# **Portada**

Número de grupo: E3.08

Url al repositorio: <https://github.com/antsermen/Acme-Toolkits-E3.08.git>

Miembros (+ emails):

- José Ramón Arias Expósito ([jrae1112jrae@gmail.com](mailto:rae1112jrae@gmail.co)),

- Manuel Carnero Vergel ([mancarver1@alum.us.es](mailto:mancarver1@alum.us.es)),

- Juan Carlos Moreno Pérez ([morenoperezjuancarlos@gmail.com](mailto:morenoperezjuancarlos@gmail.com)),

- Pablo Santos Pérez ([pablo.santos.232001@gmail.com](mailto:pablo.santos.232001@gmail.com)),

- David Sabugueiro Troya ([dsabugueiro14@gmail.com](mailto:dsabugueiro14@gmail.com)),

- Antonio Roberto Serrano Mena ([robertoserranomena@gmail.com](mailto:robertoserranomena@gmail.com))

Fecha: 31/05/2022

# **Tabla de contenidos**

Contenido

[**1.** **Portada** 1](#_Toc104915300)

[**2.** **Tabla de contenidos** 2](#_Toc104915301)

[**3.** **Resumen ejecutivo** 3](#_Toc104915302)

[**4.** **Tabla de revisión** 3](#_Toc104915303)

[**5.** **Introducción** 4](#_Toc104915304)

[**6.** **Contenidos** 5](#_Toc104915305)

[**7.** **Conclusiones** 8](#_Toc104915306)

[**8.** **Bibliografía** 8](#_Toc104915307)

# **Resumen ejecutivo**

En este documento podemos encontrarnos con 8 apartados: una portada, que contiene el número de grupo, la url al repositorio en github, los miembros (junto a sus emails) y la fecha de creación del documento; una tabla de contenidos, que sería un índice típicamente conocido para un documento; un resumen ejecutivo, en el que se resumen los contenidos de cada apartado del documento; una tabla de revisión, formada por los datos de las revisiones realizadas sobre el documento (número de revisión, fecha y descripción); una introducción, que explica de manera resumida el tema a tratar (testing); los contenidos del documento, donde se recogen los conceptos que hemos considerado relevantes a la hora de elaborar un documento sobre nuestros conocimientos en el ámbito de testing; unas conclusiones, extraídas a partir de la finalización del documento y una bibliografía, que contiene las fuentes que nos sirvieron de apoyo para el documento (en caso de no encontrarse ninguna, aparecerá “Intencionalmente en blanco.”).

# **Tabla de revisión**

|  |  |  |
| --- | --- | --- |
| **N.º** | **Fecha** | **Descripción** |
| 1 | 26/05/2022 | Creación del documento y relleno parcial |
| 2 | 30/05/2022 | Puesta a punto del documento |
| 3 | 31/05/2022 | Preparación para entrega |

# **Introducción**

A lo largo del contenido de este documento encontraremos información que hemos podido recopilar gracias a las clases teóricas de la asignatura, las revisiones de las clases prácticas, y nuestra experiencia propia a la hora de realizar las tareas de cada entregable, sobre el testing de las funcionalidades (features) que hemos implementado en el proyecto. Toda la información estará basada principalmente en nuestra experiencia y será reforzada con apuntes de teoría.

La estructura del documento está dividida en 8 apartados, de los cuales los que más peso tienen son Contenidos, Resumen ejecutivo y Tabla de revisión, ya que en estos 3 apartados podemos encontrar esencialmente lo que nos pide el reporte de progreso: el contenido del progreso, las revisiones del mismo, etc.

# **Contenidos**

**-¿Qué entendemos por testing?**

El testing es una buena práctica utilizada por los programadores e indispensable en cualquier software, con el objetivo de encontrar en este comportamientos anómalos causados por bugs hasta entonces desconocidos, que en prácticamente el 100% de códigos acaban apareciendo en las distintas fases del proyecto a desarrollar.

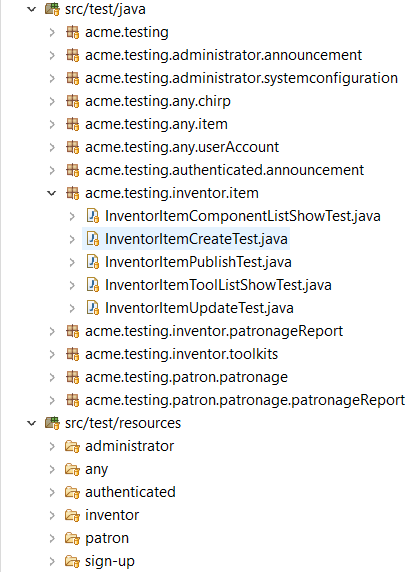
**-¿En qué consiste el testing?**

Consiste en indicar una serie de acciones que la aplicación deberá completar y de la que ya conocemos el resultado que debería devolver una vez finalizadas dichas acciones. Una vez la aplicación las ha cumplido, se comprueba el resultado de esta con el esperado, si no coinciden, se debe buscar el origen del bug y solucionarlo, si coinciden, significa que en ese caso la aplicación no se ha comportado de manera extraña, pero no podemos afirmar que no existan bugs en la aplicación, incluso pueden seguir existiendo bugs en la feature que se está probando. Lo óptimo sería poder implementar testing de la mayor cantidad de código posible y de todas las formas posibles para tener la máxima certeza posible a la hora de estipular que un software va a actuar de forma adecuada. Cuanto mayor sea el porcentaje de código cubierto por el testing, más confiable será un proyecto

**-¿Cómo hemos realizado nuestro testing en Acme Framework?**

Para nuestros tests, teníamos como objetivo simular las interacciones que un usuario podría tener con la aplicación Acme-Toolkits que hemos estado desarrollando durante todos los entregables. Hemos usado JUnit5 y Selenium para implementarlos y para ejecutarlos hemos ejercido el uso de Gecko Driver para poder simular los pasos a seguir por el usuario mediante una ventana de Mozilla Firefox en modo “marioneta”.

En primer lugar hemos creado una clase con el nombre de la acción de la feature a probar seguida de la palabra “Test” dentro de un paquete denominado “acme.testing.FEATURE” donde FEATURE equivale a la feature a probar. Estos paquetes están a su vez en la carpeta src/test/java. De esta forma tendremos estructurado el sistema de testing y perderemos poco tiempo a la hora de encontrar el test que buscamos, ya que esto puede complicarse a medida que crece el proyecto y es tiempo que nos podemos ahorrar. También encontramos otra carpeta llamada src/test/resources en la que añadiremos los archivos .csv con datos para los test que hacen uso de datos predefinidos.



*Estructura de las carpetas relacionadas con Testing.*

Todas esas clases extienden de TestHarness, excepto los tests de las features Chirp y Announcement, que extienden de TemporalAwareTestHarness. Estos necesitan ser “congelados en el tiempo” debido a problemas con las fechas a la hora de comprobar su correcto funcionamiento. Esta clase a su vez extiende de TestHarness. Están conformadas por estos métodos:

* Un **PositiveTest** donde se le pasa una serie de instrucciones que se espera que el navegador en modo marioneta pueda cumplir sin encontrar ningún tipo de fallo. Esto se consigue gracias al uso de diferentes comandos como:
* checkListingExists(): Comprueba que se pueda listar.
* checkFormExists(): Comprueba que existe un formulario para, por ejemplo, crear una herramienta.
* checkColumnHasValue(): Comprueba que los valores de las columnas correspondan con los del archivo de test .
* checkInputBoxHasValue(): Comprueba que los valores de la base de datos coinciden con los del archivo del test.
* Un **NegativeTest** consistente en una serie de acciones que esperan ser interrumpidas por errores como por ejemplo acciones inválidas o datos incorrectos. Estos métodos se consideran exitosos si la aplicación devuelve que ha encontrado errores mientras seguía los pasos. Para los test negativos, usamos los mismos comandos que para los test positivos, pero hay un comando que es solo de los test negativos:
* checkErrorsExist(): Con este comando, comprobamos que hay errores, los necesarios para que el test negativo vaya bien.
* Un **HackingTest**. Aunque pudiera parecer que este Test es lo mismo que un NegativeTest, la realidad es que no son exactamente los mismo, ya que aunque ambos esperan un resultado “insatisfactorio”, la diferencia se encuentra en que en el HackingTest buscamos lo que se conoce como “panics” (errores críticos) producidos por la ejecución de acciones ilegales en la aplicación, como intentar entrar a una página a la que no se tiene permiso modificando la URL del navegador, mientras que en un NegativeTest siempre se realizan acciones permitidas. Esto se consigue gracias al uso de diferentes comandos como:
* navigate(path): Con este comando, le pasamos el path que queramos que la máquina navegue, por ejemplo: "/inventor/item/create"
* checkPanicExists(): Este comando lo utilizamos solo en los hacking test, comprueba que no se puede acceder a la vista que le hemos pasado con el comando navigate(), de la que no tengamos permisos.

# **Conclusiones**

Como conclusión, podemos extraer que nuestro progreso en el curso nos ha proporcionado un gran conocimiento sobre el testing, diferentes tipos y modelos de este, sus utilidades y las capacidades teórico-prácticas para poder diseñar sistemas complejos de testing para automatizar las pruebas sobre nuestra aplicación.

En el desarrollo de la asignatura hemos desarrollado el testing utilizando JUnit5 y Selenium, así como utilizado el plugin GeckoDriver y el buscador Mozilla Firefox. Hemos aprendido a realizar tres diferentes tipos de tests, cada uno con sus propia utilidad, diseño y comandos.

Finalmente, podemos asegurar que nuestros conocimientos sobre tests han aumentado en el transcurso de la asignatura, siendo capaces de desenvolvernos con soltura en este aspecto a fecha de finalización de este documento.

# **Bibliografía**

Intencionalmente en blanco.

# 