Introducción

En este anexo se va a detallar los requisitos para que funcione de forma correcta la *toolkit*, cómo se instala, cómo se utiliza correctamente. Como ya hemos mencionado antes, todos estos procedimientos se encuentran en video [26].

E.2. Requisitos de usuarios

Los requisitos necesarios para hacer uso de TestingAWE son los siguientes:

- Tener un ordenador con el sistema operativo Windows, éste tiene que soportar las tecnologías con las que vamos a trabajar.
- Tener instalado en el equipo Node y Angular.
- Tener una aplicación Angular/Electron sobre la que realizar pruebas.
- Tanto el proyecto en el que se va a instalar TestingAWE como el proyecto Angular/Electron, tienen que estar en el mismo directorio.

E.3. Instalación

La instalación de TestingAWE la podemos realizar a través de su página correspondiente en NPM [27].

Node Package Manager (NPM) es un gestor de paquetes para trabajar con Node, nos permite gestionar de una forma muy sencilla tanto paquetes públicos como privados. NPM es instalado automáticamente cuando instalamos Node, por lo que con instalar este último ya podemos hacer uso de NPM.

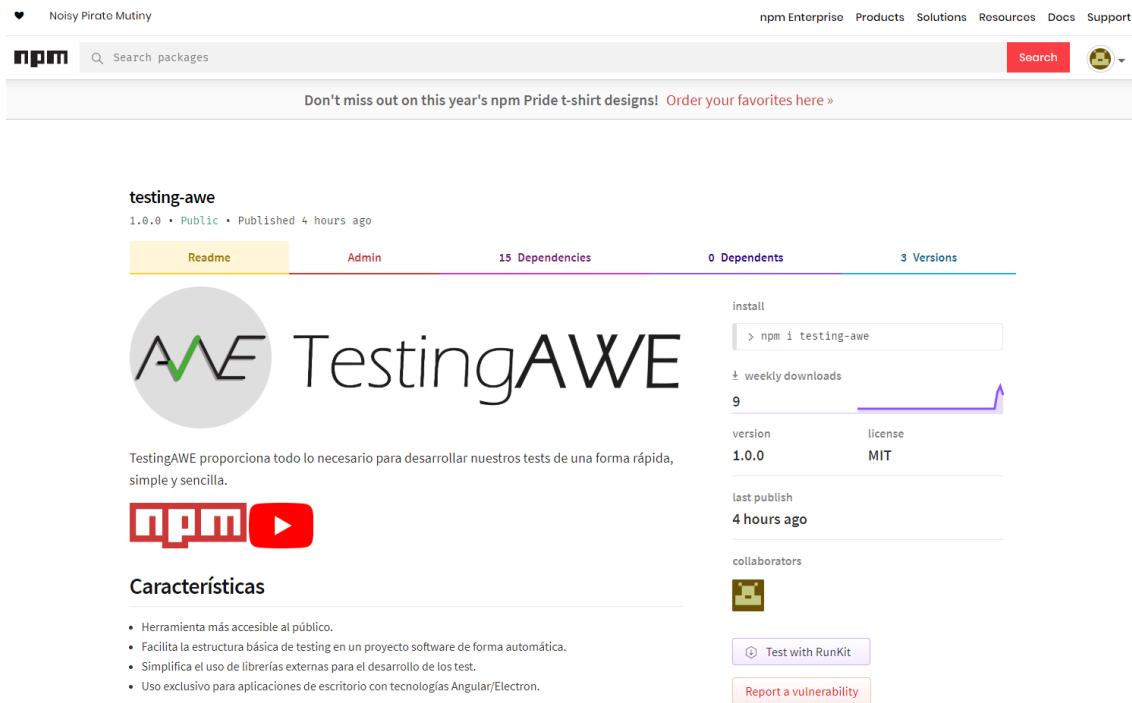


Ilustración 25 - Módulo TestingAWE en NPM

Para que la fase de instalación fuera más sencilla, en YouTube podemos encontrar el cómo se instala todo lo necesario junto con la propia instalación de TestingAWE [26].

Como recordatorio, antes de instalar la *toolkit* debemos de cumplir los siguientes requisitos:

- Tenemos que tener instalado en nuestro equipo Node. Para instalar Node, debemos acceder a su página web y descargarnos la última LTS.
- Tenemos que tener instalado en nuestro equipo Angular. Para instalar Angular solo debemos ejecutar el siguiente comando en la consola “*npm install -g @angular/cli@latest*”.

Para instalar la *toolkit* debemos realizar los siguientes pasos:

1. (Opcional) Abrimos nuestro navegador de confianza, Google Chrome, Mozilla Firefox, Microsoft Edge...
2. (Opcional) Introducimos en la barra de navegación la siguiente dirección <https://www.npmjs.com/>.
3. (Opcional) Dentro del buscador de NPM, introducimos el término “testing-awe”.

4. (Opcional) Pulsamos sobre el término buscado.
5. (Opcional) Nos abre la página correspondiente al módulo TestingAWE, y en ella aparece el comando de instalación para el paquete, en nuestro caso es “*npm install testing-awe*”.
6. Abrimos la consola, y nos situamos sobre el proyecto de pruebas en el que queremos instalar TestingAWE.
7. Introducimos el comando “*npm install testing-awe*” y pulsamos la tecla “Enter” o “Intro”.
8. Cuando haya terminado de instalarse todas las dependencias necesarias, la instalación habrá terminado, y se podrá hacer uso de TestingAWE.

E.4. Manual de usuario

En este apartado se va a explicar la configuración y el uso de TestingAWE para asegurarnos un correcto funcionamiento de la *toolkit*.

Una vez instalado Node y Angular (necesario para hacer uso de la *toolkit*), lo siguiente que debemos hacer es crearnos un nuevo proyecto de pruebas, es en éste donde vamos a instalar TestingAWE. Además, el proyecto de pruebas y el proyecto Angular/Electron tienen que estar al mismo nivel, es decir, en el mismo directorio. Para crear el proyecto de pruebas, tenemos que realizar los siguientes pasos:

1. Abrimos la consola.
2. No dirigimos a través del comando “*cd*” a la ruta en la que queremos crear el proyecto de pruebas.
3. Introducimos el comando “*npm init*” y pulsamos la tecla “Enter” o “Intro”.
4. Seguimos los pasos que nos indica la ejecución anterior.
5. Finalmente habrá finalizado la creación del proyecto de pruebas.

Una vez realizado el paso anterior, podemos instalar ya TestingAWE, para ello seguimos los puntos establecidos en el apartado anterior.

Configuración

Posteriormente tenemos que realizar una breve configuración, los pasos que debemos de seguir son los siguientes:

1. En el proyecto de pruebas tendremos un fichero “*package.json*”, éste lo tenemos que abrir con el editor de texto de confianza. Inicialmente dicho fichero tendría un contenido similar a este:

```
{
  "name": "spectron-2e2",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\"Error: no test specified\\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "testing-awe": "^0.1.1"
  }
}
```

Ilustración 26 - Package.json inicial

2. Lo primero que debemos hacer es cambiar el atributo “*main*”, ya que éste tiene que hacer referencia al main de Electron en el proyecto Angular/Electron. En la siguiente captura “*libraryTest*” representa al proyecto Angular/Electron, además, el “*..*” refleja que ambos proyectos están en el mismo directorio.

```
"main": "../libraryTest/main.js"
```

Ilustración 27 - Atributo main actualizado

3. Después tenemos que cambiar el atributo “*test*”, ya que éste tiene que hacer uso de mocha, el valor de dicho atributo tiene que ser el siguiente. Cabe destacar que “*test/**/*.ts*” refleja el directorio donde se encuentran todos los test que se van a ejecutar, lo podemos cambiar si queremos ejecutar un test en específico o una carpeta:

```
"test": "mocha -r ts-node/register test/**/*.ts --timeout 1000000"
```

Ilustración 28 - Atributo test actualizado

4. Finalmente, debemos añadir un nuevo atributo en el apartado scripts, éste es el que vamos a ejecutar para que TestingAWE realice toda la automatización.

```
"initTestingAWE": "cd ./node_modules/testing-awe && npm run init",
```

Ilustración 29 - Atributo de automatización

5. Una vez que hemos realizado este proceso, el resultado final de este fichero debe ser similar al siguiente:

```
{
  "name": "spectron-2e2",
  "version": "1.0.0",
  "description": "",
  "main": "../libraryTest/main.js",
  "scripts": {
    "initTestingAWE": "cd ./node_modules/testing-awe && npm run init",
    "test": "mocha -r ts-node/register test/**/*.{ts,js} --timeout 1000000"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "testing-awe": "^0.1.1"
  }
}
```

Ilustración 30 - Resultado final del fichero package.json

Realizando el paso anterior conseguimos establecer la configuración en el fichero “*package.json*”, pero necesitamos otra configuración para que la *toolkit* funcione de forma correcta, esta configuración es la del fichero “*aweconfig.json*”.

Si hemos seguido los pasos de forma correcta, en este punto veremos que nuestro proyecto de pruebas en el que está instalado TestingAWE, no tiene ningún fichero “*aweconfig.json*”, tenemos dos posibilidades para crearlo:

- Generar de forma manual dicho fichero.
- Ejecutar el comando “*npm run initTestingAWE*”, el cual se va a encargar de generar un fichero por defecto.

Se aconseja tomar la segunda opción, ya que de esta forma estamos garantizando la correcta estructura del fichero, es por ello que para completar este manual vamos a realizar esta opción. Por lo tanto, los pasos que debemos seguir son los siguientes:

1. Abrimos la consola, y nos situamos sobre el proyecto de pruebas que tiene instalado TestingAWE.
2. Introducimos el comando “*npm run initTestingAWE*”.

3. Pulsamos la tecla “Enter” o “Intro”.
4. Nos generará el archivo “aweconfig.json”.

Si hemos seguido los pasos anteriores, en la consola nos mostrará el siguiente mensaje:

```
NOTA: modifique el fichero aweconfig.json
ERROR: el directorio electronProjectPath no existe
```

Ilustración 31 - Mensaje inicial TestingAWE

Este mensaje lo que significa es que hemos creado un fichero “aweconfig.json” genérico. Por lo tanto, lo que debemos hacer es abrir el fichero “aweconfig.json” y modificar tanto la ruta del proyecto Angular/Electron, como la ruta del proyecto Spectron.

```
{
  "directories": {
    "electronProject": "electronProjectPath",
    "spectronProject": "spectronProjectPath"
  },
  "ignore": [
    "node_modules", "dist", "e2e", ".git"
  ]
}
```

Ilustración 32 - Aweconfig.json inicial

Analizando este fichero encontramos dos atributos:

- Directories: en el que establecemos las rutas tanto del proyecto Angular/Electron (electronProject), como del proyecto de pruebas en el que está instalado TestingAWE (spectronProject).
- Ignore: en el que establecemos las carpetas/ficheros del proyecto Angular/Electron que queremos ignorar, es decir, aquellas carpetas/ficheros que estén en la raíz del proyecto.

Por lo tanto, realizamos las modificaciones dentro del fichero “aweconfig.json”, y el resultado ha de ser similar a este:

```
{
  "directories": {
    "electronProject": "C:\\\\Users\\\\mario\\\\OneDrive\\\\Escritorio\\\\ManualDeUsuario\\\\libraryTest",
    "spectronProject": "C:\\\\Users\\\\mario\\\\OneDrive\\\\Escritorio\\\\ManualDeUsuario\\\\spectron-2e2"
  },
  "ignore": ["node_modules", "dist", "e2e", ".git"]
}
```

Ilustración 33 - Aweconfig.json final

Finalmente, si hemos seguido todos los pasos de forma correcta, al volver a ejecutar el comando “*npm run initTestingAWE*” veremos que nos crea los ficheros, es decir, los que contienen los elementos HTML. Estos ficheros se encuentran dentro de la carpeta “testingAWE”, la cual se crea en la raíz del proyecto Spectron (proyecto en el que está instalado TestingAWE).

Cabe recordar dos cosas:

- Se va a generar un fichero por cada HTML que se encuentre en el proyecto Angular/Electron.
- Solo se van a identificar elementos HTML que tengan un id, y que éste sea correcto.

Uso

En este apartado vamos a reflejar el cómo se usaría TestingAWE una vez que hemos generado los ficheros para así poder realizar pruebas. Para ello anteriormente hemos tenido que configurar el proyecto de pruebas, tal y como se mostraba en el apartado anterior.

Como recordatorio cabe mencionar, que en el canal de YouTube de TestingAWE podemos encontrar una serie de video-tutoriales para la configuración y uso de esta *toolkit* [26].

Una vez que hemos ejecutado el comando “*npm run initTestingAWE*” en el proyecto en el que está instalado TestingAWE, nos generará los ficheros necesarios para poder hacer pruebas sobre los elementos HTML que hay en la aplicación Angular/Electron.

El siguiente paso es crearnos los ficheros de pruebas, para ello lo que tenemos que hacer es crearnos un nuevo archivo TypeScript dentro de la carpeta “test” del proyecto Spectron, es decir, el proyecto en el que está instalado TestingAWE.

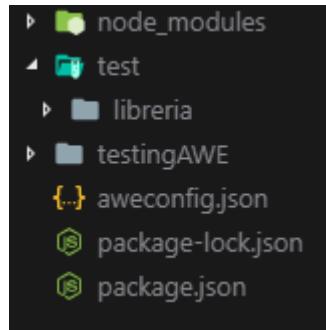


Ilustración 34 - Proyecto Spectron antes de crear un test

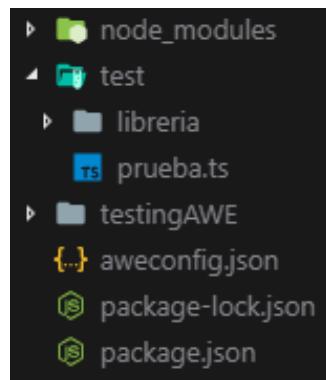


Ilustración 35 - Proyecto Spectron después de crear un test

Una vez que hemos creado el archivo, en este caso “prueba.ts”, lo abrimos para así comenzar el desarrollo del test.

Para el desarrollo del test tenemos que seguir la estructura que nos proporciona mocha, ésta la podemos dividir en tres partes:

- **Cabecera:** se corresponde con la parte superior del fichero, en ella importamos todo lo necesario para realizar el test. En nuestro caso, tenemos que importar Chai y applicationInstance obligatoriamente (ésta es la instancia de la aplicación Angular/Electron, y la importamos desde el fichero “applicationInstance.awe.ts” generado por la *toolkit* en la carpeta “testingAWE”).
- **Describe:** nos permite agrupar bloques de pruebas relacionadas entre sí.
- **It:** cada bloque it es una prueba, se pueden definir tantos it como se quiera dentro de un describe.

Un ejemplo de test podría ser el siguiente:

```
// Importación de la Librería Chai
import { assert } from "chai";

// Importación de la app con el patrón Singleton
import { applicationInstance } from "../testingAWE/applicationInstance.awe";
// Importación de los objetos
import { appPMain } from '../testingAWE/src/app/app.awe';

// Creación del TEST
describe("Prueba", () => {

    before(async () => {
        await applicationInstance.startApplication();
    });

    after(async () => {
        await applicationInstance.stopApplication();
    });

    it.only("getText", async () => {

        // Comprobamos el método getText()
        let text: string = await appPMain.getText();
        let expected: string = "Esto es un main";
        assert.equal(text, expected);
    });
});
```

Ilustración 36 - Ejemplo test TestingAWE

Como podemos observar en la anterior ilustración, realizamos tres importaciones:

- La librería Chai.
- El objeto applicationInstance, el cual es la instancia de la aplicación Angular/Electron.
- El objeto appPMain, el cual es la instancia del elemento HTML cuyo id es “app_p_main”.

En este caso el id del objeto appPMain es estático (no se calcula el id al arrancar la aplicación) por lo que se proporciona el objeto directamente (dicho objeto se encuentra en la fichero correspondiente generado por TestingAWE), sin embargo, si el id es dinámico (se calcula al arrancar la aplicación) se proporciona un objeto parametrizado, es decir, no se proporciona directamente el objeto, sino que se facilita una función a la cual le tenemos que pasar por cabecera el parámetro (dicha función se encuentra en la fichero correspondiente generado por TestingAWE), y esta función se encargará de generar el objeto correspondiente. Por si no ha quedado claro se recomienda ver los videos en YouTube [26].

Después de realizar las importaciones, definimos el *describe*, dentro de cada *describe* tenemos que arrancar la aplicación antes de todas las pruebas, y cerrarla después de que haya terminado de ejecutarse las mismas, es por ello que necesitamos definir dos métodos (son propios de Mocha):

- **Before:** método en el cual arrancamos la aplicación.
- **After:** método en el cual cerramos la aplicación.

Finalmente, definimos cada una de las pruebas que va a tener el *describe*, y dentro de ellas es donde realizamos los test, es decir, lo que queremos probar. En este caso comprobamos que la salida del método *getText()* es igual a lo que esperamos.

Cabe mencionar que los métodos proporcionados para cada objeto, son similares a los que proporciona WebdriverIO, por lo que si se quiere saber qué es lo que hace un método se puede acceder a la *Api* que proporciona dicha librería [28].

Para terminar, la ejecución del comando “*npm run test*” en la consola, nos permite probar los tests, tal y como se muestra en la siguiente ilustración:

```
C:\Users\mario\OneDrive\Escritorio\ManualDeUsuario\spectron-2e2>npm run test
> spectron-2e2@1.0.0 test C:\Users\mario\OneDrive\Escritorio\ManualDeUsuario\spectron-2e2
> mocha -r ts-node/register test/**/*.ts --timeout 1000000

Prueba
  ✓ getText (46ms)

1 passing (5s)
```

Ilustración 37 - Ejecución del test

Bibliografía

- [1] «Licencia de software», *Wikipedia, la enciclopedia libre*. 18-jun-2019.
- [2] «Licencia MIT», *Wikipedia, la enciclopedia libre*. 04-oct-2018.
- [3] andrearrrs, «Cómo elegir licencias open source para tu proyecto», *Hipertextual*, 22-may-2014. [En línea]. Disponible en: <https://hipertextual.com/archivo/2014/05/como-elegir-licencias-open-source/>. [Accedido: 18-jun-2019].
- [4] «Apache License», *Wikipedia, la enciclopedia libre*. 16-may-2019.
- [5] «Directrices de software libre de Debian», *Wikipedia, la enciclopedia libre*. 29-sep-2015.
- [6] «GNU General Public License», *Wikipedia, la enciclopedia libre*. 27-abr-2019.
- [7] «Free Software Foundation», *Wikipedia, la enciclopedia libre*. 03-abr-2019.
- [8] «Open Source Initiative», *Wikipedia, la enciclopedia libre*. 21-mar-2018.
- [9] «Copyleft», *Wikipedia, la enciclopedia libre*. 10-jun-2019.

- [10] «Creative Commons — Attribution 4.0 International — CC BY 4.0». [En línea]. Disponible en: <https://creativecommons.org/licenses/by/4.0/>. [Accedido: 18-jun-2019].
- [11] R. M. Agut, «Especificación de Requisitos Software según el estándar de IEEE 830», p. 19.
- [12] «IEEE 830-1998 - IEEE Recommended Practice for Software Requirements Specifications». [En línea]. Disponible en: <https://standards.ieee.org/standard/830-1998.html>. [Accedido: 19-jun-2019].
- [13] «TestingAWE - Azure DevOps». [En línea]. Disponible en: <https://dev.azure.com/juanmanuelzamarrenoperez/Observatorio-TestingAWE>. [Accedido: 19-jun-2019].
- [14] «Modelo–vista–controlador», *Wikipedia, la enciclopedia libre*. 13-abr-2019.
- [15] «MVC (Model, View, Controller) explicado.», *CódigoFacilito*. [En línea]. Disponible en: <https://codigofacilito.com/articulos/mvc-model-view-controller-explicado>. [Accedido: 27-abr-2019].
- [16] «Singleton», *Wikipedia, la enciclopedia libre*. 10-oct-2018.
- [17] «Patrones de diseño en automatización: PageObjects», *QA:news*, 07-ago-2014. .
- [18] Mohsin, «Page Object Model (POM) | Design Pattern», *tajawal*, 24-may-2018. .
- [19] «Repositorio TestingAWE», *GitLab*. [En línea]. Disponible en: <https://gitlab.com/mus0007/testing-awe>. [Accedido: 25-jun-2019].

- [20] G. Peraferrer, «Como crear un módulo NPM - Germán Peraferrer», *Medium*, 23-ago-2015. [En línea]. Disponible en: <https://medium.com/@peraferrer/como-crear-un-m%C3%B3dulo-npm-6baef161a96>. [Accedido: 21-jun-2019].
- [21] «Visual Studio Code - Code Editing. Redefined». [En línea]. Disponible en: <https://code.visualstudio.com/>. [Accedido: 21-jun-2019].
- [22] F. de Node.js, «Node», *Node.js*. [En línea]. Disponible en: <https://nodejs.org/es/about/>. [Accedido: 21-jun-2019].
- [23] F. de Node.js, «Node instalación», *Node.js*. [En línea]. Disponible en: <https://nodejs.org/es/>. [Accedido: 21-jun-2019].
- [24] «Git». [En línea]. Disponible en: <https://git-scm.com/>. [Accedido: 21-jun-2019].
- [25] «GitKraken», *GitKraken.com*. [En línea]. Disponible en: <https://www.gitkraken.com/>. [Accedido: 21-jun-2019].
- [26] «YouTube TestingAWE Oficial», *YouTube*. [En línea]. Disponible en: <https://www.youtube.com/channel/UCqUSDpoDw3RbqPb6PjAAAnBA>. [Accedido: 25-jun-2019].
- [27] «TestingAWE», *npm*. [En línea]. Disponible en: <https://www.npmjs.com/package/testing-awe>. [Accedido: 23-jun-2019].
- [28] «WebdriverIO documentation». [En línea]. Disponible en: <https://webdriver.io/index.html>. [Accedido: 27-abr-2019].



Esta obra está bajo una licencia Creative Commons Reconocimiento 4.0
Internacional ([CC-BY-4.0](#))