UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Graduado en Ingeniería del Software

Creación de pruebas automatizadas para modelos ejecutables de software

Automatic tests generation for executable software models

Realizado por
Ismael Alonso Gómez Calero

Tutorizado por
Antonio Jesús Vallecillo Moreno
Paula Muñoz Ariza

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, junio de 2021

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DEL SOFTWARE

Creación de pruebas automatizadas para modelos ejecutables de software

Automatic tests generation for executable software models

Realizado por
Ismael Alonso Gómez Calero

Tutorizado por
Antonio Jesús Vallecillo Moreno
Paula Muñoz Ariza

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO DE 2021

Fecha defensa: julio de 2021

Resumen

El Desarrollo Guiado por Comportamiento (conocido como BDD, por sus siglas en inglés, *Behavior Driven Development*) es una estrategia para desarrollar software usando la especificación del comportamiento y los resultados esperados del programa para su desarrollo y para definir las pruebas.

Por otro lado, la Ingeniería de Software Dirigida por Modelos (ISDM) se basa en el uso de modelos en todas las fases de desarrollo del software, aumentando así el nivel de abstracción y por tanto reduciendo notablemente la complejidad añadida al trabajar con lenguajes de programación que implementan las aplicaciones software.

En este proyecto se desarrolla una propuesta para la unión de estos dos conceptos, creándose una herramienta que implementa las pruebas necesarias para verificar el comportamiento de modelos UML en USE con ayuda de Gherkin y Cucumber. USE es un entorno de especificación de sistemas que utiliza UML como lenguaje de modelado para definir su estructura y comportamiento general y SOIL para definir su comportamiento detallado. Por su parte, Gherkin utiliza el patrón *"Given-When-Then"* para especificar las pruebas. Cada palabra clave define un aspecto de la prueba: las precondiciones (*Given*), las condiciones de la acción que se va a ejecutar (*When*) y el resultado esperado (*Then*). A partir de las especificaciones, se detallarán las pruebas en lenguaje SOIL. Tras esto, se usará la herramienta Cucumber para la generación automática de dichas pruebas.

Palabras clave: BDD, ISDM, USE, Gherkin, Cucumber

Abstract

Behavior Driven Development (BDD) is a strategy used to develop software using the behavior specification and the expected outputs from the program in order to develop it and define the tests for it.

On the other hand, in Model-Driven Development (MDD), models are used in every software development phase, increasing the abstraction level and decreasing the complexity added by the programming languages used to create the software applications.

This project develops a proposal for joining these two concepts, developing a tool that implements the required tests to verify the behavior of USE models using Gherkin and Cucumber. USE is a system specification environment based on UML as a modeling language to implement its structure and general behavior, and SOIL to implement its detailed behavior. Gherkin uses the "Given-When-Then" pattern. Every keyword represents a different side of the test: the preconditions (Given), the conditions of the action that will be performed (When) and the expected outputs (Then). Beginning with these specifications, the tests will be itemized in the programming language called SOIL. After this, the tests will be automatically generated using Cucumber.

Keywords: **BDD, MDD, USE, Gherkin, Cucumber**

Índice

1

Introducción

En este capítulo se desarrollarán tanto los distintos aspectos que han llevado a la decisión de realizar este proyecto como las partes que han hecho posible su realización, pasando por el resultado final esperado del mismo. Además, se explicarán tanto la metodología utilizada como las fases realizadas durante su desarrollo, al igual que la estructura que compone esta memoria.

1.1 Motivación

Behaviour Driven Development (BDD) o Desarrollo Guiado por Comportamiento (a partir de ahora nombrado como DGC), es una estrategia que se utiliza para desarrollar software usando la especificación del comportamiento y los resultados esperados del programa como guía de desarrollo y para definir sus pruebas. DGC forma parte también de lo que se conoce como Desarrollo Dirigido por Pruebas

(TDD, *Test Driven Development*) que convierte los requisitos en casos de prueba antes de que el software esté completamente desarrollado, y utiliza esos casos de prueba para guiar el desarrollo. Otra gran ventaja de utilizar esta estrategia es que utiliza un lenguaje común, bastante cercano al lenguaje natural, para especificar las pruebas, de forma que puedan ser fácilmente entendibles para todas las personas que intervienen en el desarrollo del sistema (propietario, usuarios finales, desarrolladores, etc.). Esta estrategia utiliza principalmente el patrón "*Given-When-Then*" como lenguaje común, estos apartados definen lo siguiente:

- *Given* 'Dado': especifica las precondiciones de la prueba y define el escenario.
- *When* 'Cuando': especifica las condiciones que se deben llevar a cabo durante la prueba.
- *Then* 'Entonces': valida el resultado esperado de ejecutar las condiciones necesarias sobre el escenario especificado.

Por otro lado, la Ingeniería de Software Dirigida por Modelo (a partir de ahora ISDM) se basa en el uso de modelos en todas las fases del desarrollo, elevando el nivel de abstracción y por tanto reduciendo notablemente la complejidad accidental que introducen los lenguajes de programación que implementan las aplicaciones software. En ISDM, los modelos se usan para la descripción, validación, verificación y generación de código de la aplicación. Actualmente también se trabaja con *modelos ejecutables,* que ya no necesitan ser traducidos a lenguajes de programación, sino que contienen suficiente información para ser ejecutados directamente. Como artefactos de software que son, dichos modelos ejecutables deben ser probados. Sin embargo, el desarrollo de las pruebas de este tipo de modelos es un tema que aún necesita ser estudiado a fondo.

USE (*UML-based Specification Environment*) es una de las herramientas más utilizadas para la creación de modelos ejecutables, ya que permite definir modelos utilizando UML (*Unified Modeling Language*), al igual que definir su comportamiento usando el lenguaje SOIL (*Simple OCL-based Imperative Language*). Esta herramienta es muy utilizada tanto en investigación como en docencia, y se utiliza de ayuda para impartir muchas materias relacionadas tanto con modelado como con métodos formales.

Actualmente la herramienta USE carece de cualquier tipo de ayuda a la hora de implementar pruebas de manera sencilla, de forma que para poder participar en esta etapa de desarrollo es necesario tener conocimientos suficientes como para poder utilizarla sin ningún tipo de problema. Otra herramienta de apoyo que ayudase a implementar pruebas de forma sencilla sería bastante beneficiosa para ella, ya que está siendo muy utilizada en distintos ámbitos.

La finalidad de este proyecto es ayudar a mejorar la forma en que se generan estas pruebas para la herramienta USE, con la ayuda de herramientas externas que se utilizan para la creación de pruebas automáticas, como Gherkin y Cucumber, que se explicarán más adelante en esta memoria.

1.2 Objetivos

El objetivo principal de este proyecto es desarrollar una herramienta que permita generar pruebas de forma automática con la ayuda de la especificación de distintos escenarios y su código correspondiente en lenguaje SOIL, de forma que podremos implementarlo en USE.

Se utilizará Gherkin para la definición de escenarios de prueba. Gherkin utiliza ficheros con terminación '*.feature'* para definir los escenario. Esta herramienta posee distintos aspectos que nos serán de ayuda a la hora de implementar la nuestra.

A su vez, se utilizará Cucumber para definir las pruebas en lenguaje Java. Esta herramienta se encarga en ejecutar las pruebas en el orden en el que se encuentran en Gherkin, ejecutando así el patrón *Given-When-Then*.

Ya que Cucumber está definido en este contexto para Java, se creará también un fichero que permita traspasar el código necesario para ser implementado en USE, esto es, añadir los componentes Java necesarios al código USE para que no dé error al estar en un fichero Java, ya que este lenguaje no permite sintaxis de la herramienta nombrada. Además, este fichero permite verificar la correcta implementación de las partes necesarias, utilizando también Java, de forma que las pruebas se podrán ejecutar en USE para validar nuestros modelos ejecutables.

Una vez que todo esto haya sido implementado, la herramienta deberá recibir por parámetro los ficheros de USE y SOIL, de forma que pueda añadir en ellas las pruebas necesarias y la ejecución de las mismas.

1.3 Metodología

Para el desarrollo de este proyecto se ha utilizado la metodología ágil Scrum adaptada a un solo participante, en el que la función de los *Product Owners* ha sido desempeñada por los tutores del proyecto.

Esta metodología es muy flexible a cambios, ya que se atienden a reuniones frecuentes, normalmente semanales, que permite mostrar el trabajo realizado y descubrir cualquier tipo de error con la mayor brevedad posible, permitiendo así solucionar errores durante el desarrollo y no cuando haya acabado el proyecto, suponiendo así grandes desperdicios para el proyecto en cuestión.

1.4 Fases de trabajo

Este proyecto se dividió en las siguientes fases:

Fase 1 - Especificación de requisitos

En esta fase se han especificado las distintas funcionalidades que se van a poder conseguir con la herramienta.

Fase 2 - Estudio de las herramientas a utilizar

En esta fase se ha realizado un estudio sobre las distintas herramientas que se utilizaron para la creación de este proyecto, principalmente han sido USE y el lenguaje SOIL, junto a Gherkin y Cucumber.

Fase 3 - Diseño y modelado

En esta fase se ha planteado la estructura que se llevaría a cabo a la hora de desarrollar la herramienta.

Fase 4 - Implementación

En esta fase se han desarrollado los distintos ficheros de código que serán utilizados en distintos ámbitos de la herramienta.

Fase 5 - Pruebas

En esta fase se han desarrollado distintas pruebas para validar el correcto funcionamiento de la herramienta. En primer lugar se utilizó el modelo de un brazo robótico que movía *ítems* y, posteriormente, el modelo de un microondas con sus distintas funciones, este último lo tomamos de otra propuesta que empleaba desarrollo guiado por comportamiento para la validación del comportamiento de un sistema definido por diagramas de estado [1].

Fase 6 - Desarrollo de la memoria y documentación

En esta fase se ha redactado la memoria del proyecto, al igual que un manual de instalación y de usuario que se encuentra en la misma.

1.5 Estructura de la memoria

En este apartado se enumera cada capítulo próximo junto a una breve explicación de su contenido.

Capítulo 2 - Estado del arte

En este capítulo se expondrán los distintos aspectos del Desarrollo Guiado por Comportamiento y de la Ingeniería del Software Dirigida por Modelos, teniendo en cuenta los aspectos que más se pueden relacionar con este proyecto. Además de esto, se nombrarán distintas herramientas similares a la desarrollada en este proyecto, enumerando las ventajas y desventajas de cada una frente a la misma.

Capítulo 3 - Herramientas utilizadas

En este capítulo se detallarán todas las herramientas utilizadas para el desarrollo de este proyecto, destacando que partes de ellas ha sido fundamental para su desarrollo.

Capítulo 4 - Desarrollo de la solución

En este capítulo se hablará sobre el funcionamiento de nuestra herramienta, centrándonos en la implementación y en problemas resueltos con ella.

Capítulo 5 - Validación y pruebas

En este capítulo se enumerarán las distintas pruebas que se han utilizado para validar el correcto funcionamiento de la herramienta explicando las razones por las que se han escogido cada una de ellas.

Capítulo 6 - Conclusiones

Para finalizar la memoria, se hablará sobre los problemas encontrados en el desarrollo de la herramienta. A su vez, se nombrarán algunas posibles líneas futuras para extender la misma.

2

Estado del arte

En este capítulo se hablará sobre el estado actual de los distintos ámbitos de los que forma parte este proyecto, estos son el Desarrollo Guiado por Comportamiento y la Ingeniería del Software Dirigida por Modelos.

Sobre el Desarrollo Guiado por Comportamiento se hará una introducción y se hablará sobre las ventajas que posee frente a otros métodos de desarrollo.

En la siguiente sección se hablará sobre la Ingeniería del Software Dirigida por Modelos. En dicha sección se mostrará una introducción y se hablará de los principios básicos de este paradigma.

Finalmente, se hablará de distintas herramientas que tienen una funcionalidad similar a la desarrollada en este proyecto, al igual que de las ventajas que se pueden encontrar en esta.

2.1 Desarrollo Guiado por Comportamiento

El Desarrollo Guiado por Comportamiento [2] (DGC o BDD de sus siglas en inglés, *Behaviour Driven Development*) es una técnica ágil orientada al desarrollo de software teniendo como base el comportamiento de la aplicación. Su principal objetivo es tener constancia de qué debe hacer la aplicación, antes y durante todo el desarrollo de la misma. Esto es posible a través de la generación de pruebas, que en este caso se utilizan desde un primer momento. De esta forma se podrá implementar a la vez que se valida el correcto funcionamiento del programa, ya que a medida que se avanza en él, se ejecutarán las pruebas que ayudarán a certificar que lo implementado está hecho de forma correcta.

DGC forma parte también del Desarrollo Guiado por Pruebas (conocido como TDD, por sus siglas en inglés, *Test-Driven Development*). A diferencia de TDD, que se centra en pruebas unitarias, DGC se basa en pruebas en el comportamiento de la aplicación, dejando de lado la manera en la cual esté implementada. Esto permite que tanto DGC como TDD puedan usarse en conjunto, y además, hace que sea muy recomendable.

La mayoría de las herramientas que permiten el uso de GDC adoptan un lenguaje natural y entendible por todas las personas relacionadas con el proyecto, haciendo así que tanto el jefe de proyecto, analistas o incluso el cliente puedan participar en el desarrollo de la especificación y validación de las pruebas.

Para hacer una pequeña demostración de este lenguaje natural se puede poner un ejemplo. En este caso se puede pensar en un escenario que defina el uso de un automóvil. Se quiere que al introducir la llave y girarla, el motor del automóvil comience a funcionar. Para este caso se puede pensar en introducir la llave como una precondición, girarla sería el evento que desencadena en el objetivo y, por lo tanto, que el motor funcione sería el resultado esperado. Para este caso la especificación sería como se muestra en la figura 1.

```
Scenario: El motor del automóvil funciona al arrancar
  Given introducimos la llave en el automóvil
  When giramos la llave
  Then el motor del automóvil comienza a funcionar
```

Figura 1. Especificación de arrancar un automóvil

En este ejemplo se puede apreciar el patrón que se utiliza en DGC, más conocido como patrón "*Given-When-Then*". En este patrón se definen las precondiciones (*Given*), los eventos (*When*) y el resultado esperado (*Then*). Actualmente, en la mayoría de herramientas que permiten el uso de GDC, es posible usar este patrón en nuestro idioma nativo, pero como el inglés es considerado el idioma internacional de la actualidad, se utilizará el mismo para esta parte.

Los conflictos que se solucionan al usar GDC frente a otras técnicas de desarrollo convencionales son:

1. Evita que se envíe algo no deseado, ya que si es necesario que se realice un cambio de requisitos, lo primero que se cambiará son las especificaciones y por

lo tanto el código actual no pasará las pruebas, haciendo así que deban cambiarse las partes obsoletas.

2. Partir de las especificaciones implica que en todo momento los desarrolladores poseen la información necesaria para poder avanzar con la siguiente parte, por lo que nunca sería posible desarrollar sin pensar la funcionalidad.

3. Como en este método se desarrolla la funcionalidad a partir de la especificación, nunca se desarrollará código que después no sea usado, ya que nunca se desarrollaría una parte del programa sin tener la correspondiente especificación de las pruebas.

4. Como consecuencia de los puntos anteriores, esta estrategia de desarrollo permite ir directos a los problemas y necesidades del desarrollo actual, por lo que ayuda a no perder tiempo de forma innecesaria.

2.2 Ingeniería del Software Dirigida por Modelos

La Ingeniería del Software Dirigida por Modelos [3] (ISDM) es un paradigma de desarrollo de software que se centra en los modelos y las transformaciones entre ellos para el desarrollo de software. Esto nos permite elevar el nivel de abstracción y evitar complejidades añadidas con los lenguajes de programación utilizados para el desarrollo de software.

En este paradigma, todos los elementos software deben contemplarse como modelos que interactúan entre ellos, formando de esta manera un nuevo modelo. Elevando el nivel de abstracción, un lenguaje de modelado debe ser considerado un modelo. En

ISDM se contempla esta opción y se denomina a dicho lenguaje como un metamodelo. Este aspecto se vuelve recursivo y para modelar un metamodelo, se deberá definir un meta-metamodelo.

Los primeros lenguajes de programación de alto nivel pretendían facilitar a los desarrolladores las dificultades generadas por el uso de los entornos en los que se desarrollaba el software. Además, la aparición de los sistemas operativos tenía como objetivo facilitar la experiencia de usuario, de esta forma no era necesario tener un conocimiento elevado sobre cómo comunicarse de forma directa con el hardware.

Aunque la abstracción generada por la creación de los lenguajes de programación fue muy positiva, el avance tecnológico nos ha llevado a aumentar el nivel de complejidad de las plataformas usadas para el desarrollo. También se debe tener en cuenta que a día de hoy, la mayoría de las aplicaciones que se desarrollan se mantienen de forma manual, lo que implica un gasto añadido de tiempo y esfuerzo. Esto no solo ocurre en la etapa de desarrollo, sino también en la de despliegue, configuración y validación.

En la actualidad, muchas de las soluciones que se plantean para este problema se centran en el desarrollo de tecnologías enmarcadas en el paradigma de la ISDM.

La ISDM tuvo su gran momento con la aparición de UML (*Unified Modeling Language*), una notación que nos permite especificar, visualizar y documentar los diferentes modelos de un sistema software. Las 4 funciones principales de los modelos son las siguientes:

1. Documentar el desarrollo del software.
2. Razonar sobre el sistema.

3. Comunicar ideas sobre las distintas partes del sistema.
4. Generar partes del sistema transformando estos modelos.

2.3 Herramientas similares

Anteriormente al inicio de este proyecto, se realizó una pequeña investigación con el fin de encontrar herramientas similares a la que se planteaba desarrollar, en este caso, para la generación automática de pruebas sobre modelos de software. Tras esta investigación se logró encontrar distintas herramientas con funcionalidades parecidas, pero ninguna idéntica. Tras ello, se empezó el desarrollo de la misma, ya que al no encontrar ninguna con la misma funcionalidad, se decidió que el desarrollo de esta era idóneo. A continuación se enumeran las herramientas encontradas más destacables de esta investigación, junto a sus cualidades y diferencias respecto a la herramienta desarrollada en este proyecto.

2.3.1 *Test Vector Generation System*

Test Vector Generation System [4] es una herramienta creada por *T-VEC Tchnologies*. Esta acepta requerimientos y comportamiento de un sistema en un lenguaje propio llamado T-VEC Linear Form.

Esta herramienta utiliza la teoría de pruebas de dominio [5], llamada *Domain Testing Theory* en inglés, para generar dichas pruebas. Además, analiza las expresiones lógicas en el modelo especificado, generando así pruebas automáticas que alcanzan los casos extremos dentro de cada dominio. Estas pruebas incluyen también las salidas esperadas.

A diferencia de la desarrollada en este proyecto, esta herramienta no tiene soporte para UML u otros lenguajes de modelado. Además, tampoco se basa en la estrategia DGC, que es el principal punto del desarrollo de este proyecto .

2.3.2 *Conformiq Creator 4.0*

Conformiq Creator 4.0 [6] es una herramienta desarrollada por *Conformiq Software Ltd.* Esta acepta diagramas de estado en UML como modelo del sistema bajo pruebas, además de soporte para propiedades a tiempo real.

Esta herramienta genera las pruebas de forma automática utilizando notación TTCN [7], las cuales podrán se probadas sobre el sistema posteriormente.

Aunque esta herramienta tiene soporte para UML, al igual que USE, no utiliza un lenguaje común para la generación de las pruebas, como utiliza la desarrollada en este proyecto, gracias a la utilización de la técnica de DGC. Uno de los aspectos a destacar de la herramienta explicada en este punto es que muestra la totalidad de las partes del modelo cubiertos por las pruebas generadas, funcionalidad que no posee la desarrollada en este proyecto.

2.3.3 *Reactis*

Reactis [8] es una herramienta desarrollada por *Reactive Systems Inc.* Esta acepta modelos en los lenguajes de modelado *SimuLink* [9] y *StateFlow* [10].

Esta herramienta permite la generación de pruebas de modo manual y automático. Además, muestra las partes del modelo cubiertas por las pruebas en función de la cobertura sintáctica y estructural del modelo.

A diferencia de la herramienta desarrollada en este proyecto, *Reactis* no posee soporte para UML o lenguajes similares, ni utiliza un lenguaje común, como se precisa en el uso de DGC. Además de esto, *Reactis* poseela función de mostrar las partes del código cubiertas por las pruebas, opción que no posee nuestra herramienta.

2.3.4 *Jcontract*

Jcontract [11] es una herramienta desarrollada por *The Parasoft Company*. Esta precisa del uso de *Design by Contract* [12], y utiliza el contrato como como modelo para predecir el comportamiento de la clase.

Jcontract junto a *Jtest* [11] proveen generación de pruebas y ejecución de forma automática, probando los caminos a partir de las precondiciones, y verificando con la ayuda de afirmaciones, invariantes y postcondiciones de cada método. Además, tiene soporte para calcular la cobertura de dichas pruebas en función de las ramas de las distintas ramas que se pueden seguir en el modelo.

La principal diferencia de *Jcontract* con la herramienta de este proyecto, al igual que las herramientas anteriores, es la falta del uso de un lenguaje natural.

2.3.5 Conclusión

Como ya se ha podido ver a medida que se han explicado las herramientas anteriores, existen gran variedad de estas, las cuales se basan en la generación de pruebas para modelos de forma automática.

Una diferencia que se ha podido contemplar al investigar sobre este tipo de herramientas, es que actualmente no hay ninguna desarrollada que se basen en el uso

de un lenguaje natural como utiliza la técnica DGC. Como el principal objetivo de este proyecto es el desarrollo de pruebas a partir del uso de un lenguaje natural, se ha visto como una herramienta altamente novedosa en este ámbito tras esta investigación llevada a cabo, por lo cual se decidió seguir adelante con este proyecto.

3

Herramientas utilizadas

En este capítulo se describirán las herramientas utilizadas para el desarrollo de este proyecto, poniendo especial énfasis en qué partes del proyecto se utilizan.

3.1 Maven

Maven [13] es una herramienta destinada a la mejora de la gestión y compresión de los proyectos software. Esta se basa en el concepto de un modelo de objeto de proyecto (POM, de sus siglas en inglés, *Project Object Model*), de esta forma, Maven

puede gestionar la creación de un proyecto, informar y documentar desde una pieza de información central.

El principal objetivo de Maven es facilitar a los desarrolladores a comprender el estado del esfuerzo de desarrollo en el menor tiempo posible. Para cumplir con este objetivo, Maven tiene que lidiar con los siguientes conceptos:

1. **Hace que el proceso de creación del proyecto sea sencillo.** Aunque sigue siendo necesario tener cierto conocimiento de los procesos fundamentales, Maven permite que el desarrollador desconozca ciertos detalles que no sean necesarios conocer.

2. **Provee un sistema de creación uniforme.** Maven utiliza POM y un conjunto de *plugins* para la creación de los proyectos, una vez que los desarrolladores se familiarizan con su uso, conocen como se crean todos los proyectos. Esto hace posible que se ahorre tiempo cuando se quiere navegar entre proyectos.

3. **Provee información de calidad sobre los proyectos.** Maven utiliza POM para facilitar información sobre el proyecto a los desarrolladores, creada en parte por los archivos del proyecto. Además, código de terceros añade sus informes a la información base dada por Maven.

4. **Anima a utilizar buenas prácticas.** Maven agrupa los principios actuales del concepto de buenas prácticas enfocadas en el desarrollo de software, de esta forma intenta guiar todos sus proyectos por este camino.

En este proyecto se ha utilizado Maven para generar proyectos de Cucumber vacíos, a partir de los cuales con la ayuda de la implementación de ciertos archivos se permite utilizar los beneficios de Cucumber y Gherkin para el desarrollo de pruebas de software para la herramienta USE.

3.2 IntelliJ IDEA

IntelliJ IDEA [14] es un entorno de desarrollo integrado (IDE, por sus siglas en inglés, *Integrated development Environment)* creado para facilitar el desarrollo de sistemas informáticos. Esta herramienta está diseñada para maximizar la productividad del desarrollador. Con la ayuda de sus asistencia de codificación inteligente y su diseño ergonómico, facilita que el desarrollo de software sea productivo y agradable. Este IDE está diseñado especialmente para su uso con Java, aunque también es posible utilizarlo para otros lenguajes de forma nativa o con la ayuda de distintos *plugins*. Además, permite integrar distintos *frameworks* que nos pueden ser de ayuda.

En este proyecto se ha utilizado este IDE como entorno de desarrollo, ya que permite integrar Gherkin y Cucumber, facilitando así su desarrollo, a la vez que acepta añadir implementación propia en lenguaje Java, por lo que ha sido la mejor opción frente a otros entornos de desarrollo disponibles. Además, al integrar Gherkin y Cucumber, permite navegar entre sus archivos de forma mucho más sencilla y dinámica.

3.3 USE

USE [15] (*UML-based Specification Environment*) es una herramienta de modelado utilizada para la especificar sistemas de información, basada en un subconjunto de UML [16][17] (*Unified Modeling Language*).

En USE se utiliza la definición de clases UML para describir los modelos de forma textual, lo que permite definir las clases con sus atributos, las relaciones entre ellas y las funciones típicas de un diagrama de clases UML. Para definir la integridad de los modelos se usa OCL [18] (*Object Constraint Language*), esto permite describir las distintas condiciones que debe cumplir el sistema antes y después de ejecutar distintas funciones sobre el mismo, a la vez que verifica la integridad del sistema, definiendo distintas restricciones que se deben cumplir en todo momento.

USE utiliza SOIL [19] (*Simple OCL-based Imperative Language*) para definir las llamadas que se ejecutan en el sistema. SOIL fue definido para su uso en USE como un lenguaje de programación imperativo. Sus sentencias se introducen a través de la línea de comandos que ejecuta la máquina virtual de Java para USE.

USE posee una herramienta de código abierto en la que se pueden visualizar modelos, al igual que nos ofrece funcionalidades extra como puede ser la visualización de los diagramas de secuencia y las máquinas de estado de los sistemas, entre otras cosas. En la figura 2 se puede ver el modelo de un microondas, en el cual se visualizan sus atributos, al igual que sus relaciones con otros componentes que posee el microondas.

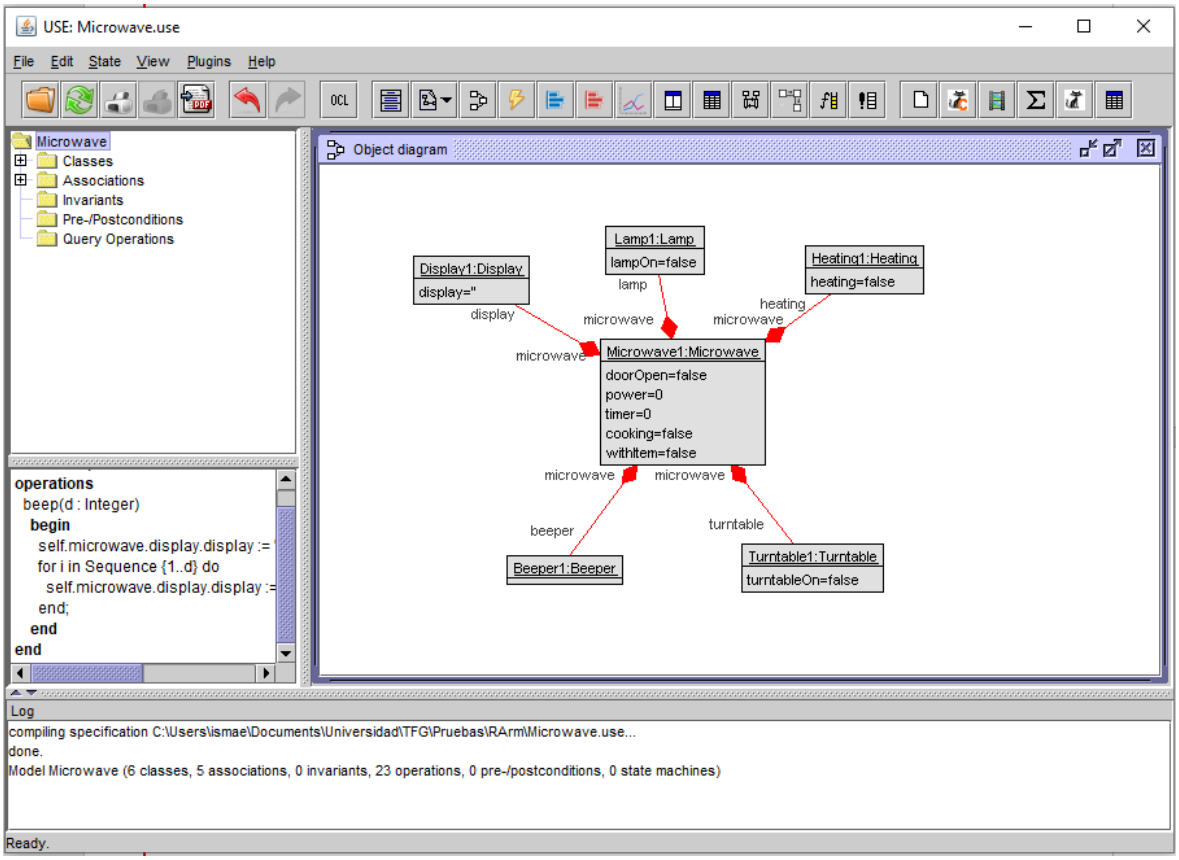

Figura 2. Ejemplo de la herramienta USE con un diagrama de objetos

La herramienta USE permite ejecutar distintas operaciones sobre los modelos, dichas operaciones deben ser descritas anteriormente. En la figura 3 se puede apreciar cómo se ejecutan las distintas operaciones. En este ejemplo se ejecutan sobre el microondas. Lo que se hace en este caso es abrir la puerta del microondas, introducir lo que se quiere calentar en él, se vuelve a cerrar la puerta e se incrementa la potencia y el tiempo. En la figura 4 se puede apreciar que el estado del microondas ha cambiado tras ejecutar estas operaciones.

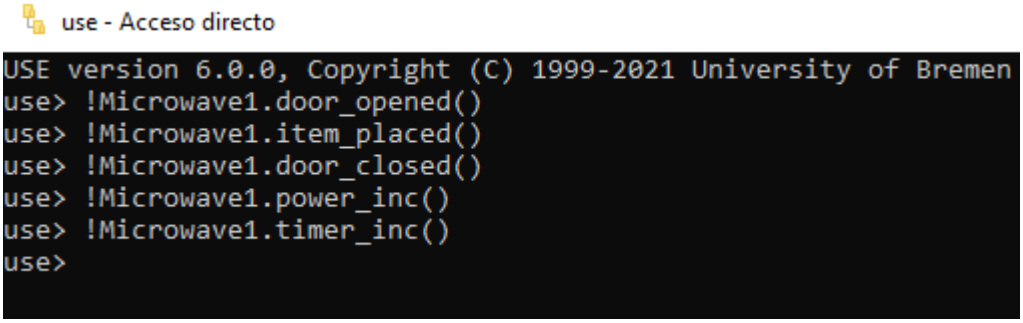

Figura 3. Ejemplo de ejecución de operaciones.

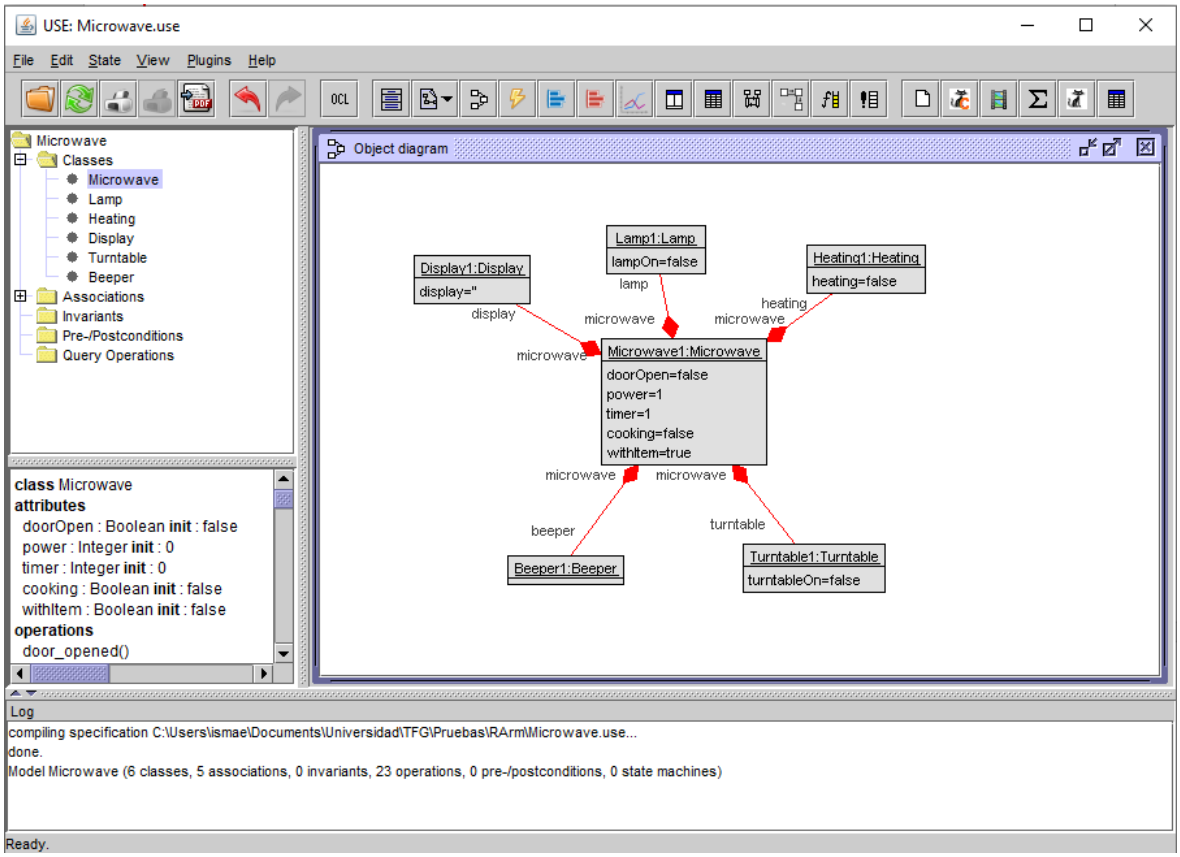

Figura 4. Ejemplo del microondas tras ejecutar diversas funciones

En este proyecto USE tiene uno de los papeles más importantes, debido a que es la herramienta sobre la que se quiere trabajar, ya que el objetivo principal del mismo es la creación de una aplicación que permita generar pruebas de forma sencilla y automática sobre distintos modelos de USE.

3.4 Gherkin

Gherkin [20] es un Lenguaje Específico de Dominio (DSL, de sus siglas en inglés, *Domain Specific Language*), es decir, un lenguaje que está creado para resolver un problema concreto.

Gherkin contiene una estructura generada por distintos elementos que definen su comportamiento. Estos elementos se pueden dividir en distintos grupos, que son los siguiente:

1. **Característica**
    a. **Feature:** este elemento define un fichero de Gherkin, se utiliza tanto de extensión de fichero como de palabra clave en el mismo, con la cual se define con brevedad la finalidad de dicho fichero.
2. **Comportamiento**
    a. **Scenario:** palabra clave utilizada para definir brevemente el escenario que se va a implementar.
    b. **Examples:** palabra clave que indica el comienzo de una serie de parámetros que se van a utilizar al ejecutar el escenario actual.
    c. **Scenario Outline:** palabra clave que indica que el siguiente escenario se ejecutará varias veces. Este tipo de escenario debe contener la palabra clave *Examples*, ya que esta definirá el número de veces que se ejecutará el escenario, en función de los parámetros.
    d. **Background:** palabra clave que indica uno o varios elementos que se ejecutarán juntos a los escenarios siguientes.
    e. **Rule:** palabra clave que define una regla de negocio que debe de ser implementada en un conjunto de escenarios que engloba.

3. **Acción**
    a. **Given:** palabra clave que define las precondiciones de un escenario
    b. **When:** palabra clave que define los eventos que van a ocurrir en el transcurso de un escenario.
    c. **Then:** palabra clave que define el resultado esperado del escenario.
    d. **And:** esta palabra clave se utiliza cuando hay varias palabras claves de acción del mismo tipo de forma correlativa, con lo cual no debemos repetirlas varias veces.
    e. **But:** esta palabra clave tiene el mismo uso que la palabra clave *And*, solo que se utiliza para casos en los que un paso implica algo opuesto al paso anterior, aunque podríamos usar *And* sin ningún tipo de problema.
4. **Símbolos**
    a. **Comillas '"':** se utiliza para denotar un tipo cadena de texto en nuestros escenarios.
    b. **Almohadilla '#':** se utiliza para denotar comentarios.
    c. **Arroba '@':** se utiliza para marcar escenarios.
    d. **Línea vertical '|':** se utiliza para separar los parámetros que se van a utilizar en un escenario.

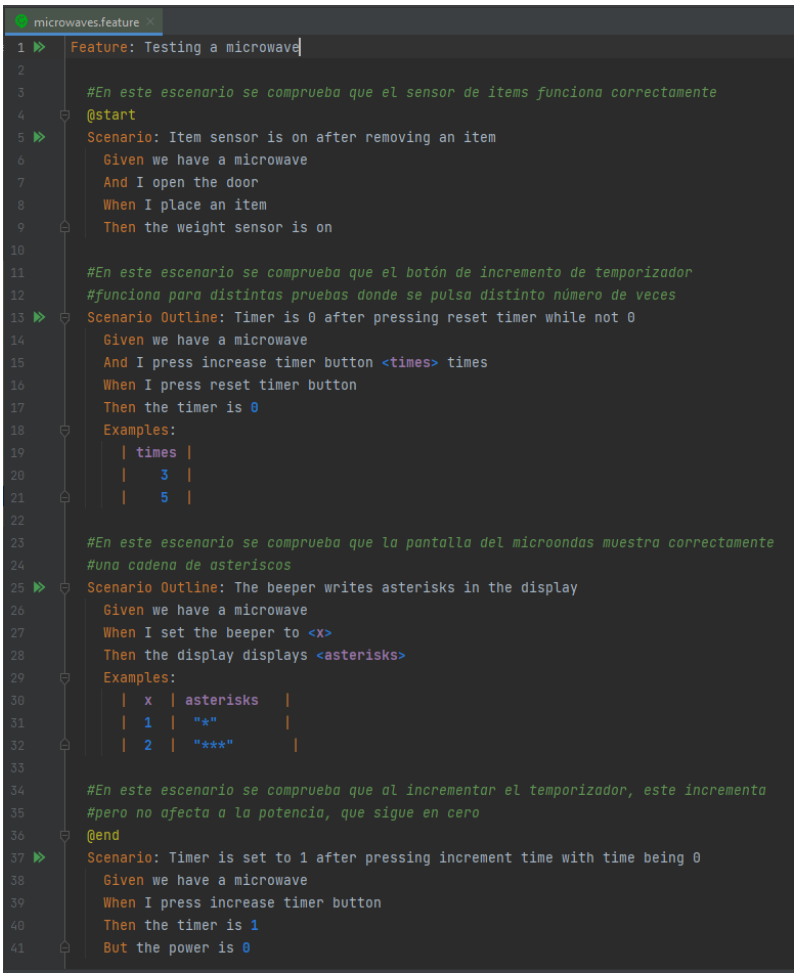

Figura 5. Ejemplo de sintaxis de Gherkin

En la figura 5 se muestra un ejemplo con varios escenarios, en el cual se utiliza la sintaxis de Gherkin como se puede apreciar. En este ejemplo se hacen varias pruebas para un microondas. Como se puede observar, estas pruebas utilizan el lenguaje natural, por lo que son bastante comprensibles para personas que no tienen conocimientos de lenguajes de programación.

En este proyecto se utiliza Gherkin para definir las pruebas que se quieren ejecutar sobre los modelos USE a validar.

3.5 Cucumber

Cucumber [21] es una herramienta de software que soporta Gherkin. Cucumber se utiliza para ejecutar los distintos escenarios especificados en sintaxis Gherkin enfocados al lenguaje de programación elegido, en nuestro caso Java.

En la herramienta Cucumber se definen los distintos pasos especificados en Gherkin. Estos pasos son los apartados de cada escenario marcados por cada una de las distintas palabras claves mencionadas anteriormente (*Given*, *Then*, *When*, *And* y *But*). Cuando ejecutamos un fichero Gherkin, con extensión *Feature*, lo que se hace es indicar a Cucumber qué operaciones debe ejecutar y en qué orden, de forma que las pruebas avanzan en el orden correcto, y así se evita que se compruebe un resultado antes de que se ejecuten todos los eventos predecesores.

En el caso de esta herramienta, un fichero Java implementa la funcionalidad de Cucumber añadiendo varias librearías. En este fichero se implementa el código correspondiente a cada uno de los distintos pasos especificados en Gherkin. En Cucumber, estos pasos tienen asignados funciones Java, una por cada uno. Los pasos y las funciones se relacionan con una pequeña especificación que se hace en la línea anterior a cada función. En la figura 6 se puede ver un ejemplo de un paso Gherkin relacionado con una función en Java. Esta función se ejecutará cuando deba ejecutarse dicho paso.

```
@Given("I place an item in the oven")
public void i_place_an_item_in_the_oven() {
   pw.println("m.item_placed();");
}
```

Figura 6. Ejemplo de función Java relacionada a Gherkin

En la figura 6 se ve la forma en la que Cucumber indica a que paso está relacionado esta función. Utilizando el símbolo '@', Cucumber indica a qué tipo de paso está asociado. Estos pasos pueden ser de tipo *Given*, *Then* y *When*, y no de tipo *And* y *But*, ya que estos dos últimos siempre representan a uno de los tres anteriores. Tras denotar que tipo de paso se va a ejecutar, Cucumber indica el predicado del mismo, ya que esta función está asociada al paso "*Given I place an item in the oven*".

En este proyecto se ha utilizado Cucumber para ejecutar las pruebas, específicamente en Java, ya que es un lenguaje que se ha usado con frecuencia a lo largo del grado. El uso dado a distintos componentes de Cucumber que han sido de ayuda en este proyecto serán explicados en la sección en la cual se explican los distintos inconvenientes encontrados en el desarrollo.

3.6 Java

Java [22] es un lenguaje de programación y una plataforma informática comercializada por primera vez en 1995. Este lenguaje de programación orientado a objetos tuvo como principal objetivo ser multiplataforma. Debido a esto, los ficheros Java deben ser ejecutados por una máquina virtual llamada JVM (*Java Virtual Machine*). De esta forma, cada sistema operativo tendrá su propia JVM, por lo que se podrán compartir ficheros de uno a otro.

Una de las grandes ventajas que poseía Java frente a otros lenguajes de programación era que JVM poseía un recolector de basura, permitiendo que los desarrolladores dejasen de preocuparse por este tipo de problemas y pudiesen desarrollar los programas de forma más eficiente.

En este proyecto Java se ha utilizado en dos ámbitos distintos. Primero, se utiliza ya que USE está desarrollado en Java, por lo cual de base era necesario. Segundo, se ha utilizado Java para implementar código propio que nos ha ayudado a crear la herramienta que propone este proyecto de forma satisfactoria.

4

Desarrollo de la solución

En este capítulo se hablará sobre cómo se ha desarrollado la solución, explicando principalmente el código que ha sido necesario desarrollar y por qué se ha hecho de esta forma. Además, se mostrará un breve ejemplo de prueba, para que sea más sencillo entender las partes importantes del código. Todos los pasos a seguir se explicarán de forma más detallada en el manual de usuario.

4.1 Generación de código

Para desarrollar esta herramienta ha sido necesario programar distintas funciones adicionales a las que nos aportaban las herramientas Gherkin y Cucumber. Estas funciones son utilizadas mayormente para poder pasar nuestras pruebas en lenguaje de USE a nuestro fichero Java y, que tras esto, fuese posible imprimir en nuestros ficheros USE y SOIL el código necesario para poder ejecutar y validar nuestras pruebas. Además de esto, también hay una parte de este código Java que se encarga de validar que nuestro fichero de Gherkin contiene una serie de elementos necesarios para el correcto funcionamiento de la herramienta.

En primer lugar se tiene un fichero Java llamado *soilToJava.java*, el cual posee distintas funciones que serán las que se usarán para estos pasos. Estas funciones se encuentran dentro de la clase formada en este fichero, también llamada *soilToJava*, aunque algunas de ellas son privadas, ya que sólo se ejecutan al usar otras funciones principales. Para la instanciación de esta clase es necesario pasarle varios parámetros, entre los cuales se encuentran los siguientes, todos de tipo cadena de texto:

- Nombre del fichero de Gherkin, con la extensión añadida, como por ejemplo puede ser *pruebas.feature*.
- Nombre del fichero Java que se usará para ejecutar las pruebas.
- Nombre del fichero de texto que usaremos para escribir las pruebas en código SOIL.
- Nombre del fichero de USE en el cual tenemos nuestro modelo desarrollado, al que queremos hacerle las pruebas.
- Nombre del fichero SOIL, el cual necesitamos para ejecutar las distintas pruebas individualmente.

Dentro de nuestra clase desarrollada, la primera función que se debe destacar es la llamada *checkFeatures*. Esta función es privada, ya que se ejecuta dentro de otra de la que hablaremos más adelante. Esta función se encarga de validar nuestro fichero *feature*, ya que hay varias restricciones que debe cumplir. Antes de explicar qué valida esta función, se debe entender por qué debe emplearse un elemento en especial dentro del ámbito de Gherkin, que en este caso serían los tags, usando el símbolo '@'.

Este elemento permite detectar cuándo se ejecutan pruebas específicas. En este caso se debe saber cuándo se ejecuta la primera y última prueba, ya que al ejecutarse en código Java, se precisa crear y eliminar distintos elementos de Java, como en este caso pueden ser *FileWriter*, *BufferedWriter* y *PrintWriter*. Esto se debe a que se necesita pasar estas pruebas a nuestros ficheros USE y SOIL, por lo cual es necesario crear una instancia de estos tipos al empezar las pruebas y de vuelta cerrarlas al terminar, de forma que no haya problemas a la hora de terminar de usar la herramienta.

Otro aspecto relacionado con este elemento sería en el caso de *@Before* y *@After*. En este caso no es necesario remarcarlos en el fichero Gherkin, ya que sólo debe especificarse en el fichero Cucumber. Estos elementos hacen referencia al momento de antes y después de cada escenario, respectivamente. Esto ayuda a varias cosas:

1. Permite recoger el nombre del escenario a ejecutar.
2. Permite ejecutar código Java antes de cada escenario, en este caso es beneficioso ya que así el usuario no debe tener en cuenta el código necesario en USE que se debe escribir al inicio de cada prueba.
3. Permite ejecutar código Java después de cada prueba, que es beneficioso en igual medida que el anterior, pero en este caso para cerrar cada prueba.

Una vez se tiene esto en cuenta, lo que hace la función *checkFeatures* es validar que nuestro fichero Gherkin posee los tags necesarios, en nuestro caso llamados *@start* y *@end*. Además de esto, también tiene en cuenta que el primer y último escenario sean tipo *Scenario* y no *Scenario Outline*, por ejemplo, ya que en el caso de este último se ejecutarían varias veces la instanciación o cierre de los elementos usados en Java, produciendo así que el uso de esta herramienta no fuese del todo efectivo.

Esta función se ejecuta dentro de otra función pública llamada *getJavaFromSoil*. Esta función se encarga de sobrescribir un fichero Java. Este fichero será posteriormente utilizado por Cucumber para generar las pruebas. Esta función evita que el usuario final deba realizar ciertas operaciones que en este caso pueden ser automatizadas, ya que en todos los casos son iguales. En este caso son:

- Instanciación de elementos necesarios en Java al inicio de ejecutar las pruebas (*FileWriter*, *BufferedWriter* y *PrintWriter*).
- Código necesario antes y después de cada prueba en USE, que debe ejecutarse antes y después de cada escenario.
- Para cada línea de código USE, se encarga de pasarlo a formato de línea de texto y añadirlo a la función de *PrintWriter*, en el fichero Java, que se encarga de imprimirlo en el fichero USE. Esta parte usando una pequeña función privada llamada *strAddPrint*.
- Imprimir en el fichero SOIL la llamada a cada función de USE que ejecuta las pruebas.
- Cierre de los elementos instanciados al terminar las pruebas.

Por último se ha implementado otra función pública, en este caso llamada *createTXTFromJava*. Esta función se encarga de imprimir nuestro fichero Java, el

que se ha creado en la primera parte del uso de la herramienta, en un fichero de texto, siendo este fichero Java el que se utilizará para ejecutar las pruebas en Cucumber. La forma de cómo se debe generar este fichero para Cucumber se explicará en el apartado del manual de usuario, ya que no es algo que tenga relación con cómo se ha desarrollado esta herramienta.

La razón por la cual se ha decidido implementar estas últimas funciones es debido a que queremos que esta herramienta sea lo más automática y sencilla posible, por lo que para esto se debía evitar que el usuario final implementase cosas que se podían automatizar, y que sobre todo, pudiese escribir las pruebas directamente en lenguaje utilizado por USE, sin necesidad de añadir elementos Java por sí mismo.

Para hacer esto posible ha sido necesario utilizar un fichero de texto plano, ya que en este se puede escribir en cualquier lenguaje sin error, a diferencia que en Java, que si se escribe en un lenguaje no reconocido no permitiría ejecutar el fichero. De esta forma acepta definir nuestras pruebas en el lenguaje entendido por USE, y tras esto, nuestra implementación se encarga de generar el código necesario para pasar estas pruebas a nuestro fichero USE.

4.2 Ejemplo de uso

En esta sección se muestra un breve ejemplo del uso de esta herramienta, de forma que sea más sencillo comprender lo explicado en la sección anterior. En esta sección se explican directamente los pasos relacionados con las partes implementadas en este proyecto, ya que las partes relacionadas con Gherkin y Cucumber serán explicadas de forma detallada en el manual de usuario.

Para este ejemplo se va a implementar un modelo sencillo de un coche. En este ejemplo, la clase del coche tendrá varios atributos, en este caso habrá uno llamado *carIsOn*, el cual es de tipo *Boolean*, indicando si el coche está encendido o apagado. Otro atributo será *speed*, el cual indica la velocidad del coche. Además de esto, este modelo tendrá 3 funciones distintas, la primero que es *turnOnOff*, encargada de encender o apagar el coche en función de varias restricciones que se explicarán después. Las otras 2 funciones serán *speedUp* y *speedDown*, las cuales acelerarán y decelerarán el coche respectivamente. Sobre las restricciones tendremos las siguientes:

- Si la velocidad del coche es mayor a cero, este no se podrá apagar.
- Una vez que la velocidad del coche sea cero, la función *speedDown* no bajará el valor del atributo relacionado con la velocidad.
- La última de las restricciones será que no se podrá aumentar la velocidad del coche si este no está encendido.

En la figura 7 se muestra el diagrama de clases de este ejemplo.

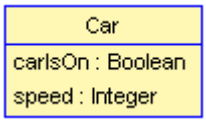

Figura 7. Diagrama de clase del ejemplo del coche

Siguiendo la metodología DGC, lo primero que se debe hacer es implementar las pruebas necesarias, antes de haber implementado el modelo. Para este ejemplo se han implementado los escenarios que se muestran en la figura 8.

```
Feature: Testing a car

  @start
  Scenario: The car turns on
    Given we have a car
    When we turn on the car
    Then the car is turned on

  Scenario: The car turns off
    Given we have a car
    When we turn on the car
    And we turn off the car
    Then the car is turned off

  Scenario: The car speeds up
    Given we have a car
    When we turn on the car
    And we speed up the car
    Then the speed is equal to 1

  @end
  Scenario: The car speeds down
    Given we have a car
    When we turn on the car
    And we speed up the car
    And we speed down the car
    Then the speed is equal to 0
```

Figura 8. Implementación de los distintos escenarios en Gherkin

Una vez se tienen los distintos escenarios definidos, se deben implementar todos los pasos en el fichero *StepDefinitions*. La forma de hacerlo se explicará más adelante como se dijo anteriormente. En la figura 9 se muestra cómo quedaría este fichero.

```java
public class StepDefinitions {
    @Given("we have a car")
    public void we_have_a_car() {

    }
    @When("we turn on the car")
    public void we_turn_on_the_car() {

    }
    @Then("the car is turned on")
    public void the_car_is_turned_on() {

    }
    @When("we turn off the car")
    public void we_turn_off_the_car() {

    }
    @Then("the car is turned off")
    public void the_car_is_turned_off() {

    }
    @When("we speed up the car")
    public void we_speed_up_the_car() {

    }
    @Then("the speed is equal to {int}")
    public void the_speed_is_equal_to(Integer int1) {

    }
    @When("we speed down the car")
    public void we_speed_down_the_car() {

    }
}
```

Figura 9. Implementación de los pasos en Cucumber

Una se ha creado este fichero vacío, se debe ejecutar el fichero *soilToJava*. Al ejecutar pedirá los siguientes parámetros:

1. Nombre del fichero *feature*.
2. Nombre del fichero Java.
3. Nombre del fichero de texto en el que implementaremos las pruebas de USE.
4. Nombre del fichero USE.
5. Nombre del fichero SOIL.

Tras esto, la herramienta da a elegir entre distintas opciones. Para este caso se requiere seleccionar la primera opción, la cual creará un fichero de texto en el cual se deben implementar los distintos pasos anteriores en el lenguaje utilizado por USE. En la figura 10 se puede ver la ejecución para este ejemplo.

```
Indique el fichero feature: car.feature
Indique el fichero java: StepDefinitions.java
Indique el fichero de texto: car.txt
Indique el fichero USE: Car.use
Indique el fichero SOIL: Car.soil
Seleccione una opción:
1 - Crear fichero de texto.
2 - Pasar fichero de texto a java.
3 - Salir.
```

Figura 10. Ejemplo de ejecución del fichero *soilToJava*

Una vez se haya ejecutado este fichero, eligiendo la primera opción, se creará un fichero de tipo texto (con el mismo nombre que se le ha asignado en esta ejecución) junto al fichero *StepDefinitions*. En este fichero de texto es necesario especificar las distintas pruebas en lenguaje de USE. En la figura 11 se muestra la definición de alguno de los pasos para este ejemplo.

```
@Given("we have a car")
public void we_have_a_car() {
    declare c:Car;
    c:= new Car();
}
@When("we turn on the car")
public void we_turn_on_the_car() {
    c.turnOnOff();
}
@Then("the car is turned on")
public void the_car_is_turned_on() {
    result := c.carIsOn;
}
@When("we turn off the car")
public void we_turn_off_the_car() {
    c.turnOnOff();
}
```

Figura 11. Implementación de algunos pasos para el modelo del coche

Una vez se haya completado este paso, es necesario volver a ejecutar el fichero *soilToJava* igual que en la ejecución anterior, pero esta vez seleccionando la segunda opción. En este caso se sobrescribirá el fichero Java *StepDefinitions*, añadiendo el código necesario para imprimir las pruebas a los fichero USE y SOIL al ejecutar de nuevo el fichero *feature*.

Una vez se haya finalizado el paso anterior, todo estará listo para volver a ejecutar nuestro fichero *feature*. Si todo ha salido bien, la herramienta no debería devolver ningún error, y al abrir nuestro fichero USE se podrá comprobar que se han escrito todas estas pruebas al final del mismo, al igual que se habrán escrito las llamadas a todas estas pruebas en el fichero SOIL. En las figuras 11 y 12 se muestran dos pequeñas capturas de los ficheros USE y SOIL tras la ejecución de las pruebas, respectivamente.

```
--Test: The car turns on
Test1(): Boolean
begin
declare c:Car;
c:= new Car();
c.turnOnOff();
result := c.carIsOn;
end
post: result

--Test: The car turns off
Test2(): Boolean
begin
declare c:Car;
c:= new Car();
c.turnOnOff();
c.turnOnOff();
result := not c.carIsOn;
end
post: result
```

Figura 12. Ejemplo de pruebas implementadas en fichero USE

```
reset
!new Test('t')
!t.Test1();
!t.Test2();
!t.Test3();
!t.Test4();
check
```

Figura 13. Ejemplo de llamada a pruebas en fichero SOIL

Una vez que todos estos pasos se han completado con éxito, es hora de pasar a la parte importante, comprobar que el modelo pasa todas las pruebas. Para este ejemplo se puede ver un objeto del tipo coche en la figura 14, al igual que es posible ver la implementación completa de este modelo en la figura 15. En la figura 16 se muestra

la salida de la ejecución, que en este caso muestra que ha habido 0 fallos, validando así todas las pruebas implementadas.

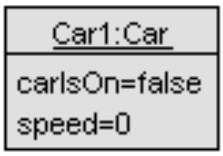

Figura 14. Objeto tipo coche en USE

```
model Car

class Car
  attributes
    carIsOn:Boolean init:false
    speed:Integer init:0
  operations
    turnOnOff()
        begin
            if (self.carIsOn=true and self.speed=0) then
            self.carIsOn := false else
            self.carIsOn := true;
            end;
        end
   speedUp()
        begin
            if self.carIsOn = true then
            self.speed := self.speed+1;
            end;
        end
    speedDown()
        begin
            if (self.carIsOn = true and self.speed > 0)
            then self.speed := self.speed-1;
            end;
        end
end
```

Figura 15. Implementación del modelo del coche en USE

```
use> open Car.soil
Car.soil> reset
Car.soil> !new Test('t')
Car.soil> !t.Test1();
Car.soil> !t.Test2();
Car.soil> !t.Test3();
Car.soil> !t.Test4();
Car.soil> check
checking structure...
checked structure in 0ms.
checking invariants...
checked 0 invariants in 0.001s, 0 failures.
```

Figura 16. Ejemplo de validación de todas las pruebas.

En el caso en que algunas de las pruebas no se validase, la propia ejecución pararía en dicha prueba, permitiendo así saber en qué prueba se encuentra el error. Se vería algo similar a la figura 17, donde se produce un fallo en la prueba 2.

```
use> open Car.soil
Car.soil> reset
Car.soil> !new Test('t')
Car.soil> !t.Test1();
Car.soil> !t.Test2();
[Error] 1 postcondition in operation call `Test::Test2(self:t)' does not hold:
  post2: result
    result : Boolean = false

  call stack at the time of evaluation:
    1. Test::Test2(self:t) [caller: t.Test2()@<input>:1:0]

+-----------------------------------------------------------------+
| Evaluation is paused. You may inspect, but not modify the state. |
+-----------------------------------------------------------------+
```

Figura 17. Ejemplo de fallo en la validación de las pruebas.

5

Validación y pruebas

En este capítulo se presentan las distintas pruebas que se han llevado a cabo para la validación de que la herramienta funciona de forma correcta y esperada.

5.1 Pruebas realizadas

Para este proyecto en concreto, al tratarse de una herramienta para generación automática de pruebas, la mejor forma de realizar las pruebas es probándola con distintos ejemplos, por lo que se han seleccionado varios, a partir de los cuales se ha probado y, poco a poco, se han ido cambiando algunos aspectos de la misma, de forma que se ha conseguido finalizar con una herramienta completa y funcional.

5.1.1 Brazo robótico

Como primer ejemplo se ha utilizado el modelo de un brazo robótico. Este modelo se basaba en el uso de distintos brazos, los cuales podían moverse por ciertas coordenadas que todos compartían, de forma que dos brazos no pueden estar en la misma posición. Además de esto, también existen distintos *items*, los cuales pueden ser agarrados y transportados por los brazos. Como estos *items* también comparten las mismas coordenadas, otra de las restricciones era que dos *items* no pueden estar en la misma posición ni ser agarrados por dos brazos diferentes. Este ejemplo posee también dos clases que no se han mencionado, la primera, llamada *PositionedElement*, de la que heredan tanto la clase del brazo robótico como la clase del *item*, que permite a ambos objetos poseer atributos de sus coordenadas. La segunda clase se llama *Parameters*, que está relacionada con cada brazo robótico e indica la posición máxima que puede alcanzar dicho brazo.

Para entender este ejemplo de forma más detallada, en la figura 17 se muestra el diagrama de clases, a su vez, en la figura 18, se muestra un ejemplo de un diagrama de objetos para dos brazos, uno con un *item* agarrado y otro sin él. Para este caso se han realizado más de 90 pruebas diferentes, aunque algunas eran modificaciones de otras, como por ejemplo para validar el correcto movimiento de los brazos, ya que se prueban para varios desplazamientos, como puede ser para 5 como para 7 posiciones desplazadas en cualquier dirección de las 3 dimensiones. Las pruebas más relevantes han sido:

- Desplazamiento correcto de forma ascendente, frontal y lateral.
- El brazo no agarra ningún *item* al ejecutar la función "agarrar" del mismo si no hay ningún *item* en la misma posición.

- El brazo agarra un único *item* al agarrar teniendo un *item* en su misma posición.
- Cuando un brazo tiene un *item* agarrado y se mueve, dicho *item* se mueve las mismas posiciones.

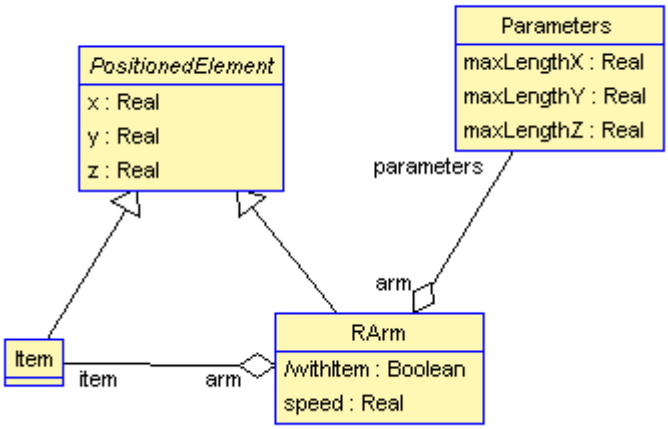

Figura 17. Diagrama de clase del brazo robótico

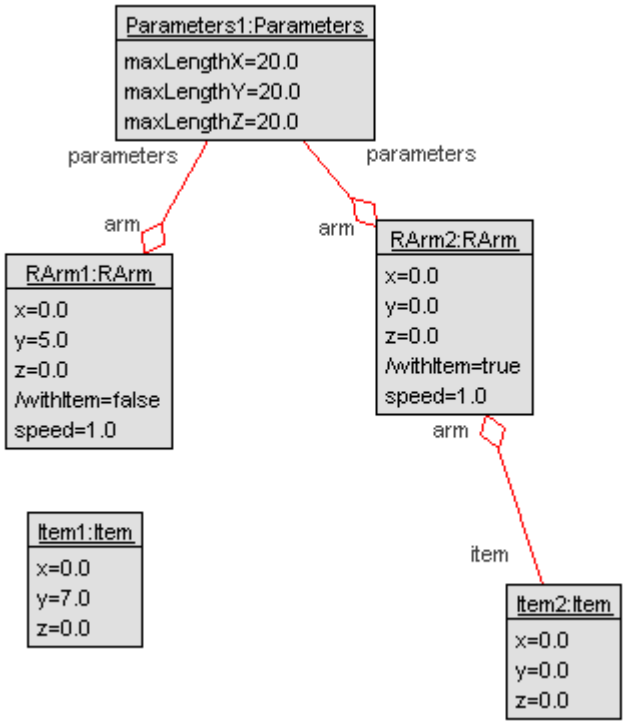

Figura 18. Ejemplo de diagrama de objetos para dos brazos

5.1.2 Microondas

Como segundo ejemplo se ha utilizado el modelo de un microondas, ya que es un ejemplo tomado de un artículo [1] sobre el uso de la técnica DGC, por lo que era una de las mejores opciones. En este ejemplo se tienen varios elementos relacionados entre sí, que hacen referencia a distintas partes del microondas. Todos estos elementos se relacionan al elemento principal, que en este caso sería el controlador. Estos elementos son:

- Controlador, como elemento conector de los demás. Posee dos atributos de tipo entero que marcan la potencia y el tiempo restante del microondas.
- Parámetros de entrada del usuario, que se compone por 3 elementos distintos que definen los 3 botones principales, los cuales interactúan con el usuario, estos son:
    - Potencia - permite aumentar y decrementar, igual que volver al valor por defecto.
    - Temporizador - igual que la potencia, pero con el valor del tiempo restante.
    - Cocinar - permite poner el marcha el microondas, al igual que pararlo en cualquier momento.
- Puerta que permite al usuario abrirla y cerrarla en cualquier momento.
- Sensor que detecta si hay algo introducido en el microondas.
- Reloj encargado de hacer avanzar el tiempo. En este caso posee una función llamada *tick*. Por cada *tick* pasa un segundo, lo que va actualizando el tiempo restante del temporizador.
- Lámpara que se enciende o apaga en función de distintos eventos, como pueden ser que la puerta esté abierta o que el microondas esté funcionando.

- Calor relacionado con la potencia. Su valor viene definido por los parámetros de entrada. También posee varias acciones a partir de otros eventos, como pueden ser activarse cuando el microondas está en funcionamiento como desactivarse cuando este se pare.
- Pantalla que puede mostrar distintos mensajes. En este caso su función principal es mostrar el tiempo restante del temporizador.
- Plato giratorio que funciona a partir de distintos eventos. En este caso se activa cuando el microondas está en funcionamiento y está quieto en caso contrario.
- Campana. Su función es avisar cuando el temporizador haya llegado a cero.

Para este ejemplo las pruebas a implementar fueron más complejas, ya que el microondas contempla muchas operaciones que dependen de otras, mayormente de seguridad. Por ejemplo, cuando el microondas está en funcionamiento y se abre la puerta, el microondas debe parar su funcionamiento de forma inmediata. Los eventos que deberían accionarse en este caso serían:

- El elemento que provee calor debe desactivarse.
- El plato giratorio debe detenerse.
- El temporizador debe pararse.

Este tipo de acciones hizo que debiesen hacerse más comprobaciones para cada escenario, lo que fue de gran ayuda ya que también validó que la herramienta funcionase correctamente para casos más complejos.

Ya que este ejemplo era bastante complejo se decidió facilitarlo un poco, implementando alguno de estos elementos incluidos dentro del propio controlador, en

este caso llamado *Microwave*, como se muestra en el diagrama de clases de la figura 19. A su vez, aparece un ejemplo de muestra de una instanciación de objetos en la figura 20.

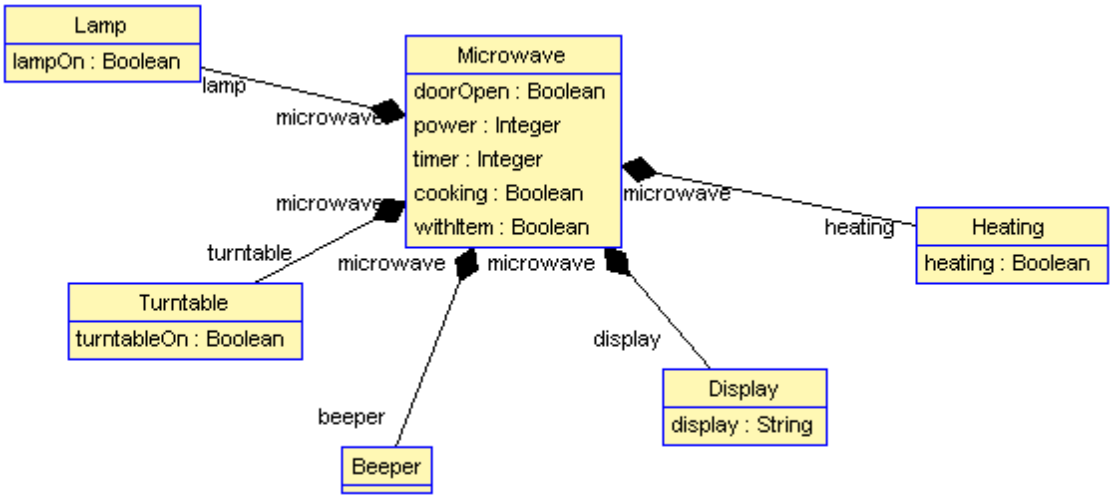

Figura 19. Diagrama de clases del modelo del microondas

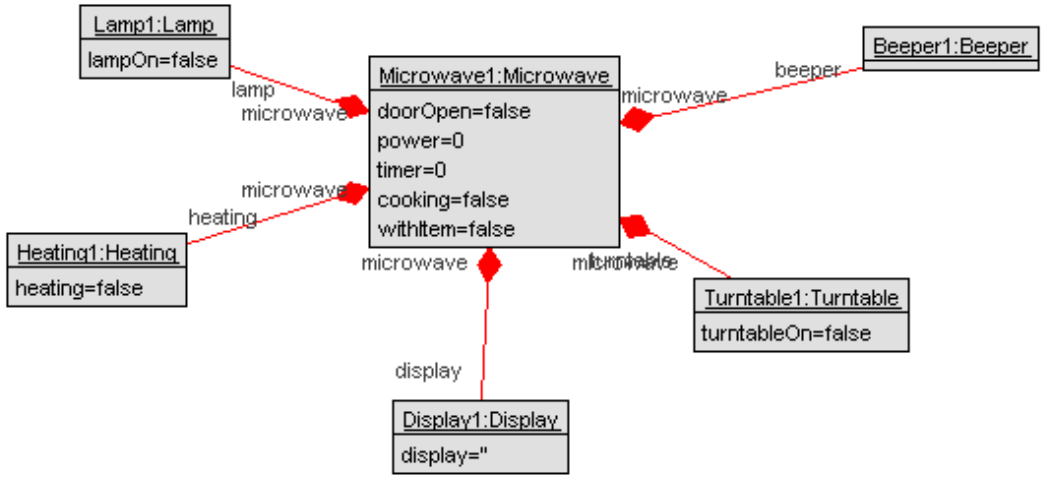

Figura 20. Ejemplo de prueba para el caso del microondas

6

Conclusiones

En este capítulo se desarrollarán las conclusiones obtenidas a medida que se ha desarrollado el proyecto y tras su finalización. En la primera sección se hablará sobre distintos conocimientos adquiridos gracias a la realización de este proyecto, al igual que las dificultades presentes. En la segunda sección se explicará una línea futura que podrá ser implementada para mejorar distintos aspectos de la herramienta.

6.1 Conocimientos adquiridos

El desarrollo de este proyecto ha sido de gran ayuda para adquirir nuevos conocimientos en distintos ámbitos relacionados con el modelado y el desarrollo del software.

En primer lugar, gracias al estudio y utilización del Desarrollo Guiado por Comportamiento, se han entendido los pilares fundamentales de esta técnica, comprendiendo así por qué puede llegar a ser tan útil para el desarrollo, principalmente debido a que evita mucho esfuerzo innecesario al prevenir que haya errores en una fase del desarrollo muy avanzada. Gracias a esto, una de las primeras conclusiones obtenidas es sobre la poca importancia que se le da a este tipo de técnica, ya que aunque se ha dado alguna asignatura sobre pruebas de software a lo largo del grado, estas pruebas siempre se han planteado una vez que se ha terminado el desarrollo del software, lo que implica que para cada fallo encontrado se pueda perder mucho tiempo adicional al intentar solucionarlo, mientras que con esta técnica esto no ocurriría, ya que se encontrarían a medida que se desarrolla el software.

En segundo lugar, se ha aprendido más sobre la Ingeniería del Software Dirigida por Modelos. Sobre este paradigma ya se ha hablado en alguna asignatura del grado, sobre todo sobre las ventajas que presenta frente a otras formas de desarrollo. Entre estas ventajas se encuentran las siguientes:

- Maximiza la compatibilidad entre los sistemas.
- Simplifica el proceso de diseño.
- Promueve la comunicación entre los individuos y equipos que trabajan en el sistema.

Gracias al desarrollo de esta herramienta se ha podido comprobar que DGC también es una forma de desarrollo muy útil que se puede aplicar sobre modelos, y que es algo que debería estudiarse más en profundidad, ya que puede ser un gran avance de cara al futuro.

Como conclusión final cabe destacar que el desarrollo de este proyecto ha sido una experiencia que ha aportado aspectos positivos al desarrollo profesional, que pueden ser de gran beneficio para futuras oportunidades, ya que se ha aprendido con más profundidad sobre los distintos aspectos contenidos en el mismo.

6.2 Líneas futuras de ampliación

Tras la realización de este proyecto se proponen las siguientes ampliaciones.

En primer lugar, y debido a que esta herramienta está pensada para ser usada en USE, se propone su adaptación al mismo, es decir, incluir en la propia herramienta USE la posibilidad de crear pruebas basándose en DGC. Esto se podría conseguir creando un plugin para la misma, que podría ser desarrollado de forma no muy compleja.

Actualmente esta herramienta se encuentra externa a USE, por lo cual al añadirla a la misma, su uso debería ser aún más sencillo. Además, no sería necesaria la instalación de componentes externos, como pueden ser Maven o IntelliJ IDEA, excluyendo a Gherkin y Cucumber, ya que estos serían necesarios para el análisis inicial del código. En este caso, exceptuando Gherkin y Cucumber, sólo se necesitaría instalar USE y el JDK de Java, ya que este último es necesario para poder usar la herramienta USE.

Por otra parte, y ya que algunas de las herramientas nombradas en el capítulo 3 lo poseen, se propone la introducción de una funcionalidad que permita al usuario saber qué partes del código está siendo cubierto por las pruebas generadas, de forma que

siempre será posible tener en cuenta las distintas partes del código no probadas y añadir pruebas nuevas que puedan cubrirlas.

Referencias

[1] T. Mens, A. Decan y N. Spanoudakis, "A method for testing and validating executable statechart models". *Softw Syst Model* 18, 837-863, 2019. Disponible en: https://link.springer.com/article/10.1007/s10270-018-0676-3

[2] M. Wynne y A. Hellesøy, *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Dallas, Tex.: Pragmatic Bookshelf, 2012.

[3] M. Brambilla, J. Cabot y M. Wimmer, *Model-Driven Software Engineering in Practice*, 2ª ed.: Morgan & Claypool, 2019.

[4] "T-VEC Vector Generation System - T-VEC Wiki", *T-vec.com*, 2009. Disponible en: http://www.t-vec.com/wiki/index.php?title=T-VEC_Vector_Generation_System.

[5] C. Kaner y S. Padmanabhan, *An introduction to the theory and practice of domain testing*. HCMC, Vietnam: VistaCom, 2010. Disponible en: http://kaner.com/pdfs/kanerPadmanabhanVISTACONDomainTesting.pdf.

[6] K. Nupponen, "Test Generation | Conformiq", *conformiq.com*, 2018. Disponible en: https://www.conformiq.com/tag/test-generation/.

[7] "TTCN - Wikipedia", *en.wikipedia.com*, 2008. Disponible en: https://en.wikipedia.org/wiki/TTCN.

[8] "Reactis: The Reactis API", *reactive-systems.com*, 2020. Disponible en: https://www.reactive-systems.com/doc/reactis-for-simulink/user018.html.

[9] "Simulink - Simulación y diseño basado en modelos", *es.mathworks.com*, 2018. Disponible en: https://es.mathworks.com/products/simulink.html.

[10] "Stateflow", *es.mathworks.com*, 2018. Disponible en: https://es.mathworks.com/products/stateflow.html.

[11] "Integrated Java Testing Tool for Application Software Development", *parasoft.com*, 2021. Disponible en: https://www.parasoft.com/products/parasoft-jtest/.

[12] "Design by Contract Introduction - Eiffel Software - The Home of EiffelStudio", *Eiffel Software - The Home of EiffelStudio*. Disponible en: https://www.eiffel.com/values/design-by-contract/introduction/

[13] M. Loukides, *Maven: the definitive guide*. Beijing: O'Reilly, 2008.

[14] *IntelliJ IDEA*. JetBrains, 2021. Disponible en: https://www.jetbrains.com/es-es/idea/.

[15] M. Gogolla, F. Büttner y M. Richters, "USE: A UML-based specification environment for validating UML and OCL". *Science of Computer Programming* 69, 27-32, 2007. Disponible en: https://www.researchgate.net/publication/222594748_USE_A_UML-based_specification_environment_for_validating_UML_and_OCL.

[16] O. M. Group, *UML 2.0 Superstructure*. 2005. Disponible en: https://www.omg.org/spec/UML/2.0/Superstructure/PDF.

[17] O. M. Group, *UML 2.0 Infrastructure*. 2005. Disponible en: https://www.omg.org/spec/UML/2.0/Infrastructure/PDF.

[18] O. M. Group, *Object Constraint Language 2.4*. 2003. Disponible en: https://www.omg.org/spec/OCL/2.4/PDF

[19] USE-OCL, *SOIL*. 2014. Obtenido en: http://useocl.sourceforge.net/w/index.php/SOIL#:~:text=SOIL%20is%20the%20imperative%20programming,unchanged%20by%20the%20imperative%20language.

[20] "Gherkin Syntax - Cucumber Documentation", *cucumber.io*, 2019. Disponible en: https://cucumber.io/docs/gherkin/.

[21] "Cucumber - Cucumber Documentation", *cucumber.io*, 2019. Disponible en:

https://cucumber.io/docs/cucumber/.

[22] H. Schildt, *Java: The Complete Reference, Eleventh Edition*. Oracle, 2018.

Apéndice A

Manual de Instalación

En las siguientes páginas se enumerarán las herramientas necesarias para el uso de esta herramienta y cómo instalarlas. Para este proyecto se ha usado el sistema operativo Windows, por lo que la instalación será explicada para el mismo. En otros sistemas operativos puede ser diferente.

A.1 Java JDK

Como ya se ha mencionado en capítulos anteriores, algunas de las herramientas precisan de la instalación del Java Development Kit para su uso. Para este proyecto se ha usado la versión 1.8.

En primer lugar deberemos dirigirnos a https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html. En esta página, el usuario debe dirigirse a la sección correspondiente a la versión que

desea descargar, dirigiéndose también a su sistema operativo. Tras descargarlo, solo bastará con ejecutarlo y seguir las instrucciones del instalador.

Una vez que JDK ha sido instalado correctamente, se deberá añadir a las variables del sistema para poder ejecutarlo desde cualquier directorio. Para esto, primero se debe buscar la carpeta de instalación del JDK y copiar su dirección. Una vez se haya hecho esto, el usuario deberá dirigirse a: Este equipo > Propiedades > Configuración avanzada del sistema > Variables de entorno. Aquí debe buscar la variable llamada *PATH*, y en el caso de no tenerla, crearla. Una vez le haya dado a editar, debe añadir la dirección que ha copiado anteriormente, en el caso de que no esté.

Además de esto, también será necesario añadir otra variable de sistema llamada *JAVA_HOME*. Para este paso es necesario seguir las instrucciones anteriores, pero en este caso, llamar a dicha variable *JAVA_HOME*. Este paso es necesario ya que se necesita esta variable de entorno para poder instalar Maven posteriormente.

A.2 Maven

La segunda herramienta necesaria es Maven, ya que será necesaria para crear el proyecto de Gherkin y Cucumber. Aunque hay otras herramientas para la creación de este proyecto, se ha elegido Maven ya que es más sencilla de utilizar.

En primer lugar hay que dirigirse a https://maven.apache.org/download.cgi, donde habrán distintos archivos para descargar, de los cuales el usuario debe seleccionar el que prefiera. Una vez descargado, debe descomprimir el archivo en cualquier directorio. Tras esto, debe buscar la carpeta llamada *bin* y copiar su dirección. Una

vez haya copiado esta dirección, debe volver a: Este equipo > Propiedades > Configuración avanzada del sistema > Variables de entorno. Aquí debe editar de nuevo la variable *PATH*, pero esta vez añadiendo el directorio de la carpeta *bin* de Maven.

Para asegurar que Maven está correctamente instalado, es posible ejecutar el comando mvn -v, el cual debe devolver información sobre Maven y Java.

A.3 IntelliJ IDEA

Para el desarrollo de este proyecto se ha utilizado IntelliJ IDEA Community Edition 2021.1.1. Para descargar e instalar esta versión, hay que dirigirse a https://www.jetbrains.com/es-es/idea/download/other.html. En esta página se debe buscar dicha versión y una vez descargado, seguir los pasos del instalador será suficiente, aunque es recomendable seleccionar los siguientes apartados en opciones de instalación:

- Add launcher dir to the PATH
- Add "Open Folder as Project"

Una vez terminada la instalación, IntelliJ IDEA debería poder usarse sin problema.

A.4 Gherkin y Cucumber for Java

Para instalar estas dos herramientas, sólo es necesario tener abierto IntelliJ IDEA. Una vez esté abierto , se debe pulsar *Control+Alt+S*, esto abrirá una ventana llamada *Settings*. Aquí es necesario buscar el apartado llamado *Plugins*, y dentro de este se debe buscar *Gherkin*. Una vez se haya encontrado solo será necesario pinchar en el botón de instalación. Tras esto, se debe buscar *Cucumber for Java* e instalarlo de igual manera. Una vez estén instaladas ambas extensiones, IntelliJ IDEA

recommendará cerrar y volver a abrirlo, lo que permitirá la correcta instalación de ambas.

A.5 USE

Lo primero que hay que hacer para instalar esta herramienta es dirigirse a https://sourceforge.net/projects/useocl/. Aquí se descargará un fichero .zip que se puede ubicar en cualquier carpeta, en la cual deberá descomprimirse. Una vez descomprimido, solo hay que entrar en: use-6.0.0 > bin, donde habrá un archivo ejecutable llamado *use* que deberá ejecutarse para usar esta herramienta.

A.6 Nuestra herramienta

Nuestra herramienta no consta de instalación, ya que se compone de un fichero .java que deberá ser añadido a el proyecto de Maven en el que se quiera usar. Más información sobre su uso será detallada en el manual de usuario.

Apéndice B

Manual de usuario

En las próximas páginas se explicarán los pasos necesarios para utilizar la herramienta, una vez que se tengan todos los componentes necesarios instalados.

B.1 Creación de proyecto Cucumber vacío

En primer lugar se debe crear un proyecto Cucumber nuevo con la ayuda de Maven. Para ello es necesario abrir la consola del sistema y ubicarla en el directorio en el que se quiera crear el proyecto. Una vez en el directorio requerido, se debe introducir y ejecutar el siguiente comando:

```
mvn archetype:generate "-DarchetypeGroupId=io.cucumber" "-
DarchetypeArtifactId=cucumber-archetype" "-DarchetypeVersion=6.10.4" "-
DgroupId=hellocucumber" "-DartifactId=hellocucumber" "-
Dpackage=hellocucumber" "-Dversion=1.0.0-SNAPSHOT" "-
DinteractiveMode=false"
```

Tras ejecutar dicho comando, deberá aparecer algo similar a la figura 21.

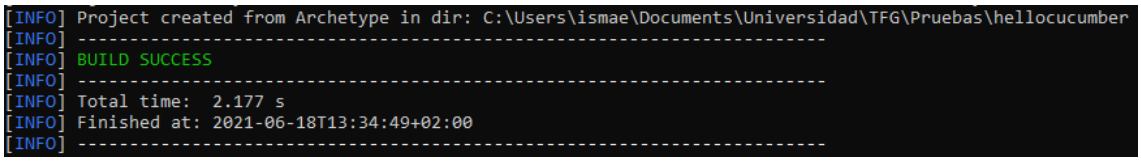

Figura 21. Salido de Maven al crear el proyecto Cucumber

Una vez haya sido creado el proyecto, es recomendable comprobar la integridad del mismo, para ello es necesario ejecutar los siguientes comandos:

1. `cd hellocucumber`
2. `mvn test`

Si todo ha salido como se esperaba, en la consola deberá aparecer algo parecido a la figura 22.

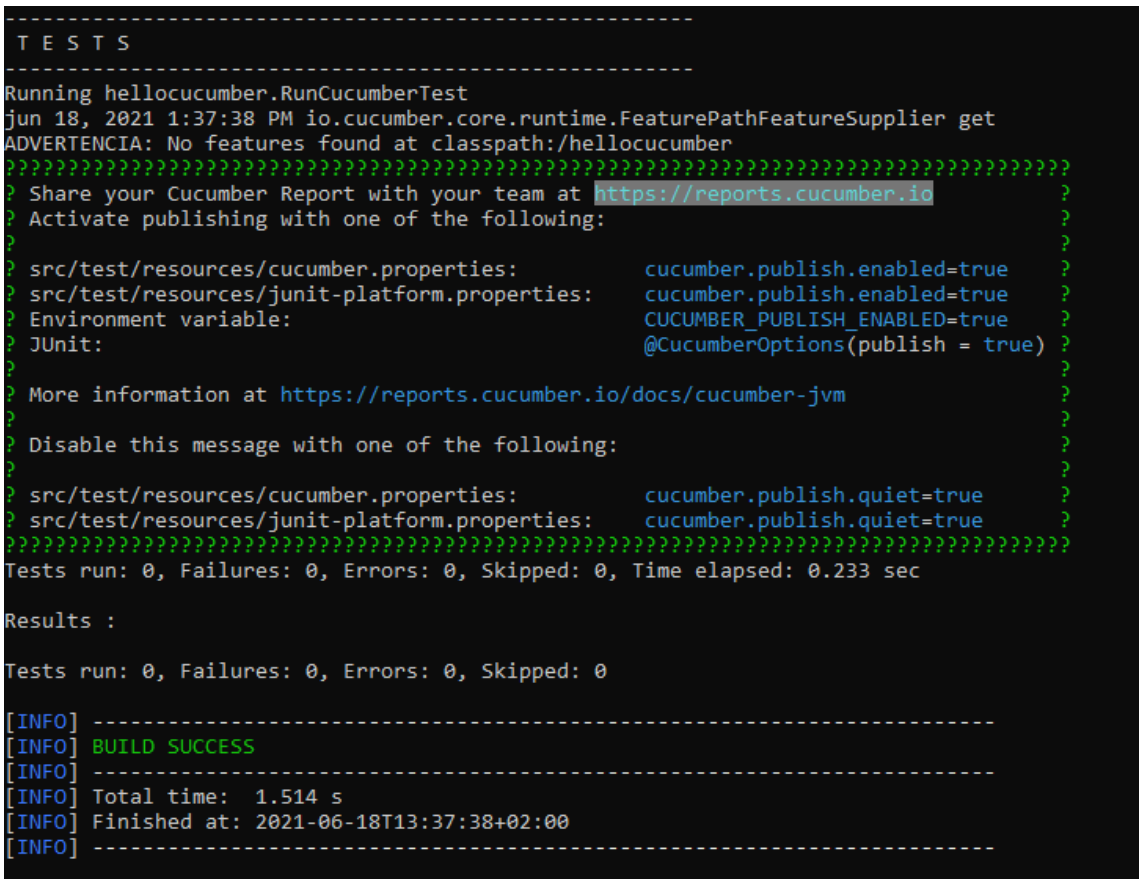

Figura 22. Salida del comando mvn test

Algo a destacar sobre este paso es que en el comando para crear el proyecto, se puede cambiar *hellocucumber* por otro nombre que se desee, aunque esto hará que sea necesario modificar una parte del fichero Java que compone la herramienta, que será mencionado posteriormente.

B.2 Abrir proyecto en IntelliJ IDEA

Una vez haya sido creado el proyecto vacío de Cucumber se debe entrar en el IDE IntelliJ IDEA. Aquí se deberá seleccionar File > Open, y deberá buscarse la carpeta que acaba de ser creada y, dentro de ella, seleccionar el archivo llamado *pom.xml*. Tras esto, el proyecto debería abrirse sin ningún problema.

B.3 Especificar los escenarios

Una vez se tenga el proyecto creado, se deberá buscar el directorio src > test > resources > hellocucumber (*hellocucumber* puede variar en función del nombre que se haya especificado a la hora de crear el proyecto). En este directorio se debe crear un fichero con extensión *feature* y con nombre a elegir.

Una vez se haya creado este fichero, se deben especificar las pruebas como se muestra en la figura 8, siendo necesario que el primer y último escenario sea de tipo *Scenario* y no de *Scenario Outline*, como se explica en la sección 4.1. Además, también es necesario que sobre el primer escenario aparezca el *tag @start*, al igual que en el último debe aparecer *@end*, como puede volver a apreciarse en la figura 8. Esto es necesario para garantizar el correcto funcionamiento de la herramienta.

B.4 Creación de fichero StepDefinitions

Una vez se hayan especificado todos los escenarios, se deben especificar dichos pasos en el fichero llamado *StepDefinition.java*, que debe encontrarse en el directorio src > test > java > hellocucumber (al igual que anteriormente, el nombre *hellocucumber* puede variar).

Para generar estos pasos, se debe seleccionar nuestro fichero *feature*, y con el botón derecho pinchar sobre él, lo que hace que aparezca una pequeña ventana, y se deberá pinchar en *Run: 'Feature nombre'*, donde el *nombre* es el que se haya escrito al crear el fichero. Tras esto, Cucumber muestra por consola un mensaje que indica que se han saltado los pasos ya que no están especificados. Debajo de todo el texto aparecerá también, en blanco, el código que se puede usar para implementar dichos pasos. Este se puede copiar y pegar en el fichero *StepDefinitions.*

Otra forma de llevar a cabo el paso anterior es ejecutando por consola y en el directorio del proyecto, el comando `mvn test`, lo que debe mostrar lo mismo que se menciona anteriormente.

Este código dado por la herramienta lanza una excepción al no estar implementado, para validar que no falta ninguno es necesario borrar esa parte y volver a ejecutar el fichero *feature*. Tras eliminar las excepciones de este fichero, deberá quedar algo similar a la figura 9. Si todo ha salido bien, deberá aparecer algo similar a la figura 23.

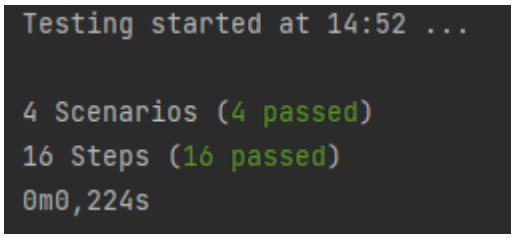

Figura 24. Salida de ejecutar escenarios implementados

En este paso también es posible crear un fichero Java distinto a *StepDefinitions*, siempre que esté en la misma carpeta que este fichero, en el cual también es posible especificar los escenarios requeridos.

B.5 Creación de fichero de texto con nuestra herramienta

Una vez se tenga el fichero Java con los pasos definidos, se puede pasar al siguiente paso, en este caso será crear el fichero de texto a partir del fichero de Java anteriormente implementado.

Para esto será necesario añadir el fichero Java que compone esta herramienta al proyecto, para ello es recomendable crear una carpeta nueva dentro de la carpeta llamada *java*, creada automáticamente por Maven al crear el proyecto, y en la cual se encuentra el fichero *StepDefinitions*, dentro , a su vez, de otra carpeta llamada igual que el nombre dado al proyecto anteriormente. A esta nueva carpeta se le puede nombrar como se quiera, sin ninguna restricción. Dentro de esta carpeta se debe añadir el fichero Java llamado *soilToJava*. Para ejecutar este fichero es necesario seleccionarlo y pinchar con el botón derecho y volver a pinchar en la opción *Run: 'soilToJava.main()'*. Al ejecutar esto debe aparece por la consola del IDE lo mismo que en la figura 10, donde se indican todos los ficheros relacionados con las pruebas. En este paso aún no es necesario indicar ficheros USE y SOIL, por lo cual, si aún no se han añadido estos archivos a la herramienta, no habría ningún inconveniente, se podría escribir cualquier línea de texto en su lugar e igualmente funcionaría sin

problemas. Para ejecutar este paso solo es necesario escribir *1* y darle a intro, una vez que se ha ejecutado sin problemas, escribimos *3* para salir.

Ahora se ha creado un fichero de texto llamado como se haya especificado, en el cual se deben especificar las pruebas en el lenguaje usado por USE. Una vez se hayan especificado dichas pruebas, debe parecer similar a la figura 11.

En este paso es necesario respetar los saltos de línea y que las especificaciones de USE estén entre las llaves, pero nunca en la misma línea donde empieza o termina cada función relacionada con los distintos pasos, ya que esto podría hacer que la herramienta no funcione correctamente. Además, en el caso que para las pruebas se usen parámetros dinámicos pasados por Gherkin, será necesario especificarlo en las pruebas de igual manera que se muestra en la figura 24, en este caso con la variable *int1*.

```
@Then("the speed is equal to {int}")
public void the_speed_is_equal_to(Integer int1) {
   pw.println("result := (c.speed = "+int1+");");
}
```

Figura 24. Ejemplo de uso de variables dinámicas de Gherkin

B.6 Actualización de fichero StepDefinitions

Como anteriormente se dijo, no es necesario que este fichero sea llamado *StepDefinitions*, pero como Maven lo crea de forma automática también se ha usado en este manual de usuario.

Para realizar este paso se deberá de haber completado los anteriores de forma correcta. Una vez que se tenga el fichero de texto completo, o en cualquier momento anterior, se deben incluir los ficheros USE y SOIL a la raíz del proyecto. Esto es posible creando dos nuevos ficheros desde IntelliJ IDEA o arrastrándolos, si es que ya existen. Una vez se tengan estos ficheros incluidos, se debe ejecutar de nuevo el fichero *soilToJava*, pero esta vez rellenando también los ficheros USE y SOIL, como aparece en la figura 10.

Tras la ejecución, se actualiza el fichero Java que contiene la especificación de los escenarios, de forma que debe contener todo lo necesario para pasar las pruebas a los ficheros USE y SOIL. Para esto sólo es necesario volver a ejecutar el fichero *feature*. Tras esto, los ficheros USE y SOIL se actualizan con el código especificado.

B.7 Validación de pruebas en USE

Una vez se hayan completado todos los pasos anteriores, y se haya desarrollado el modelo a validar completamente, se pueden validar las pruebas implementadas. Para ello es necesario abrir USE y buscar el fichero utilizado. Al abrir este fichero, la herramienta debe aparecer de forma similar a la figura 25.

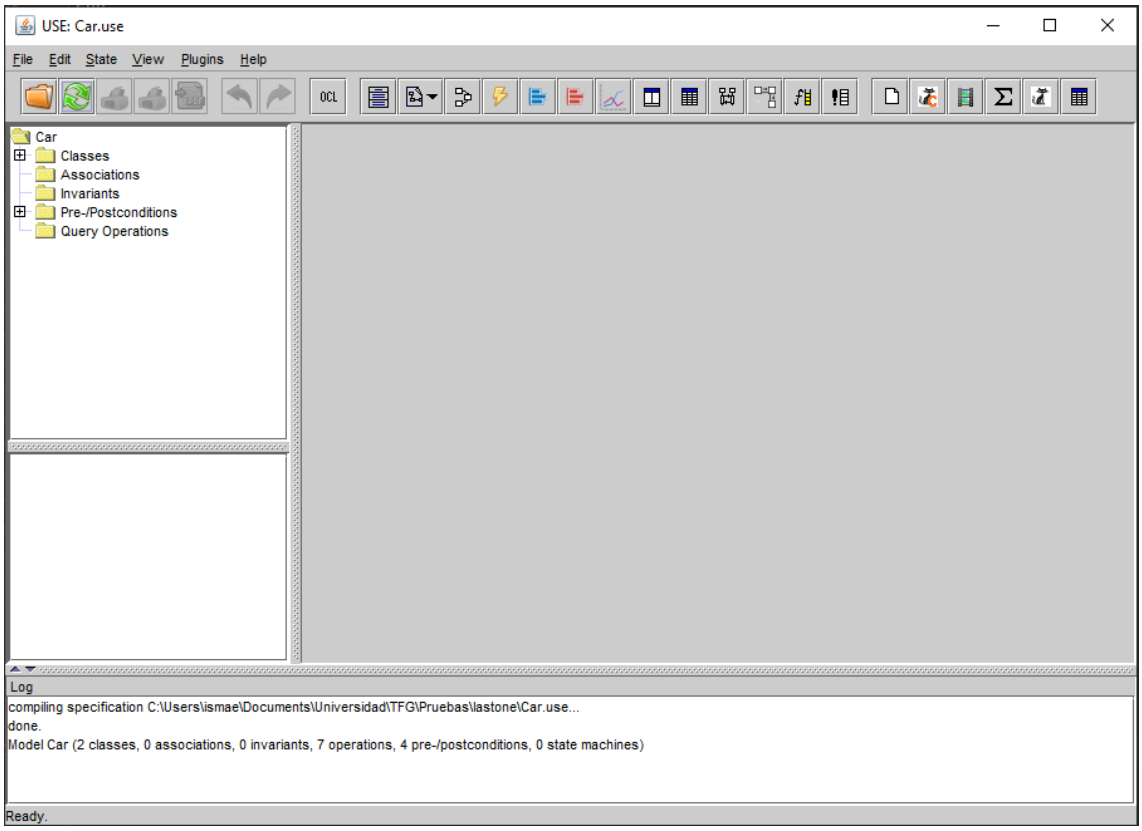

Figura 25. Herramienta USE tras cargar un modelo

Una vez se haya cargado el modelo correctamente en USE, es necesario introducir el comando `open nombre.soil`, donde *nombre* hace referencia al nombre previamente seleccionado para llamar a dicho archivo. Tras introducir este comando, y en caso de que todas las pruebas sean validadas, debe aparecer por consola algo similar a la figura 16.

B.8 Información adicional

Los pasos anteriores de este manual de usuario están explicados partiendo de la base de que no se deseen hacer modificaciones posteriores al primer uso y que esta implementación haya sido completada de forma exitosa, por lo que en esta sección se explica qué se debe hacer antes distintas adversidades que se pueden encontrar tras haber validado las pruebas.

B.8.1 Añadir pruebas nuevas

Si se desean añadir pruebas nuevas una vez que se han validado pruebas anteriores usando esta herramienta, la mejor opción es introducir dichos escenarios en el fichero *feature* y ejecutarlo. Al ejecutarlo debe mostrar un error, ya que dichas pruebas no se han especificado en el fichero de Cucumber y tras ese error deben aparecer las funciones java que podemos utilizar, al igual que se explica en el paso B.4. Una vez se obtengan estas funciones, es necesario pegarlas en el fichero de texto usado para especificar las pruebas en lenguaje de USE. Una vez que estas pruebas estén especificadas, se debe volver a realizar el paso B.6.

Una vez se haya llevado a cabo lo mencionado en el párrafo anterior, otro aspecto a tener en cuenta está relacionado con los ficheros USE y SOIL, ya que estos tendrían las pruebas anteriores, y volver a ejecutar el fichero *feature* sólo sobrescribiría de nuevo dichas pruebas, por lo que es necesario borrar el contenido del fichero SOIL completamente, igual que borrar el contenido del fichero USE a partir de la línea *class Test*, incluyendo la misma.

Tras haber realizado estos pasos, se puede volver a ejecutar el fichero *feature*, lo cual añadirá estas pruebas a los fichero USE y SOIL, añadiendo también las nuevas pruebas implementadas.

B.8.2 Alguna prueba está mal implementada

En el caso de que se descubra que alguna prueba está mal implementada y que por lo tanto da error al validar, o que no se cargue el modelo en la herramienta USE de forma correcta, lo mejor es volver a especificar dicha prueba en el fichero de texto y

volver a realizar el paso B.6. Tras esto, también se debe eliminar el contenido del fichero SOIL y el del fichero USE a partir de la línea *class Test,* incluyéndola.

B.8.3 Usar directorios diferentes a los de prueba

En el caso de que se cree el proyecto de Cucumber con otros parámetros que no sean los del ejemplo, en este caso *hellocucumber*, lo único que se debe hacer es cambiar en el fichero *soilToJava* el directorio de estos archivos. En la figura 26 se muestra la forma actual de dicho archivo, en el cual solo se debe modificar el directorio en el que se encuentran los distintos archivos necesarios para el uso de la herramienta.

```
soilToJava stj = new soilToJava( ficheroFeature: "src/test/resources/hellocucumber/"+feature,
        ficheroJava: "src/test/java/hellocucumber/"+java,
        ficherotxt: "src/test/java/hellocucumber/"+texto,
       use,
       soil);
```

Figura 26. Directorios actuales de los archivos

Adicionalmente, y en el caso en el que se desee usar otro directorio que no sea el raíz del proyecto para incluir los ficheros USE y SOIL, se puede añadir en formato cadena de texto el directorio en el que se deseen guardar, como ejemplo se tiene la figura 27, donde se guardan en el directorio src > test > files.

```
soilToJava stj = new soilToJava( ficheroFeature: "src/test/resources/hellocucumber/"+feature,
        ficheroJava: "src/test/java/hellocucumber/"+java,
        ficherotxt: "src/test/java/hellocucumber/"+texto,
        ficheroUSE: "src/test/files/"+use,
        ficheroSOIL: "src/test/files/"+soil);
```

Figura 27. Ejemplo cambio de directorio de ficheros USE y SOIL

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA