MÁSTER UNIVERSITARIO DE INVESTIGACIÓN EN INGENIERÍA DE SOFTWARE Y SISTEMAS INFORMÁTICOS

Trabajo Fin de Máster

Itinerario Ingeniería de Software (Código: 105151)

Definición de un lenguaje visual específico de dominio para el modelado y la generación automática del código de automatización de las pruebas funcionales de las aplicaciones

Autor: Safwat Alchihabi Bawadekji

Director: Eugenio Arellano Alameda

Curso Académico 2017/2018

Convocatoria Septiembre

MÁSTER UNIVERSITARIO DE INVESTIGACIÓN EN INGENIERÍA DE SOFTWARE Y SISTEMAS INFORMÁTICOS

ITINERARIO DEL TRABAJO: Ingeniería de Software

CÓDIGO DE LA ASIGNATURA: 105151

TÍTULO DEL TRABAJO: Definición de un lenguaje visual específico de dominio para el modelado y la generación automática del código de automatización de las pruebas funcionales de las aplicaciones

TIPO DE TRABAJO: Tipo B, trabajo específico propuesto por el alumno

Autor: Safwat Alchihabi Bawadekji

Director: Eugenio Arellano Alameda

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MASTER

Fecha: 28/08/2018

Quién suscribe:

Autor: Safwat Alchihabi Bawadekji D.N.I: 51710105X

Hace constar que es el autor del trabajo:

Definición de un lenguaje visual específico de dominio para el modelado y la generación automática del código de automatización de las pruebas funcionales de las aplicaciones

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo. Safwat

Impreso TFdM05_Autor. Autorización de publicación y difusión del TFdM para fines académicos

Autorización

Autorizo/amos a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del/los Autor/es

Juan del Rosal, 16
28040, *Madrid*

Tel: 91 398 89 10
Fax: 91 398 89 09

www.issi.uned.es

Resumen

En este trabajo se define un **lenguaje visual específico de dominio (DSL)** para modelar las pruebas funcionales de las aplicaciones. El lenguaje de modelado definido permite representar el funcionamiento de la aplicación tanto a alto nivel como a bajo nivel.

Se ha creado una **herramienta de modelado gráfico** que implementa el DSL visual definido. También **se** ha desarrollado un **generador automático de código** de pruebas funcionales de aplicaciones web que utiliza como entrada el modelado creado con el DSL.

En la presente memoria del trabajo se incluye un ejemplo que tiene como objetivo ilustrar el potencial del trabajo realizado.

Este trabajo se puede utilizar como base para crear herramientas de gestión y automatización de pruebas.

Palabras clave: DSL, Modelado, Automatización, Generación Automática de Código, Pruebas Funcionales

Índice

1. OBJETIVOS Y LIMITACIONES 11

1.1. Objetivos 11
1.2. Limitaciones 11

2. INTRODUCCIÓN 12

2.1. Introducción a las pruebas funcionales 12
2.2. Introducción a la automatización de las pruebas funcionales 13
2.2.1. Beneficios de la automatización de las pruebas funcionales 13
2.2.2. Criterios para automatizar las pruebas funcionales 14

3. SITUACIÓN ACTUAL 15

3.1. Análisis de herramientas de gestión y automatización de las pruebas funcionales 15
3.1.1. Herramientas de gestión de las pruebas funcionales 15
3.1.2. Herramientas de automatización de las pruebas funcionales 16
3.2. Conclusiones del análisis 19

4. SOLUCIÓN PROPUESTA DE MODELADO GRÁFICO DE LAS PRUEBAS FUNCIONALES 20

4.1. Descripción general 20
4.2. Descripción del lenguaje de modelado gráfico propuesto 20
4.2.1. Objetos de los casos de prueba y los pasos de prueba 21
4.2.2. Objetos de la GUI 23
4.2.3. Relación entre los objetos del modelado 24
4.3. Representación del modelado visual 25
4.3.1. Representación del modelado en grafo 26
4.3.2. Representación del modelado en XML 28
4.4. Ejemplo de modelado gráfico 29
4.5. Implementación de la solución en la metodología de desarrollo 31
4.5.1. Descripción general 31
4.5.2. Implementación de la solución en el ciclo de vida de desarrollo del Software 32
4.5.2.1. Fase de Análisis 32
4.5.2.2. Fase de Diseño 32
4.5.2.3. Fase de Construcción 33
4.5.2.4. Fase de Pruebas 33
4.6. Ventajas de la solución propuesta 33

5. DISEÑO Y CREACIÓN DE UNA HERRAMIENTA DE MODELADO GRÁFICO DE LAS PRUEBAS FUNCIONALES 35

5.1. Instalación del Software de modelado 35
5.2. Creación del metamodelo del DSL visual 36
5.3. Creación del proyecto de diseño de la herramienta de modelado 40
5.4. Creación de proyectos que implementan la herramienta de modelado 46
5.5. Uso de la herramienta de modelado creada 47
5.5.1. Creación del modelado arrastrando y soltando los objetos del DSL visual 47
5.5.2. Creación del modelado gráficamente en forma de árbol 50

6. CREACIÓN DE UN GENERADOR DE CÓDIGO DE AUTOMATIZACIÓN DE LAS PRUEBAS FUNCIONALES 52

6.1. Funcionamiento del generador de código 52

6.2. ARQUITECTURA DEL SOFTWARE DEL GENERADOR DE CÓDIGO 53
6.2.1. Código de la GUI (Vista de MVC) *54*
6.2.2. Código de las acciones del usuario (Controlador de MVC) *55*
6.2.3. Código del modelo de negocio (Modelo de MVC) *57*
6.2.4. Código de la clase principal del generador de código *66*
6.2.5. Código de la clase de utilidades *66*

7. EJEMPLO DE MODELADO Y AUTOMATIZACIÓN DE LAS PRUEBAS FUNCIONALES 72

7.1. FUNCIONAMIENTO DE LA APLICACIÓN WEB DE EJEMPLO 72
7.2. EJEMPLO DE MODELADO GRÁFICO DE UNA PRUEBA FUNCIONAL 73
7.3. EJEMPLO DE UNA GENERACIÓN AUTOMÁTICA DE CÓDIGO DE PRUEBAS 75
7.4. CREACIÓN DEL SCRIPT DE AUTOMATIZACIÓN DE LAS PRUEBAS FUNCIONALES 79

8. CONCLUSIONES 82

9. TRABAJOS FUTUROS 83

10. REFERENCIAS 84

11. TÉRMINOS 85

Lista de Figuras

Ilustración 1 - Las pruebas funcionales en el ciclo de vida del Software 13
Ilustración 2 - Página principal de TestLink 15
Ilustración 3 - Modelado gráfico de las pruebas funcionales 21
Ilustración 4 - Grafo dirigido 26
Ilustración 5 - Grafo del modelado de las pruebas funcionales 27
Ilustración 6 - Ejemplo (1) del flujo de los pasos de los casos de prueba de una aplicación 30
Ilustración 7 - Ejemplo (1) de modelado gráfico de las pruebas funcionales 30
Ilustración 8 - Implementación de la solución de modelado en el ciclo de vida de desarrollo del Software 32
Ilustración 9 - Paquete del Software de modelado de Eclipse 35
Ilustración 10 - Instalación del proyecto Sirius de Eclipse 36
Ilustración 11 - Metamodelo del DSL visual de las pruebas funcionales 38
Ilustración 12 - Estructura del proyecto del metamodelo 39
Ilustración 13 - Estructura del proyecto del Edit 39
Ilustración 14 - Estructura del proyecto del Editor 40
Ilustración 15 - Estructura del proyecto de diseño de la herramienta de modelado 41
Ilustración 16 - Plug-in que define el metamodelo del DSL Visual 41
Ilustración 17 - Creación del Viewpoint y el diagrama del modelado 42
Ilustración 18 - Ejemplo de las propiedades de un nodo del modelado 42
Ilustración 19 - Ejemplo de las propiedades de un enlace del modelado 43
Ilustración 20 - Definición de los nodos y los enlaces del modelado 43
Ilustración 21 - Definición de las secciones de la herramienta de modelado 44
Ilustración 22 - Ejemplo de las propiedades de un nodo de una sección de la herramienta de modelado 44
Ilustración 23 - Ejemplo de las propiedades de un enlace de una sección de la herramienta de modelado 45
Ilustración 24 - Ejemplo de la configuración de la acción de un nodo de una sección de la herramienta de modelado 45
Ilustración 25 - Ejemplo de la configuración de la acción de un enlace de una sección de la herramienta de modelado 46
Ilustración 26 - Selección del viewpoint del proyecto de modelado 46
Ilustración 27 - Estructura de un proyecto que implementa la herramienta de modelado 47
Ilustración 28 - Herramienta creada para el modelado gráfico de las pruebas funcionales – Ejemplo (1) de modelado 48
Ilustración 29 - Ejemplo (1) - Propiedades de un caso de prueba en la herramienta de modelado .. 49
Ilustración 30 - Ejemplo (1) - Propiedades de un paso de un caso de prueba en la herramienta de modelado 49
Ilustración 31 - Ejemplo (1) - Propiedades de un elemento de la GUI de la aplicación 49
Ilustración 32 - Creación del árbol del modelado de las pruebas funcionales 50
Ilustración 33 – Ejemplo (1) - Las propiedades de un objeto del árbol del modelado 51
Ilustración 34 - Ejemplo (1) - La estructura de árbol del modelado de las pruebas funcionales...... 51
Ilustración 35 - Funcionamiento general del Generador Automático de Código 52
Ilustración 36 - Interfaz gráfica de usuario del Generador Automático de Código 53

Ilustración 37 - Estructura del proyecto del generador automático de código 54
Ilustración 38 - Ejemplo (2) del flujo de los pasos de los casos de prueba de una aplicación web . 72
Ilustración 39 - Ejemplo (2) de modelado gráfico de una prueba funcional de una aplicación 74
Ilustración 40 - GUI del Generador Automático de Código con los datos de entrada completados 76
Ilustración 41 - Estructura del proyecto del script de automatización de las pruebas funcionales .. 80
Ilustración 42 - Exportar el proyecto del script de automatización de la prueba funcional a un fichero JAR ejecutable 80

Lista de Tablas

Tabla 1 - Descripción y propiedades de los objetos de los casos de prueba y los pasos de cada caso de prueba utilizados en el DSL visual 23
Tabla 2 - Descripción y propiedades de algunos objetos de los elementos de la GUI utilizados en el DSL visual 24
Tabla 3 - Los enlaces que representan la relación entre los objetos del modelado de pruebas funcionales 25
Tabla 4 – Ejemplo (1) de una prueba funcional 29
Tabla 5 - Los atributos y las relaciones de las clases del DSL visual 37
Tabla 6 - Ejemplo (2) de una prueba funcional 74

1. Objetivos y Limitaciones

1.1. Objetivos

Los objetivos principales de este trabajo son:

- Analizar las herramientas actuales de gestión y automatización de las pruebas funcionales de las aplicaciones.

- Definir un lenguaje visual específico de dominio para modelar los casos de prueba de las aplicaciones y explicar cómo implementar la solución propuesta de forma óptima en el ciclo de vida de desarrollo del Software.

- Diseño y creación de una herramienta de modelado gráfica utilizando un Software libre para implementar el DSL visual definido.

- Desarrollo de un generador automático de código de automatización de las pruebas funcionales de las aplicaciones web. El modelado gráfico creado con el DSL visual se utilizará como entrada del generador automático de código.

- Creación de un ejemplo de una aplicación web para modelar y para automatizar sus pruebas funcionales utilizando la solución propuesta.

1.2. Limitaciones

Este trabajo tiene las siguientes limitaciones:

- El piloto del generador automático de código, cubre sólo las pruebas funcionales de las aplicaciones basadas en web.

- En la herramienta de modelado y en el generador automático de código, se han incluido sólo algunos elementos de la GUI de las aplicaciones por tratarse de pilotos. El diseño y el desarrollo de los pilotos se han realizado de forma estructurada para facilitar su ampliación.

2. Introducción

Cada vez se exige que la producción del Software sea más rápida y con más calidad. Una de las tareas esenciales en el ciclo de vida de desarrollo del Software, es la realización de las pruebas funcionales para garantizar el correcto funcionamiento del sistema según los requisitos definidos. Por eso, surge la necesidad de encontrar soluciones eficientes para definir, mantener y automatizar las pruebas funcionales de forma óptima.

2.1. Introducción a las pruebas funcionales

El ciclo de vida de desarrollo de las aplicaciones empieza normalmente por la fase de definición de los requerimientos funcionales. Las siguientes fases son la fase de análisis funcional, la fase de diseño técnico y la fase de construcción. Y antes de la instalación final del Software en el entorno de Producción, es muy importante realizar las pruebas de forma manual o automática para verificar la calidad del producto.

Las pruebas del Software se pueden clasificar en dos tipos: pruebas no funcionales y pruebas funcionales.

Las pruebas no funcionales son para evaluar el comportamiento del sistema según los requisitos no funcionales. Ejemplos de este tipo de pruebas pueden ser las pruebas de seguridad, las pruebas de estrés, las pruebas de carga y las pruebas de rendimiento.

Las pruebas funcionales son pruebas de caja negra para verificar que el Software desarrollado cumple los requisitos funcionales definidos por los usuarios. En este tipo de pruebas, los probadores evalúan la respuesta del sistema a los datos de entrada sin preocuparse por su funcionamiento interno. Ejemplos de las pruebas funcionales pueden ser las pruebas de humo, las pruebas de regresión y las pruebas de aceptación.

Una prueba funcional se compone de un conjunto de casos de prueba que cubren los requisitos funcionales. Y un caso de prueba es un conjunto de pasos (o acciones) que se realizan para verificar una funcionalidad o una parte de una funcionalidad del sistema.

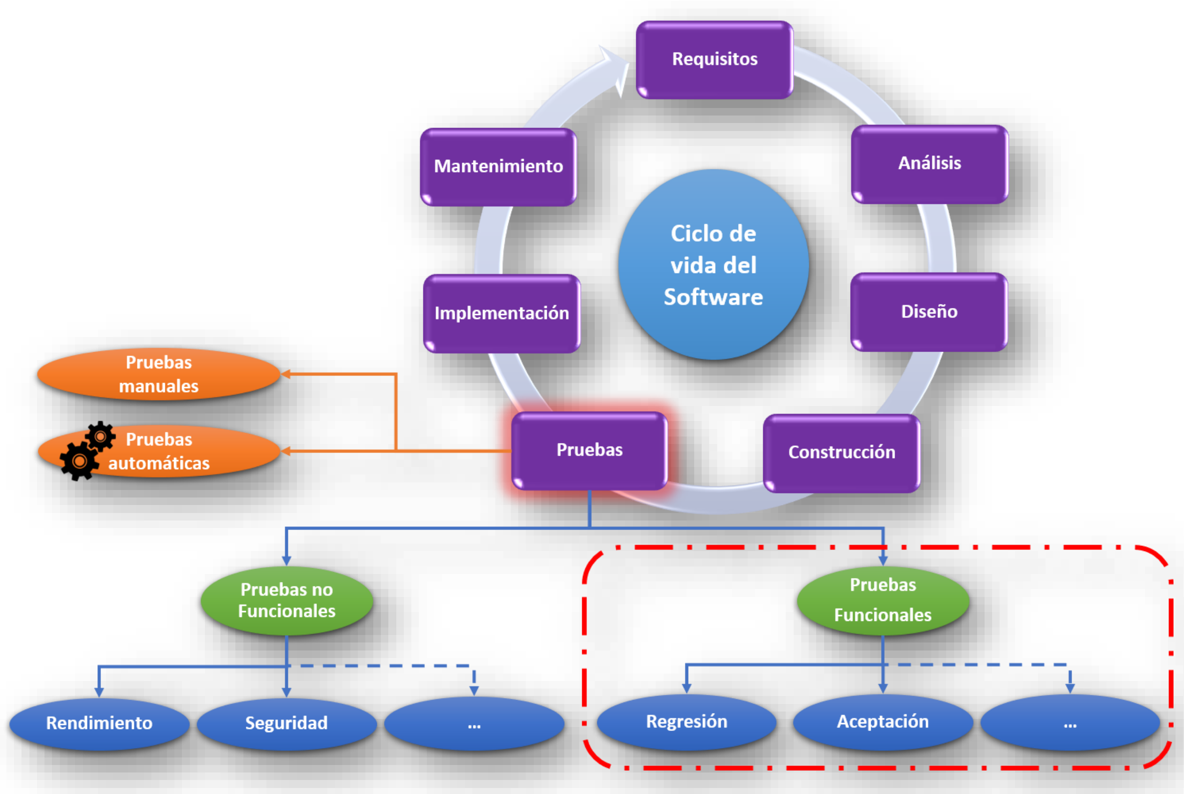

Ilustración 1 - Las pruebas funcionales en el ciclo de vida del Software

2.2. Introducción a la automatización de las pruebas funcionales

Es habitual la automatización de las pruebas funcionales repetitivas en los desarrollos de aplicaciones complejas o críticas para garantizar la calidad del Software. La automatización de las pruebas funcionales tiene grandes beneficios en los desarrollos ágiles como Scrum donde los ciclos de desarrollo (*Sprints*) son cortos.

Las pruebas funcionales que más se automatizan son las pruebas de regresión porque se ejecutan de forma frecuente. Las pruebas de regresión son para verificar que las modificaciones realizadas en el Software no han causado nuevos defectos en las funcionalidades desarrolladas y probadas previamente.

2.2.1. Beneficios de la automatización de las pruebas funcionales

Con la automatización de las pruebas funcionales se obtienen los siguientes beneficios:

- Acabar con las tareas repetitivas de los desarrolladores y los probadores (*testers*) para dedicarse a funciones que aportan más valor.
- Mejorar la calidad del Software.
- Reducción del tiempo de realización de las pruebas. Por lo tanto, se reduce el tiempo de producción del Software.
- Las pruebas funcionales automáticas simulan las acciones del usuario de forma precisa minimizando los errores humanos que se pueden producir realizando las mismas pruebas manualmente de forma repetitiva.
- La ejecución de las pruebas funcionales se puede programar para que se realice en cualquier momento.
- Se puede aumentar la cobertura funcional de las pruebas porque se reducen las limitaciones de recursos y de tiempo.

2.2.2. Criterios para automatizar las pruebas funcionales

A continuación, se detallarán algunos de los criterios que se deben considerar a la hora de tomar la decisión de automatizar los casos de prueba del Software desarrollado:

- **Reusabilidad**. Hay que ver si los casos de prueba candidatos a automatizar, van a tener cambios a corto o a medio plazo. Además, hay que prever la frecuencia de ejecución de esos casos de prueba por las actualizaciones del Software.
- **Criticidad**. Se debe priorizar la automatización de los casos prueba según su criticidad.
- **Tipo de prueba**. Hay algunos tipos de pruebas que son difíciles de automatizar porque necesitan la opinión del probador (*tester*) como las pruebas exploratorias o las pruebas relacionadas con la valoración de los aspectos visuales de la GUI de la aplicación.
- **Casos de prueba no automatizables**. Hay casos de prueba que no se pueden automatizar porque dependen de la interacción con algún hardware (escáner, impresora, sensor, lector de tarjetas, etc.). En estos casos hay que evaluar si es rentable el desarrollo de simuladores de las acciones de los usuarios con el hardware.

3. Situación actual

Se han analizado algunas de las herramientas más conocidas que se utilizan para gestionar y para automatizar las pruebas funcionales. El objetivo de este análisis es ver las necesidades de mejoras en estas herramientas.

3.1. Análisis de herramientas de gestión y automatización de las pruebas funcionales

3.1.1. Herramientas de gestión de las pruebas funcionales

Se han analizado las herramientas TestLink y Quality Center que proporcionan una gestión completa de las pruebas funcionales de las aplicaciones.

TestLink es una herramienta libre y de código abierto basada en web que se utiliza para gestionar los casos de prueba y tiene soporte basado en los foros de Internet. TestLink soporta las bases de datos MySQL, MariaDB y PostgreSQL.

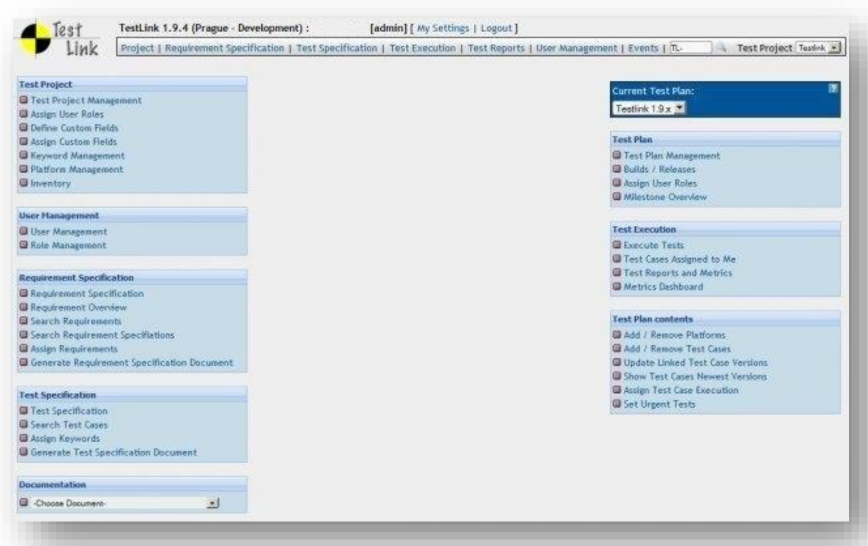

Ilustración 2 - Página principal de TestLink

Quality Center es una herramienta comercial basada en web. Quality Center se integra con la herramienta UFT de automatización de las pruebas.

Funcionamiento de las herramientas de gestión de pruebas

Con las herramientas analizadas de gestión de los casos de prueba, se permite realizar lo siguiente:

- Gestión de los requisitos funcionales.
- Creación de los casos de prueba relacionados con los requisitos funcionales.
- Agrupación de los casos de prueba en planes de pruebas. Por ejemplo, se pueden crear planes de pruebas para definir las pruebas de regresión.
- Creación de ciclos de ejecución de los casos de prueba.
- Ejecución de los casos de prueba. La ejecución de los casos de prueba puede ser manual por los probadores o automática a través de una herramienta de automatización de pruebas.
- Se registra en base de datos el resultado de ejecución de los casos de prueba de cada ciclo de prueba.
- Se puede obtener informes y métricas de los resultados de ejecución de las pruebas.
- Se integran con herramientas de gestión de defectos como Jira o Mantis.
- Gestión de perfiles y roles de los usuarios que utilizan la herramienta.
- Integración con LDAP (*Lightweight Directory Access Protocol*) para autenticar a los usuarios que acceden a la herramienta.

3.1.2. Herramientas de automatización de las pruebas funcionales

Las herramientas de automatización de las pruebas funcionales se pueden clasificar en dos tipos:

- **Herramientas basadas en la programación textual.** Con estas herramientas se describen las pruebas de usuario mediante un lenguaje de programación textual como Java. Selenium es en ejemplo de estas herramientas.
- **Herramientas basadas en la grabación de las acciones del usuario.** Con estas herramientas se graban, se editan y se reproducen las acciones realizadas por el usuario. Estas herramientas suelen soportar también la programación textual. Selenium IDE, Eggplant Functional y UFT son ejemplos de este tipo de herramientas.

A continuación, se analizarán algunas herramientas comerciales y de Software libre que se utilizan para automatizar las pruebas funcionales.

Selenium

Selenium es un entorno libre y de código abierto que se utiliza para automatizar las pruebas de las aplicaciones web. Selenium utiliza un lenguaje específico textual para escribir el código de automatización de las pruebas en varios lenguajes de programación.

Características técnicas

- Lenguajes de programación utilizados por la herramienta: C#, Haskell, JavaScript, Java, Objective-C, Perl, PHP, Python y Ruby.
- Navegadores soportados: Firefox, Internet Explorer, Safari, Opera y Chrome.
- Tecnologías soportadas: Soporta sólo aplicaciones basadas en web.
- Sistemas operativos soportados: Microsoft Windows, Apple OS X y Linux.
- Tiene soporte basado en los foros de Internet de la comunidad de Selenium.

Selenium IDE

Selenium IDE es un entorno de desarrollo libre integrado para los scripts de Selenium que permite grabar, editar, depurar y reproducir las acciones del usuario. También se puede utilizar para generar el código de automatización de las pruebas basado en el lenguaje específico de Selenium.

Características técnicas

- Navegadores soportados: Se puede utilizar sólo con el navegador Firefox.
- Tecnologías soportadas: Soporta sólo aplicaciones basadas en web.
- Almacena las pruebas en varios formatos como HTML o Ruby Scripts para su posterior importación y ejecución.
- Autocompleta los comandos de Selenium más comunes.
- Tiene soporte basado en los foros de Internet de la comunidad de Selenium.

UFT (Unified Functional Testing)

UFT (Unified Functional Testing) es una herramienta comercial que se utiliza para automatizar las pruebas de las aplicaciones. UFT proporciona una interfaz gráfica de usuario para grabar, editar, depurar y reproducir las acciones del usuario. El funcionamiento de la herramienta está basado en el descubrimiento y el almacenamiento de los elementos de la GUI de las aplicaciones como objetos o como imágenes.

Características técnicas

- Lenguajes de programación utilizados por la herramienta: La herramienta utiliza el lenguaje VBScript para programar los scripts de las pruebas.
- La herramienta se puede instalar sólo en el sistema operativo Microsoft Windows utilizando servidores físicos o máquinas virtuales.
- Herramientas de control de versiones integradas: Subversion y GIT.
- Navegadores soportados: Firefox, Internet Explorer, Microsoft Edge y Chrome.
- Tecnologías y plataformas soportadas: Web, Windows, Java, .NET, WPF (*Windows Presentation Foundation*), Delphi, SAP, PeopleSoft, Siebel, Power Builder, Flex, Stingray, Visual Basic, VisualAge, ActiveX, Terminal Emulator, API, Web Services, y plataformas móviles (Android, iOS, Windows).
- Bases de datos soportadas: Oracle, Microsoft SQL Server y MySQL.
- Informes: La herramienta genera informes del resultado de ejecución de las pruebas.

Eggplant Functional

Eggplant Funcional es una herramienta comercial que se utiliza para automatizar las pruebas funcionales de las aplicaciones. La herramienta proporciona una interfaz gráfica de usuario para grabar, editar y reproducir las acciones del usuario. El funcionamiento de la herramienta está basado en el reconocimiento y el almacenamiento de los elementos de la GUI de las aplicaciones como imágenes.

Características técnicas

- La herramienta utiliza el lenguaje SenseTalk para programar los scripts de las pruebas. También se puede utilizar los lenguajes Java, C# y Ruby a través de la interfaz de eggDrive.

- La herramienta se puede instalar en los sistemas operativos Microsoft Windows, MAC OS X y Linux.
- Navegadores soportados: Firefox, Internet Explorer, Safari, Opera y Chrome.
- Tecnologías y plataformas soportadas: Web, Java, .NET, Windows, Linux, OS X y plataformas móviles (Android, Blackberry, iOS, QNX, WinCE).

3.2. Conclusiones del análisis

Del análisis de las herramientas de gestión y automatización de las pruebas funcionales, se han obtenido las siguientes conclusiones:

- Las herramientas de gestión de pruebas funcionales tienen los siguientes puntos a mejorar:
  - No contienen información sobre los flujos de los casos de prueba ni sobre los flujos de los pasos de cada caso de prueba.
  - Es difícil el mantenimiento de los casos de prueba cuando hay modificaciones en el Software.
  - No proporcionan una forma directa y sencilla para generar el código de automatización a partir de la información de los casos de prueba.

- Las herramientas de automatización de las pruebas funcionales tienen los siguientes puntos a mejorar:
  - Las herramientas basadas en la programación textual como Selenium, necesitan un perfil técnico alto y mucha dedicación para la programación de las pruebas. Además, con este tipo de herramientas, los casos de prueba son difíciles de mantener.
  - No proporcionan la posibilidad de automatizar las pruebas funcionales de forma gráfica sin grabar las acciones del usuario.
  - Algunas herramientas no soportan todos los navegadores, como Selenium IDE que es compatible sólo con el navegador Firefox.

4. Solución propuesta de modelado gráfico de las pruebas funcionales

4.1. Descripción general

Con el objetivo de mejorar los puntos débiles encontrados en las herramientas analizadas de gestión y automatización de las pruebas, se propone la creación de un lenguaje visual específico de dominio para modelar las pruebas funcionales de las aplicaciones que se definen y se ejecutan en el ciclo de vida de desarrollo del software. El modelado creado con este lenguaje describe gráficamente la información de las pruebas funcionales.

La solución propuesta se puede utilizar como base para crear herramientas de gestión de pruebas y para crear generadores de código de automatización de pruebas basados en el modelado de las pruebas funcionales.

4.2. Descripción del lenguaje de modelado gráfico propuesto

Se ha definido un DSL visual para modelar las pruebas funcionales de las aplicaciones. El modelado visual utilizado es de tipo DSM (Modelado Específico de Dominio) que representa gráficamente los objetos que constituyen las pruebas funcionales y la relación entre ellos.

El modelado creado con el DSL visual contiene detalles del funcionamiento de las aplicaciones tanto a alto nivel como a bajo nivel.

En el lenguaje de modelado gráfico propuesto se utilizan varios tipos de objetos gráficos que representan:

- Los casos de prueba.
- Los pasos de cada caso de prueba.
- Los elementos de la GUI de la aplicación.
- La relación entre los objetos del modelado.

Se han utilizado colores distintos en cada tipo de objetos para mejorar la usabilidad del DSL visual.

En el siguiente esquema se muestra el modelado gráfico propuesto de las pruebas funcionales:

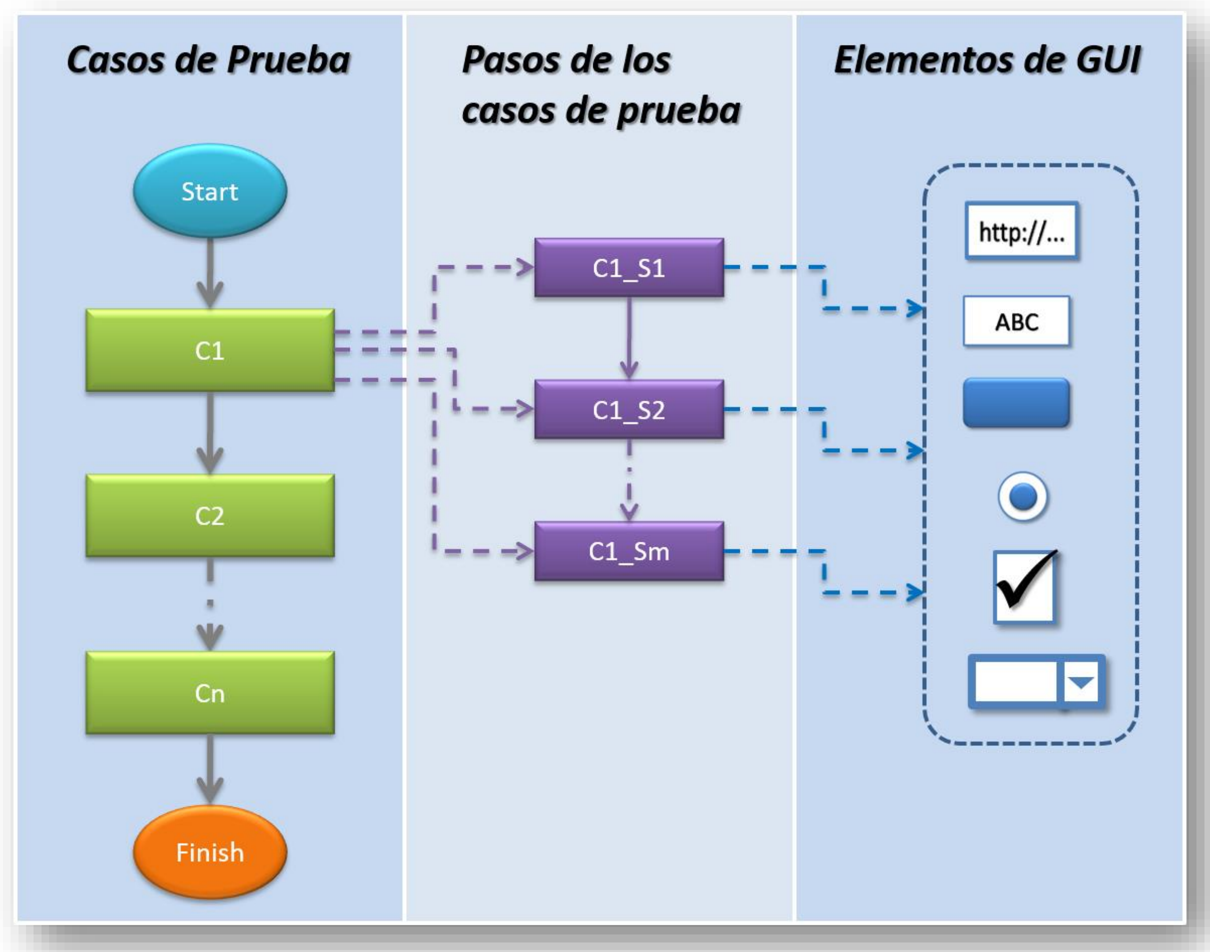

Ilustración 3 - Modelado gráfico de las pruebas funcionales

A continuación, se detallan las propiedades de cada tipo de los objetos que se utilizan para crear el modelado de las pruebas funcionales.

4.2.1. Objetos de los casos de prueba y los pasos de prueba

En la siguiente tabla se muestran la descripción y las propiedades de los objetos de los casos de prueba y los pasos de cada caso de prueba utilizados en el DSL visual:

Objetos del DSL Visual	Descripción	Propiedades
Start	Comienzo del flujo de los casos de prueba	**Name** Por defecto tiene el valor “Start”
Finish	Fin del flujo de los casos de prueba	**Name** Por defecto tiene el valor “Finish”
Cn	Caso de prueba	**Name** Es el código del caso de prueba (Cn) Donde: n es el número del caso de prueba
		Description La descripción del caso de prueba
Cn_Sm	Paso del caso de prueba	**Name** Es el código del paso del caso de prueba (Cn_Sm) Donde: n es el número del caso de prueba m es el número del paso del caso de prueba

		Description La descripción del paso del caso de prueba

Tabla 1 - Descripción y propiedades de los objetos de los casos de prueba y los pasos de cada caso de prueba utilizados en el DSL visual

4.2.2. Objetos de la GUI

En la siguiente tabla se muestran la descripción y las propiedades de algunos objetos de los elementos de la GUI de la aplicación utilizados en el DSL visual:

Objetos de la GUI del DSL Visual	Descripción	Propiedades
http://...	URL	**Name** Nombre del objeto de URL
		Value URL
	Button	**Name** Nombre del botón
ABC	Text Box	**Name** Nombre de la caja de texto
		Value La cadena de caracteres en la caja de texto
	Check Box	**Name** Nombre del CheckBox

		Value Valor que indica el estado del CheckBox. Puede tener dos valores: “True” si el objeto está marcado “False” si el objeto no está marcado
	Radio Button	**Name** Nombre del radio botón
		Value Valor que indica el estado del radio botón. Puede tener dos valores: “True” si el objeto está marcado “False” si el objeto no está marcado
	Combo Box	**Name** Nombre del combo
		Value El valor elegido de la lista del combo

Tabla 2 - Descripción y propiedades de algunos objetos de los elementos de la GUI utilizados en el DSL visual

4.2.3. Relación entre los objetos del modelado

La relación entre los objetos del modelado de las pruebas funcionales se representa por una línea dirigida que puede tener dos formas:

- **Línea dirigida continua** que representa la dirección del flujo de los casos de prueba y la dirección del flujo de los pasos de cada caso de prueba. De esta forma, el modelado contiene información sobre el orden de ejecución de los casos de prueba y el orden de ejecución de los pasos de los casos de prueba.

- **Línea dirigida discontinua** que representa la asignación de los pasos a cada caso de prueba y la asignación de los elementos de la GUI de la aplicación a cada paso del caso de prueba.

En la siguiente tabla se muestran los enlaces que representan la relación entre los objetos del modelado de pruebas funcionales:

Relación entre los objetos del modelado	Descripción
	Dirección del flujo de los casos de prueba
	Dirección del flujo de los pasos de cada caso de prueba
	Asignación de los pasos a cada caso de prueba
	Asignación del elemento de la GUI a cada paso del caso de prueba

Tabla 3 - Los enlaces que representan la relación entre los objetos del modelado de pruebas funcionales

4.3. Representación del modelado visual

El modelado gráfico creado con el DSL visual propuesto se puede representar de forma estructurada en dos formatos:

- Representación del modelado en grafo
- Representación del modelado en formato XML

4.3.1. Representación del modelado en grafo

El modelado gráfico se puede representar como un grafo dirigido. Un grafo (G) se compone de un conjunto nodos (V) unidos por conexiones (E).

G = (V , E)

Un grafo dirigido es un grafo donde las conexiones entre los nodos tienen una orientación que se representa con una flecha.

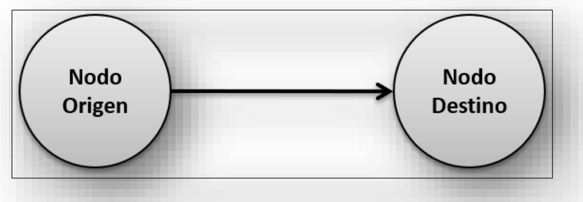

Ilustración 4 - Grafo dirigido

Los nodos del grafo representan:

- Los casos de prueba.
- Los pasos de los casos de prueba.
- Los elementos de la interfaz gráfica de la aplicación asignados a cada paso de un caso de prueba.

Las conexiones del grafo representan:

- La dirección del flujo de los casos de prueba.
- La dirección del flujo de los pasos de casa de prueba.
- La asignación de los pasos a cada caso de prueba.
- La asignación de los elementos de la GUI de la aplicación a cada paso de los casos de prueba.

El siguiente esquema muestra el grafo del modelado de las pruebas funcionales:

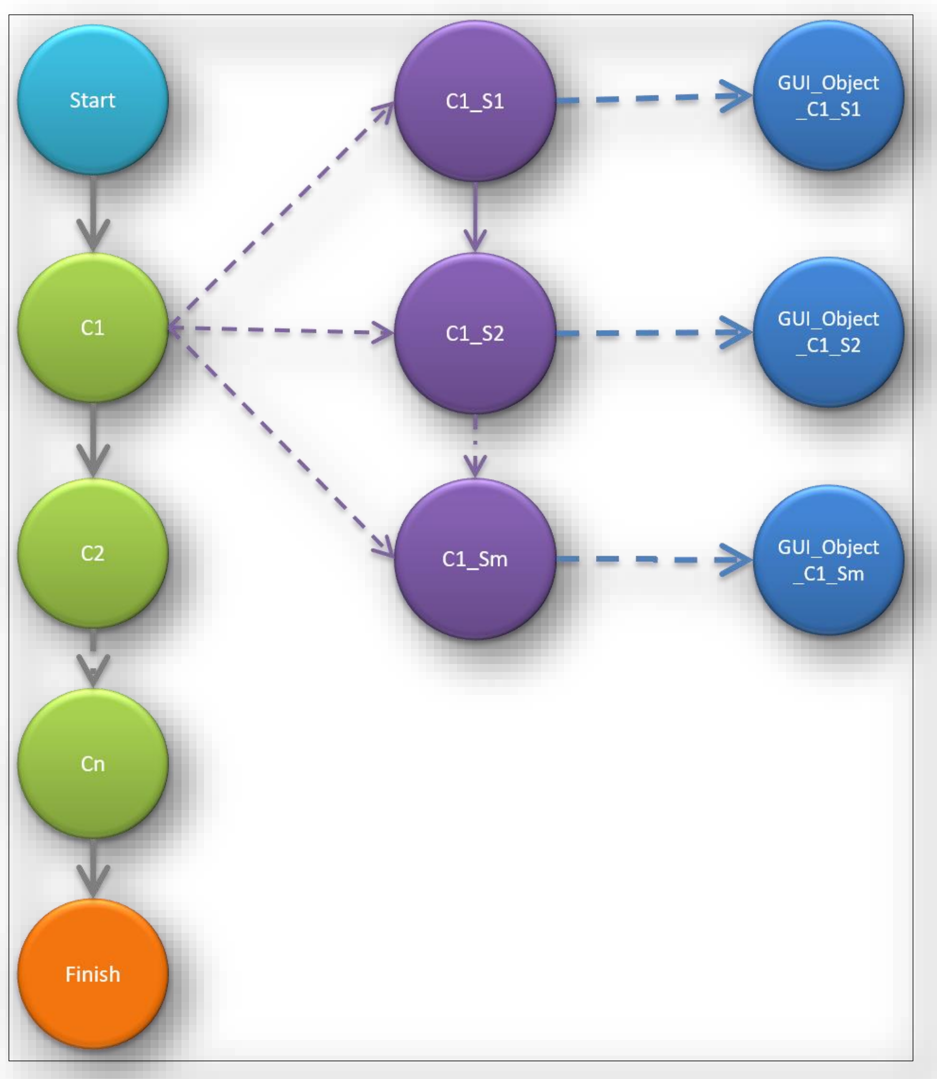

Ilustración 5 - Grafo del modelado de las pruebas funcionales

El grafo contiene:

- Los casos de prueba: C1, C2, …, Cn
- Los pasos de los casos de prueba: C1_S1, C1_S2, …, C1_Sm, C2_S1, C2_S2, …
- Los elementos de la GUI de la aplicación asignados a los pasos de cada caso de prueba: GUI_Object_C1_S1, GUI_Object_C1_S2, …, GUI_Object_C1_Sm, …

Donde:

- n es el número del caso de prueba
- m es el número del paso del caso de prueba

4.3.2. Representación del modelado en XML

El modelado gráfico se puede representar en formato XML que contiene las propiedades (nombre, descripción y valor) de los objetos.

A continuación, se muestra la estructura simplificada del modelado gráfico de las pruebas funcionales en formato XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<Modeling>
<Start/>
        <TestCase name="C1" description="[descripción del caso de prueba]">
                <Step name="C1_S1" description="[descripción del paso]">
                        <[GUI_Object_C1_S1] name=”[Nombre]" value="[Valor]"/>
                </Step>
                <Step name="C1_S2" description="[descripción del paso]">
                        <[ GUI_Object_C1_S2] name=”[Nombre]” value="[Valor]"/>
                </Step>
                ...
                <Step name="C1_Sm" description="[descripción del paso]">
                        <[ GUI_Object_C1_Sm] name=”[Nombre]” value="[Valor]"/>
                </Step>
        </TestCase>

        <TestCase name="C2" description="[descripción del caso de prueba]">
                …
        </TestCase>
        …
        <TestCase name="Cn" description="[descripción del caso de prueba]">
                …
        </TestCase>
<Finish/>
</Modeling>
```

Donde:

- n es el número del caso de prueba
- m es el número del paso del caso de prueba
- Las etiquetas “Start” y “Finish” representan el comienzo y el fin del flujo de los casos de prueba
- La etiqueta “TestCase” contiene información sobre el caso de prueba como el nombre y la descripción
- La etiqueta “Step” contiene información sobre los pasos asignados a cada caso de prueba como el nombre y la descripción

- La etiqueta “GUI_Object_Cn_Sm” contiene información sobre los elementos de la GUI de la aplicación asignados a cada paso de prueba como el nombre y el valor

4.4. Ejemplo de modelado gráfico

Se ha desarrollado una aplicación web de Gestión de Productos que se utilizará para crear ejemplos prácticos de modelado y automatización de las pruebas funcionales utilizando la solución propuesta. En el capítulo 7 se detallará el funcionamiento de esta aplicación.

Se han creado dos ejemplos:

- Un ejemplo sencillo que cubre sólo los dos casos de prueba de acceso y salida de la aplicación. Este ejemplo se utilizará a lo largo este trabajo.
- Un ejemplo que cubre todos los casos de prueba de la aplicación. Este ejemplo se detallará en el capítulo 7.

A continuación, vamos a modelar los casos de pruebas de acceso (*login*) y salida (*logout*) de la aplicación de ejemplo. La siguiente tabla contiene el detalle del plan pruebas:

Casos de prueba		**Pasos de los casos de prueba**	
Código	**Descripción**	**Código**	**Descripción**
C1	Acceso a la aplicación	C1_S1	Abrir la URL de la aplicación con un navegador web
		C1_S2	Escribir el nombre del usuario
		C1_S3	Escribir la contraseña del usuario
		C1_S4	Pulsar el botón "Acceder"
C2	Salir de la aplicación	C2_S1	Pulsar el botón "Salir"

Tabla 4 – Ejemplo (1) de una prueba funcional

El siguiente esquema muestra el flujo de los pasos de los dos casos de prueba de la aplicación de Gestión de Productos:

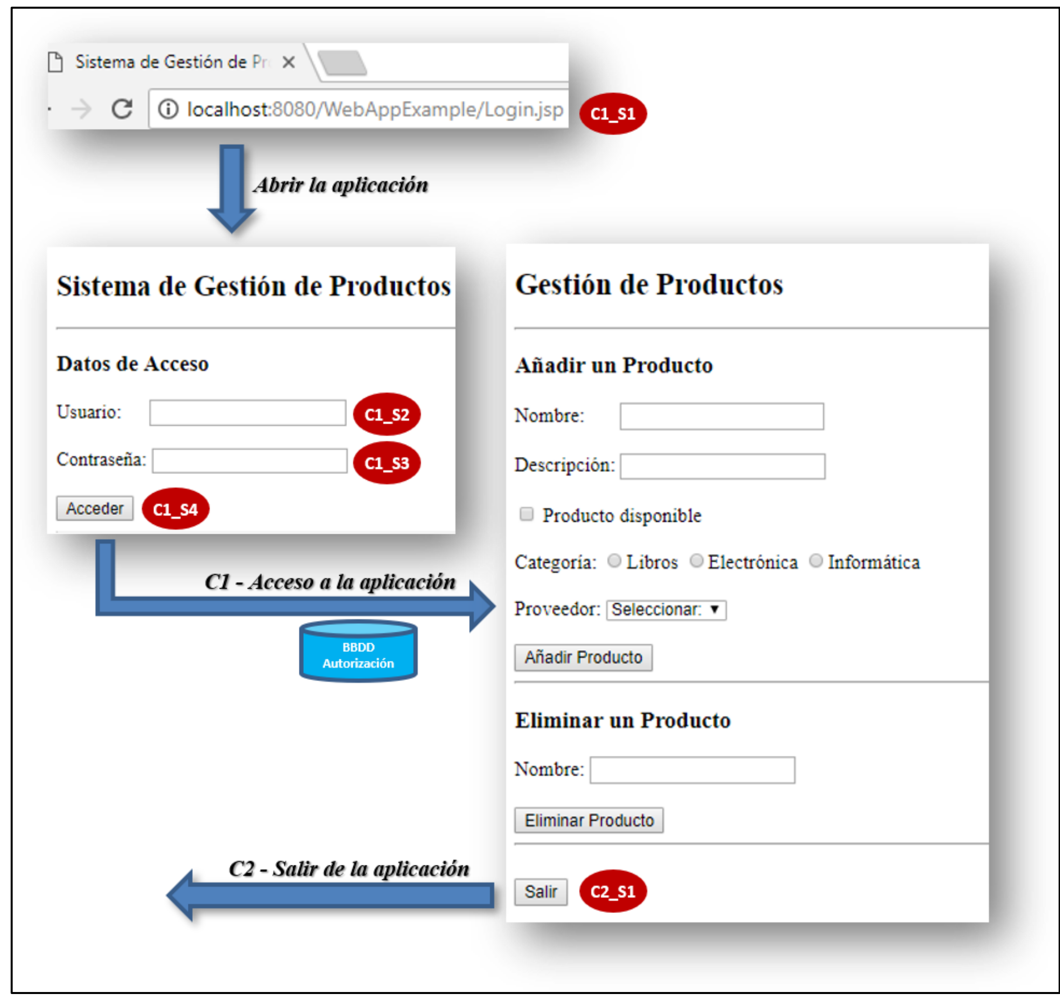

Ilustración 6 - Ejemplo (1) del flujo de los pasos de los casos de prueba de una aplicación

El siguiente esquema muestra el modelado de los dos casos de prueba de la aplicación (*login* y *logout*) utilizando el DSL visual.

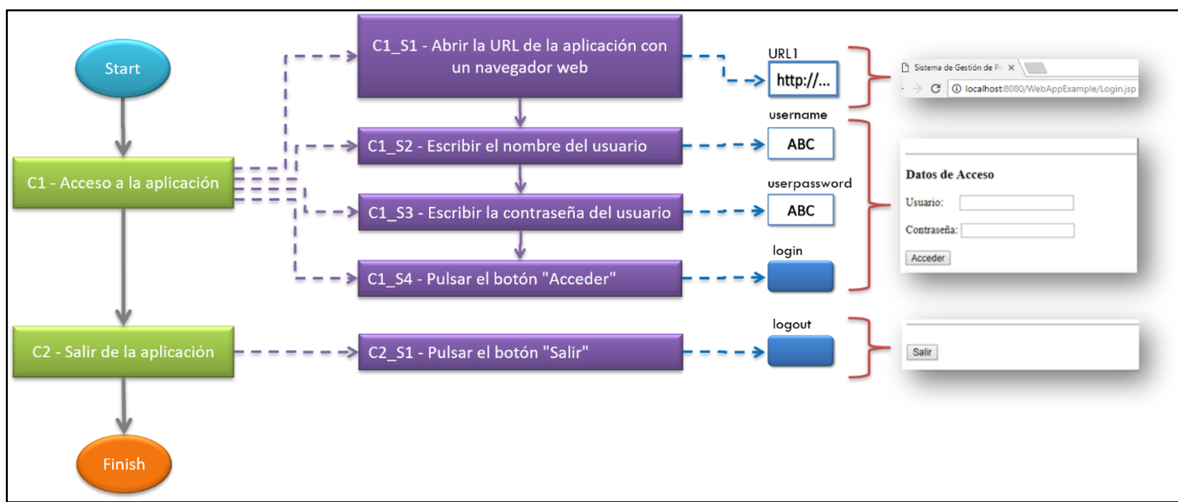

Ilustración 7 - Ejemplo (1) de modelado gráfico de las pruebas funcionales

En el modelado creado se representan gráficamente:

- El flujo de los casos de prueba C1 y C2.
- El flujo de los pasos de los casos de prueba C1_S1, C1_S2, C1_S3, C1_S4 y C2_S1.
- Los elementos de la GUI de la aplicación correspondientes a cada paso de los casos de prueba:
    - URL de la aplicación (*URL1*)
    - Cajas de Texto (*username* y *userpassword*)
    - Botones (*login* y *logout*)
- La relación entre los objetos del modelado.

4.5. Implementación de la solución en la metodología de desarrollo

4.5.1. Descripción general

Se recomienda adaptar la metodología de desarrollo para implementar la solución de forma óptima. En el ciclo de vida de desarrollo del Software, se pueden realizar los siguientes pasos utilizando la solución de modelado visual de las pruebas funcionales:

- En la Fase de Análisis, se describen los casos de prueba.
- En la Fase de Diseño, se detallan los pasos de los casos de prueba definidos.
- En la Fase Construcción, se definen los elementos de la GUI de la aplicación asignados a cada paso de los casos de prueba.
- En la Fase de Pruebas, se ejecutan las pruebas funcionales de forma manual o automática.

El siguiente esquema muestra cómo se puede implementar la solución de modelado propuesta en el ciclo de vida de desarrollo del Software:

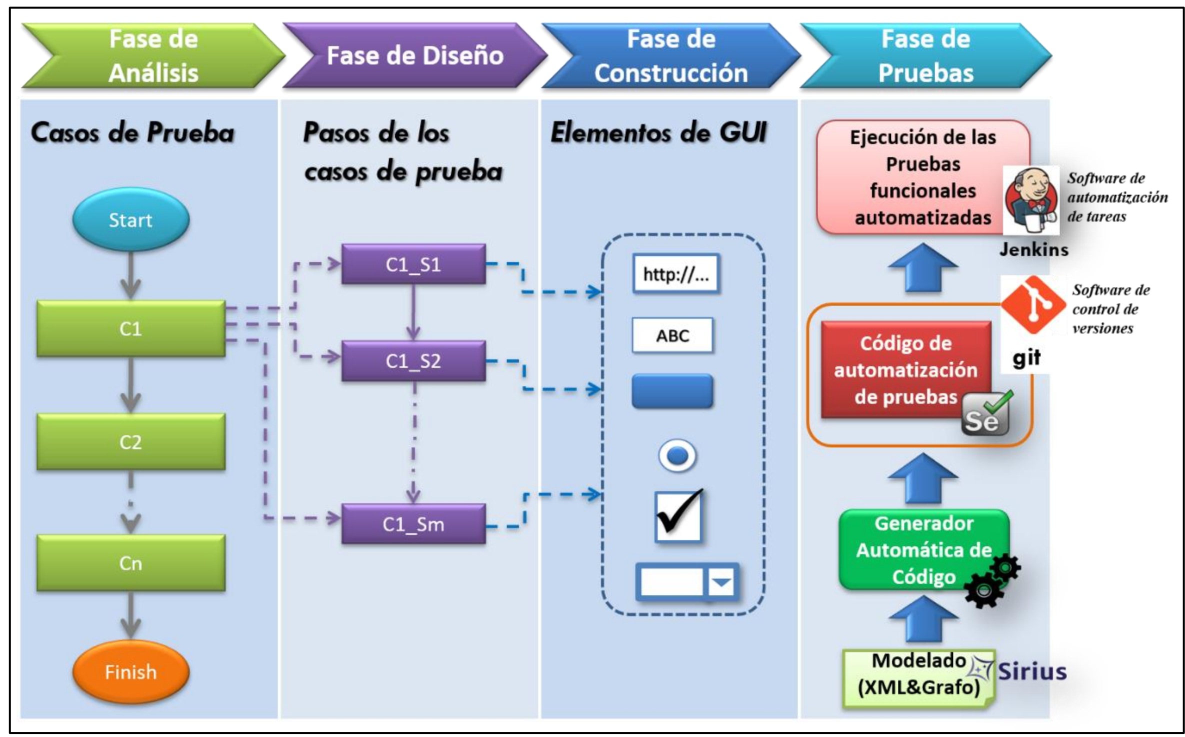

Ilustración 8 - Implementación de la solución de modelado en el ciclo de vida de desarrollo del Software

4.5.2. Implementación de la solución en el ciclo de vida de desarrollo del Software

A continuación, se detalla cómo implementar la solución en las fases del ciclo de vida de desarrollo del Software.

4.5.2.1. Fase de Análisis

En la Fase de Análisis del Software, los expertos del dominio o los analistas describen los casos de pruebas de los nuevos desarrollos según los requisitos funcionales. En esta fase se conoce el funcionamiento de la aplicación a alto nivel.

4.5.2.2. Fase de Diseño

En la Fase de Diseño del Software, los analistas detallan los pasos de cada caso de prueba. En esta fase ya se conoce el funcionamiento detallado de la aplicación.

4.5.2.3. Fase de Construcción

En la Fase Construcción del Software, los desarrolladores definen los elementos de la interfaz gráfica de usuario asignados a cada paso de los casos de prueba. En esta fase ya se conocen los detalles técnicos como los tipos (URL, TextBox, button, check box, radio button, combo box, etc.) y las propiedades (nombre, valor, etc.) de los elementos de la GUI de la aplicación.

4.5.2.4. Fase de Pruebas

En la Fase de Pruebas, se puede ejecutar las pruebas funcionales de forma manual o automática en el entorno de pruebas para verificar el correcto funcionamiento de la aplicación antes instalar el Software en el entorno de Producción.

La ejecución de las pruebas funcionales de forma manual se puede realizar por los probadores (*testers*) utilizando el modelado creado como referencia.

Para ejecutar las pruebas funcionales de forma automática, se debe realizar los siguientes pasos:

- Generar los scripts de automatización de las pruebas funcionales a partir del modelado creado utilizando un generador automático de código.
- Los scripts de automatización de las pruebas se guardan en un repositorio de una herramienta de control de versiones como Git.
- Creación y planificación de las tareas de ejecución automática de las pruebas funcionales utilizando una herramienta como Jenkins.

4.6. Ventajas de la solución propuesta

La solución propuesta se puede utilizar como base para crear herramientas de gestión de pruebas y para crear generadores de código de automatización de pruebas basados en el modelado de las pruebas funcionales.

La solución de modelado gráfico de las pruebas funcionales, tiene las siguientes ventajas:

- Se utiliza un lenguaje de modelado natural e intuitivo. El lenguaje de modelado gráfico específico de dominio, es más fácil de aprender que los lenguajes de modelado de propósito general como UML.

- El lenguaje de modelado se puede usar por los expertos del dominio sin necesidad de tener muchos conocimientos técnicos porque no se basa en programación textual.

- Tiempo de modelado rápido a comparación con otros tipos de modelado por utilizar un lenguaje gráfico.

- Con esta solución se puede empezar a modelar las pruebas funcionales de las aplicaciones desde la fase de análisis y luego se puede completar el modelado a lo largo ciclo de vida de desarrollo del Software.

- Es fácil mantener el modelado de las pruebas funcionales. No cuesta mucho esfuerzo añadir, modificar o eliminar un caso de prueba.

- El modelado contiene información gráfica sobre el orden de ejecución de los casos de prueba y el orden de ejecución de los pasos de los casos de prueba.

- En el mismo modelado hay detalles del funcionamiento de la aplicación tanto a alto nivel como a bajo nivel. De esta forma, el código de automatización generado a partir del modelado, es mantenible y es fácil de leer por contener información funcional sobre los casos de prueba y sobre los pasos de cada caso de prueba.

- El modelado se puede representar en formatos estructurados como XML y grafo. Eso facilita la generación de un código estructurado de automatización de las pruebas con un generador automático de código.

En los siguientes capítulos se detallará cómo se puede la implementar la solución para crear herramientas de gestión y automatización de las pruebas funcionales.

5. Diseño y creación de una herramienta de modelado gráfico de las pruebas funcionales

En este capítulo, se detallarán las tareas realizadas de diseño y creación de una herramienta de modelado que implementa el DSL visual de las pruebas funcionales de las aplicaciones. Además, se explicará cómo se puede usar la herramienta de modelado creada a través de ejemplos prácticos.

Los pasos principales para implementar el DSL visual son:

- Instalación del Software de modelado.
- Creación del metamodelo del DSL visual.
- Creación del proyecto de diseño de la herramienta de modelado.
- Creación de proyectos que implementan la herramienta de modelado. Se crearán dos ejemplos de modelado de las pruebas funcionales.

5.1. Instalación del Software de modelado

Se ha instalado el paquete del Software de modelado desde el sitio web de Eclipse.

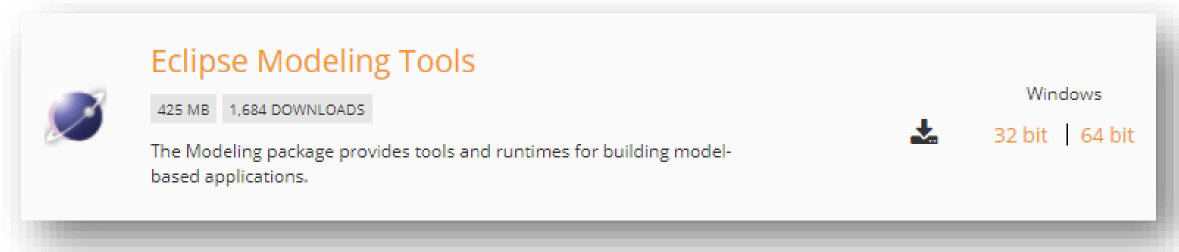

Ilustración 9 - Paquete del Software de modelado de Eclipse

Y se ha instalado el proyecto Sirius de Eclipse que facilita la creación de herramientas de modelado gráfico. Sirius es un Software libre y de código abierto basado en EMF y GMF y tiene soporte basado en los foros de Internet de la comunidad de Eclipse.

GMF (*Graphical Modeling Framework*) es un entorno de modelado gráfico de Eclipse para desarrollar editores gráficos basados en EMF (Entorno de Modelado de Eclipse) y GEF (Entorno de Edición Gráfica de Eclipse). GMF tiene dos componentes principales:

- GMF Tooling (definición de herramientas): Es un componente para definir el editor gráfico generado.
- GMF Runtime (entorno de ejecución): Es un componente para ejecutar el editor gráfico generado.

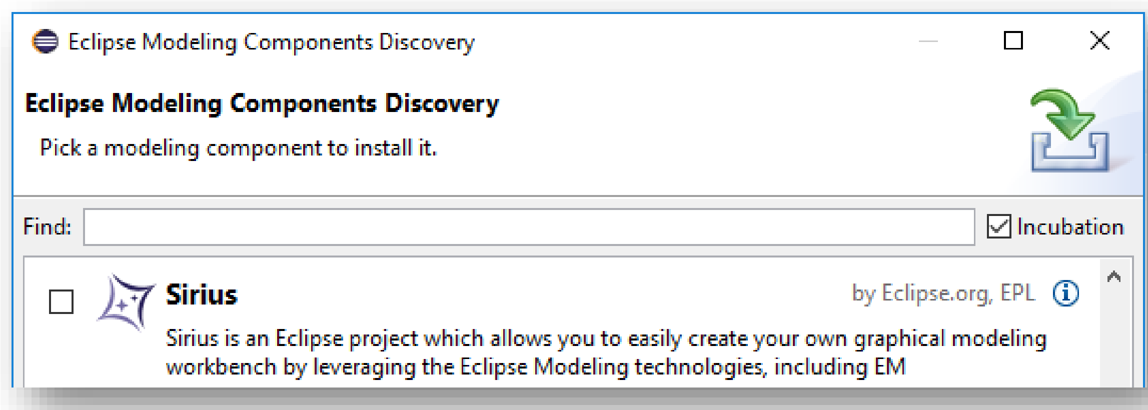

Ilustración 10 - Instalación del proyecto Sirius de Eclipse

El entorno de modelado creado con Sirius se compone de un conjunto de editores de Eclipse (diagramas, tablas y árboles) que permiten a los usuarios crear, editar y visualizar los modelados de EMF.

Se ha utilizado el proyecto Sirius por su capacidad para modelar arquitecturas complejas de dominios específicos y por la facilidad de configuración de las propiedades visuales y las acciones de los elementos gráficos de la herramienta de modelado creada.

5.2. Creación del metamodelo del DSL visual

Para definir el metamodelo del DSL visual, se crea un proyecto de modelado de Eclipse “testingModeling” utilizando las herramientas de Ecore (*Ecore Tools*) basadas en EMF. Las herramientas de Ecore proporcionan un entorno gráfico de trabajo de Eclipse para crear

metamodelos basados en diagramas de clases de UML que se guardan en ficheros con la extensión ".ecore".

Se crean las clases del metamodelo que representan los objetos del DSL visual (comienzo de la prueba, fin de la prueba, los casos de prueba, los pasos de prueba y los elementos de la GUI de la aplicación) y se definen las relaciones entre ellas.

En la siguiente tabla, se detallan los atributos y las relaciones de las clases del DSL visual:

Clase	Atributos	Relación con otras clases
TestingModeling	name (String)	
Start	name (String)	Relación de composición con la clase TestingModeling
Finish	name (String)	Relación de composición con la clase TestingModeling
TestCase	name (String), description (String)	Relación de composición con la clase TestingModeling
Step	name (String), description (String)	Relación de composición con las clases TestingModeling y TestCase
Elementos de la GUI (TextBox, Button, URL, CheckBox, RadioButton, ComboBox, …)	name (String), value (String)	Relación de composición con las clases TestingModeling y Step

Tabla 5 - Los atributos y las relaciones de las clases del DSL visual

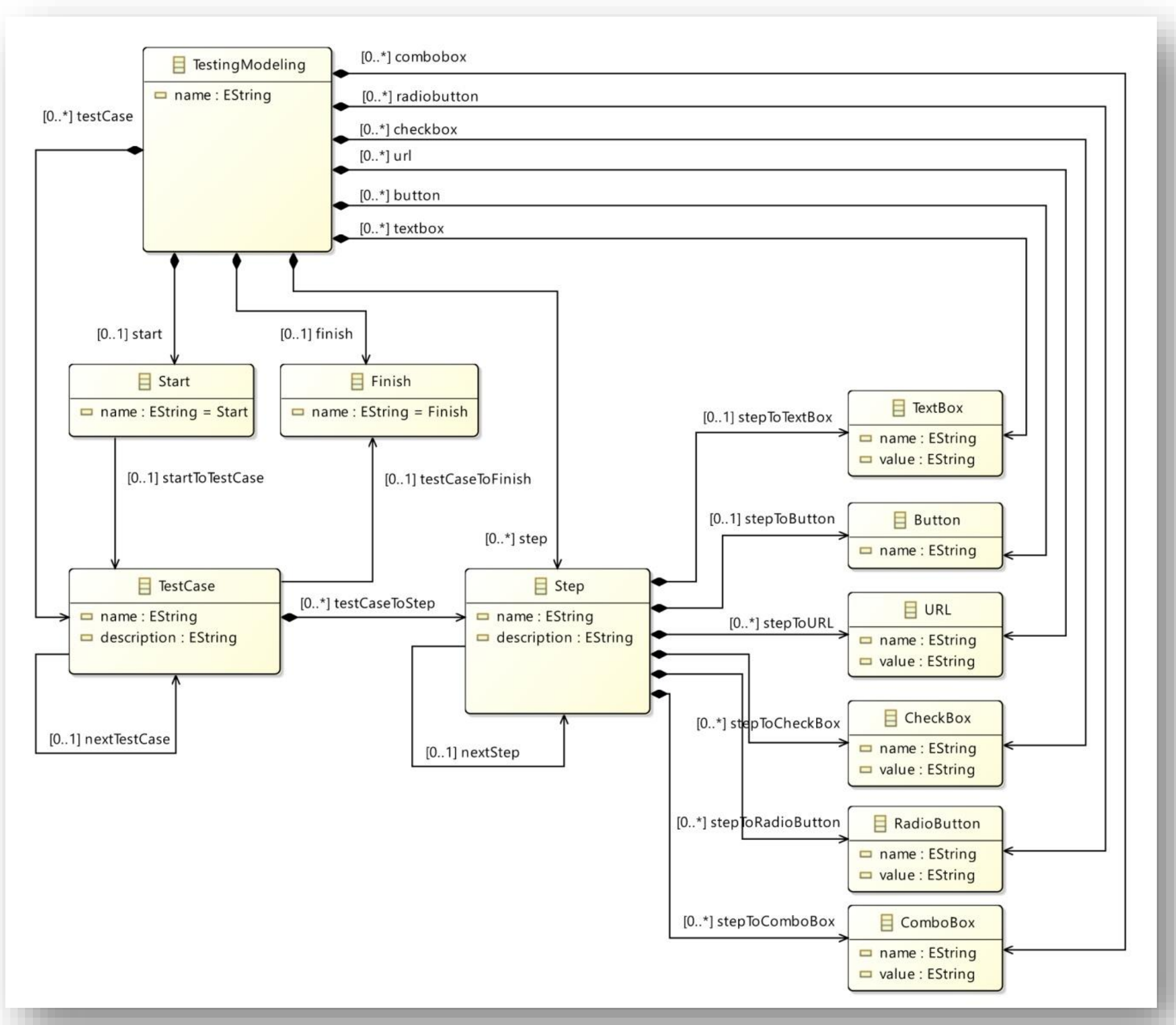

Ilustración 11 - Metamodelo del DSL visual de las pruebas funcionales

A través de EMF se genera el código fuente Java del metamodelo y se generan los dos proyectos “Edit” y “Editor” necesarios para definir el editor gráfico del modelado especifico de dominio:

- testingModeling.edit
- testingModeling.editor

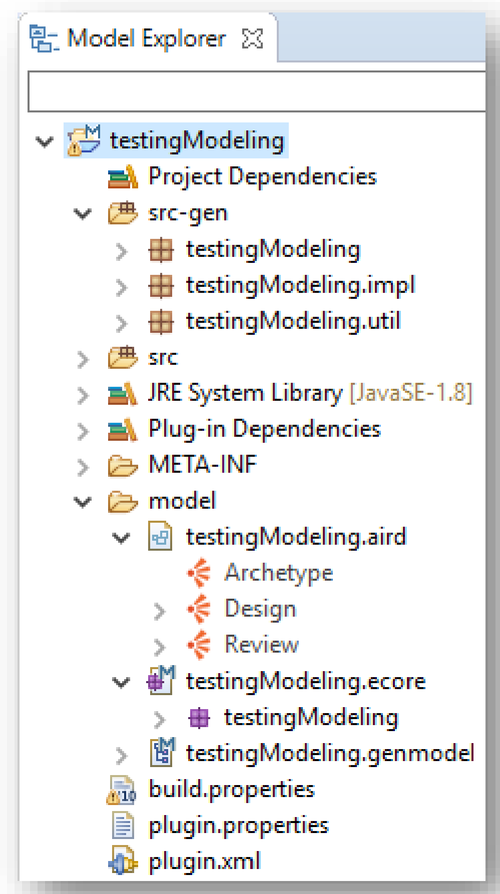

Ilustración 12 - Estructura del proyecto del metamodelo

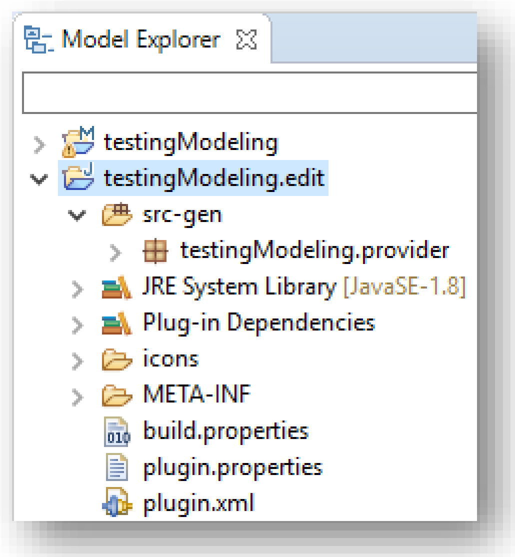

Ilustración 13 - Estructura del proyecto del Edit

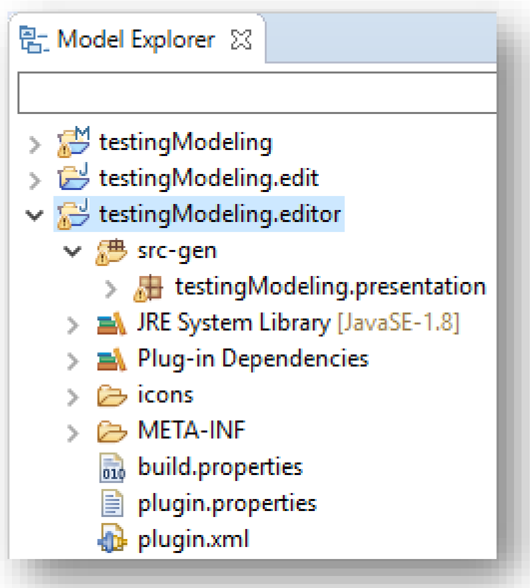

Ilustración 14 - Estructura del proyecto del Editor

Para implementar el metamodelo definido, hay que ejecutar un nuevo entorno de Eclipse.

5.3. Creación del proyecto de diseño de la herramienta de modelado

Se crea un proyecto "testingModeling.design" de Sirius tipo "*Viewpoint Specification Project*". Con este proyecto se define el diseño de la herramienta de modelado.

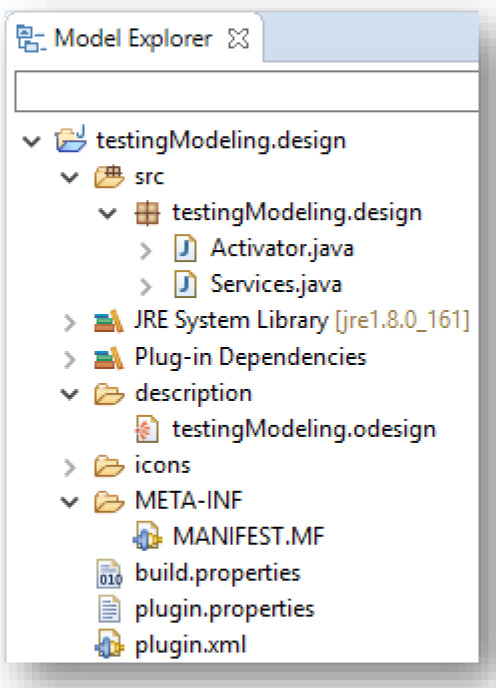

Ilustración 15 - Estructura del proyecto de diseño de la herramienta de modelado

Desde el archivo *MANIFEST.MF* del proyecto, se añade el complemento (*plug-in*) que define el metamodelo "testingModeling" del DSL Visual.

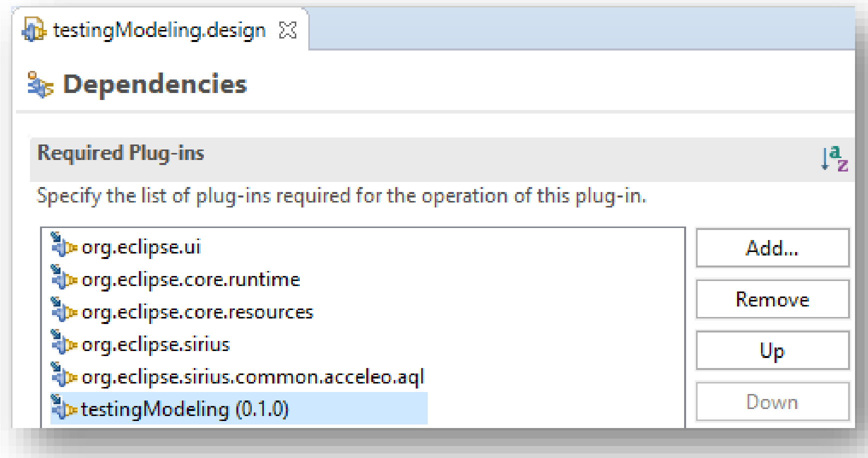

Ilustración 16 - Plug-in que define el metamodelo del DSL Visual

Se crea un *viewpoint* y se nombra como “TestingModeling”. El *viewpoint* proporciona un conjunto de representaciones (diagramas, tablas o árboles) que el usuario final podrá instanciar.

Se añade un diagrama con el nombre “TestingModeling diagram” al *viewpoint* y se le asigna el metamodelo “testingModeling” que define los tipos de objetos utilizados el modelado.

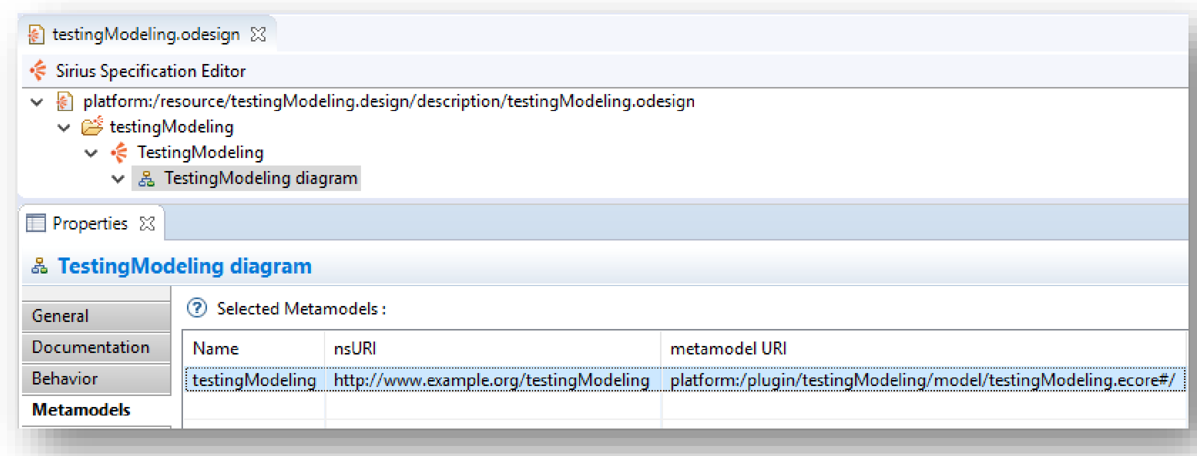

Ilustración 17 - Creación del Viewpoint y el diagrama del modelado

Se añaden y se definen los elementos de la herramienta de modelado. Los elementos son de dos tipos:

- **Nodos** que representan los objetos del modelado.
- **Enlaces** (o conexiones) que interconectan los objetos del modelado.

Se configuran las propiedades de cada nodo de modelado como el Id, el nombre de la clase de dominio, la etiqueta, el color y el estilo.

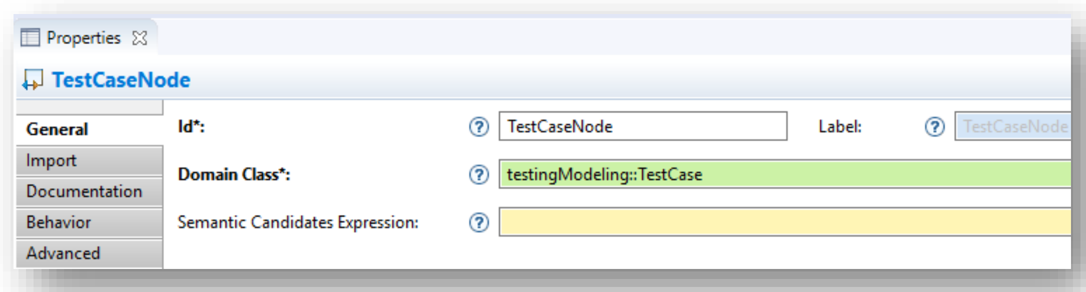

Ilustración 18 - Ejemplo de las propiedades de un nodo del modelado

Y se configuran las propiedades de cada enlace del modelado como el Id, la etiqueta, el nodo origen, el nodo destino, el color y el estilo.

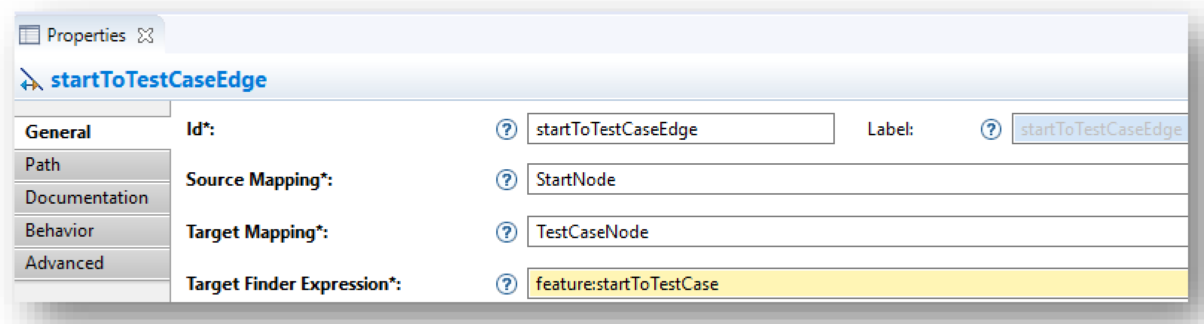

Ilustración 19 - Ejemplo de las propiedades de un enlace del modelado

Se han diseñado los iconos que representan los elementos de la GUI de las aplicaciones y se han añadido al proyecto.

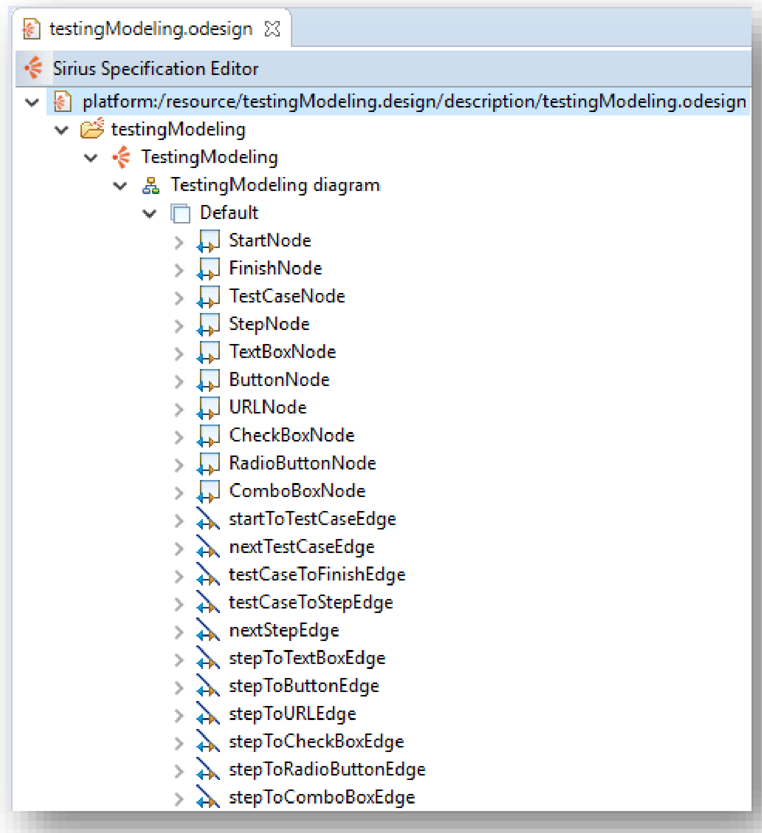

Ilustración 20 - Definición de los nodos y los enlaces del modelado

Los elementos de la herramienta de modelado se pueden agrupar en tres secciones (*TestCases*, *Steps* y *GUI Objects*) para facilitar su uso por los usuarios.

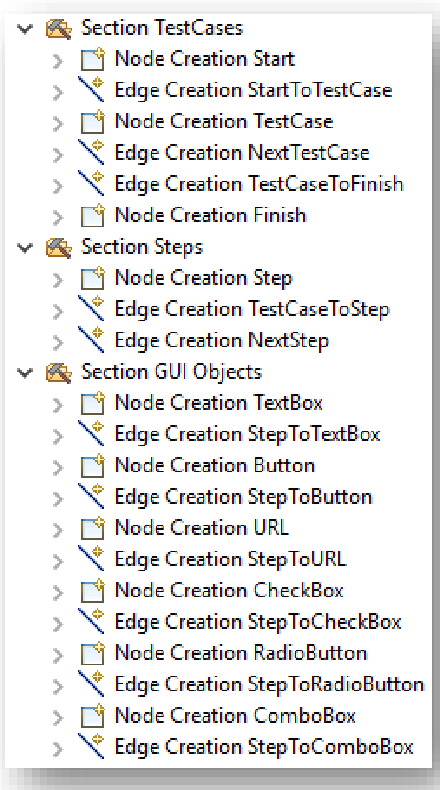

Ilustración 21 - Definición de las secciones de la herramienta de modelado

Se configuran las propiedades de cada nodo de las secciones de la herramienta de modelado como el Id, la etiqueta y el mapeo del nodo.

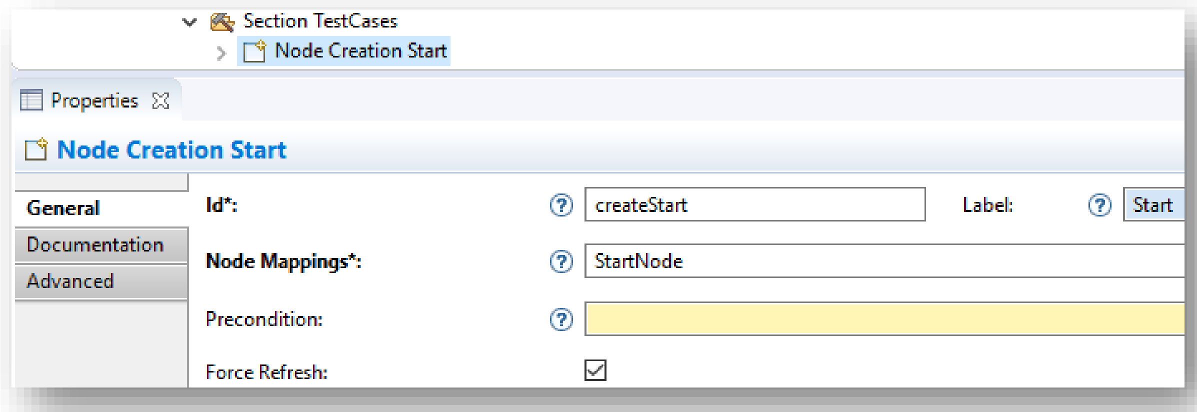

Ilustración 22 - Ejemplo de las propiedades de un nodo de una sección de la herramienta de modelado

Se configuran las propiedades de cada enlace de las secciones de la herramienta de modelado como el Id, la etiqueta y el mapeo del enlace.

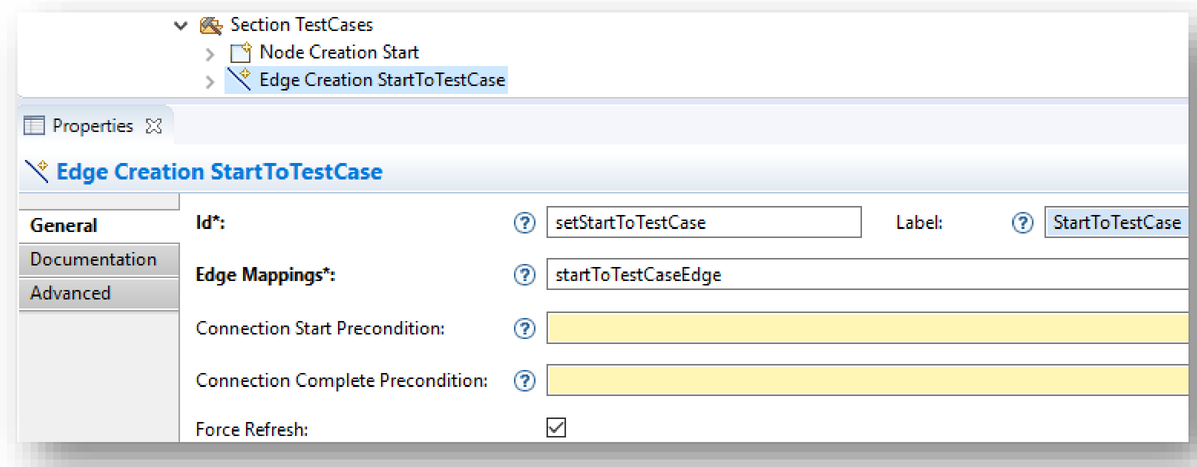

Ilustración 23 - Ejemplo de las propiedades de un enlace de una sección de la herramienta de modelado

Se configura la acción de cada nodo de la sección de la herramienta.

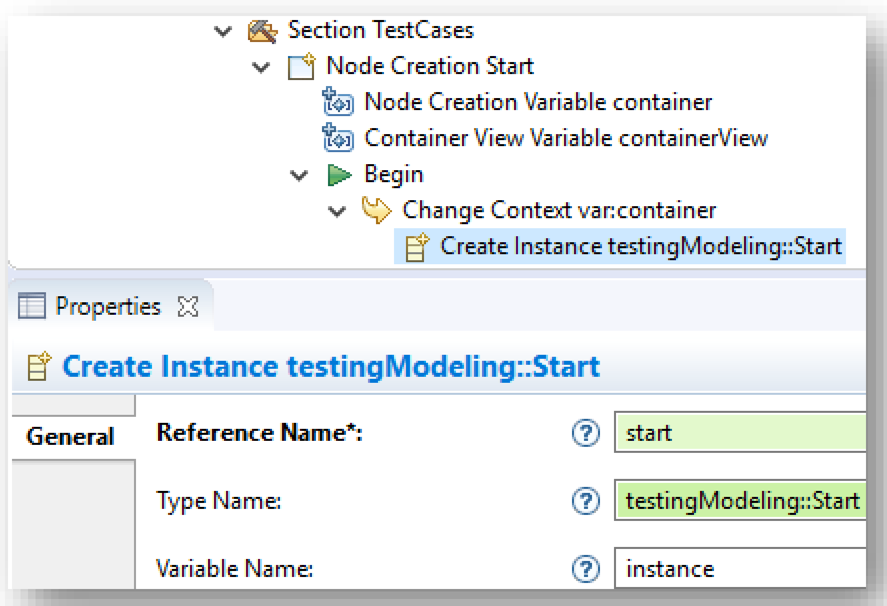

Ilustración 24 - Ejemplo de la configuración de la acción de un nodo de una sección de la herramienta de modelado

Se configura la acción de cada enlace de las secciones de la herramienta.

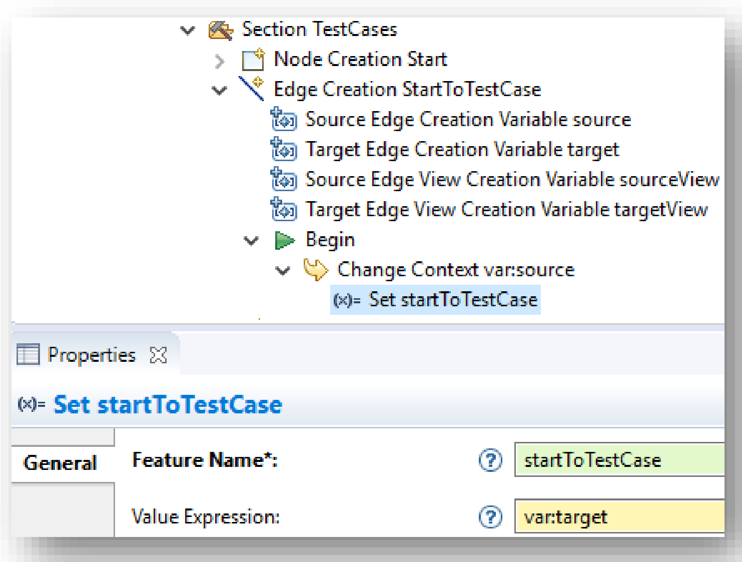

Ilustración 25 - Ejemplo de la configuración de la acción de un enlace de una sección de la herramienta de modelado

5.4. Creación de proyectos que implementan la herramienta de modelado

Para implementar la herramienta de modelado, se crean proyectos de modelado de Sirius y se les asigna el *viewpoint* "TestingModeling" creado con el proyecto de diseño de la herramienta de modelado.

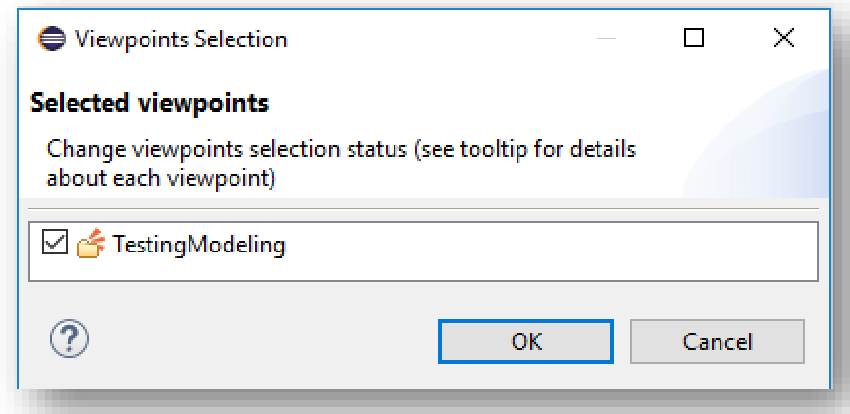

Ilustración 26 - Selección del viewpoint del proyecto de modelado

Se han creado dos proyectos como ejemplos prácticos de modelado de las pruebas funcionales de la aplicación web de Gestión de Productos:

- testingModeling.example
- testingModeling.example2

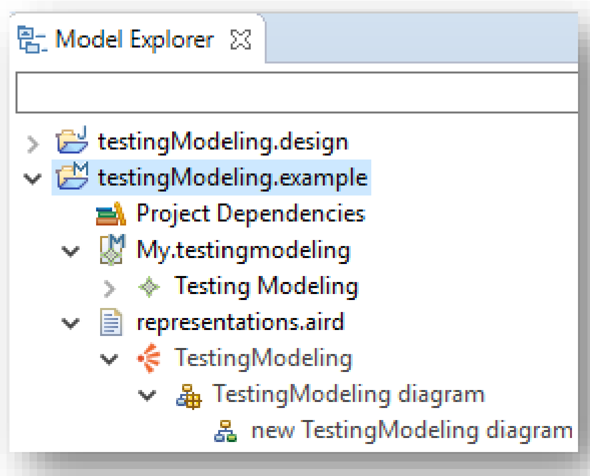

Ilustración 27 - Estructura de un proyecto que implementa la herramienta de modelado

5.5. Uso de la herramienta de modelado creada

La herramienta de modelado creada es intuitiva y es fácil de usar. Se puede crear el modelado de las pruebas funcionales gráficamente en dos formas principales:

- Creación del modelado arrastrando y soltando (*drag and drop*) los objetos del DSL visual.
- Creación del modelado gráficamente en forma de árbol.

5.5.1. Creación del modelado arrastrando y soltando los objetos del DSL visual

La herramienta creada proporciona una interfaz gráfica de usuarios para modelar las pruebas funcionales utilizando una paleta que contiene iconos que repreentan los objetos del modelado y las conexiones entre esos objetos. La paleta de la herramienta de modelado se ha dividido en tres secciones:

- Casos de prueba

- Pasos de los casos de prueba
- Los elementos de la GUI de la aplicación

Se puede utilizar el ratón y el teclado para arrastrar y soltar (*drag and drop*) los iconos de los objetos para formar el modelado de las pruebas funcionales.

A continuación, se muestra un ejemplo de modelado de los dos casos de prueba de acceso (*login*) y salida (*logout*) de la aplicación web de Gestión de Productos utilizando la herramienta de modelado creada que implementa el DSL visual:

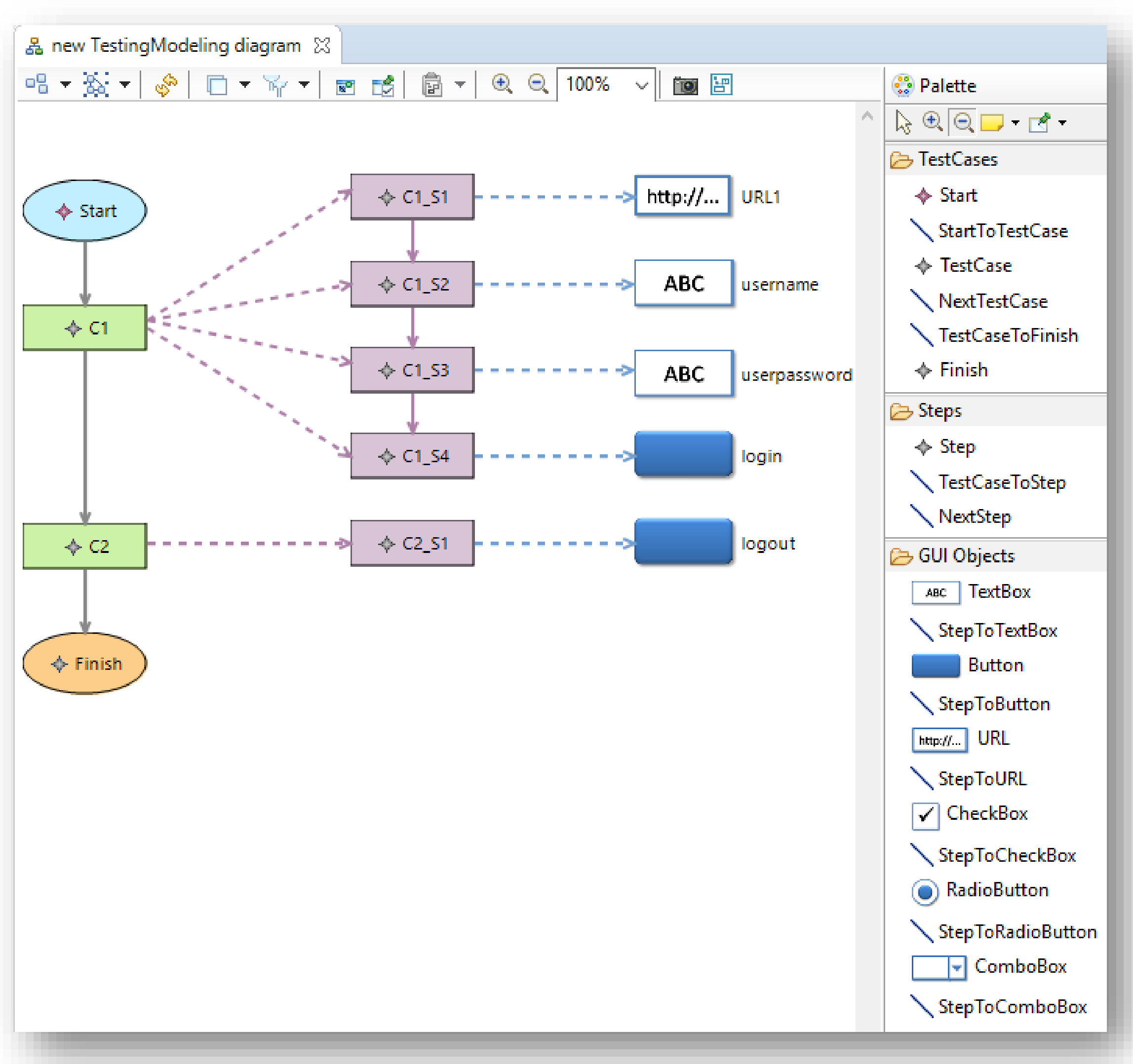

Ilustración 28 - Herramienta creada para el modelado gráfico de las pruebas funcionales – Ejemplo (1) de modelado

Después de crear los objetos del modelado gráfico, se completan las propiedades como el nombre, la descripción y el valor de cada objeto. Además, se puede cambir los colores y los estilos de esos objetos.

A continuación, se muestra cómo se configuran las propiedades de los casos de prueba, los pasos de los casos de prueba y los elementos de la GUI de la aplicación web de Gestión de Productos.

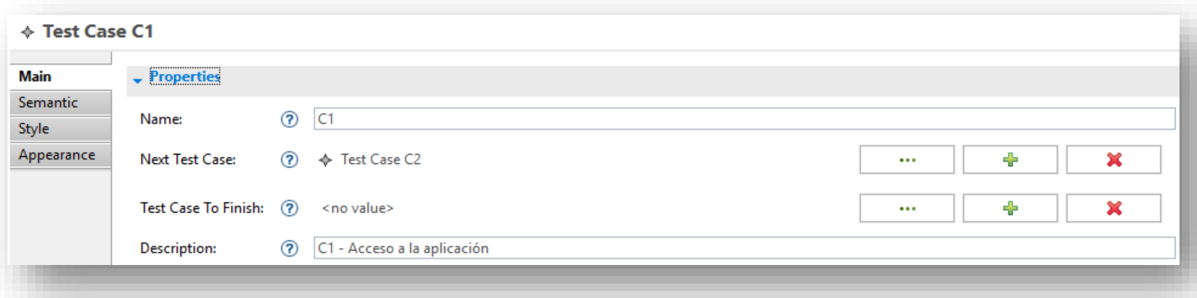

Ilustración 29 - Ejemplo (1) - Propiedades de un caso de prueba en la herramienta de modelado

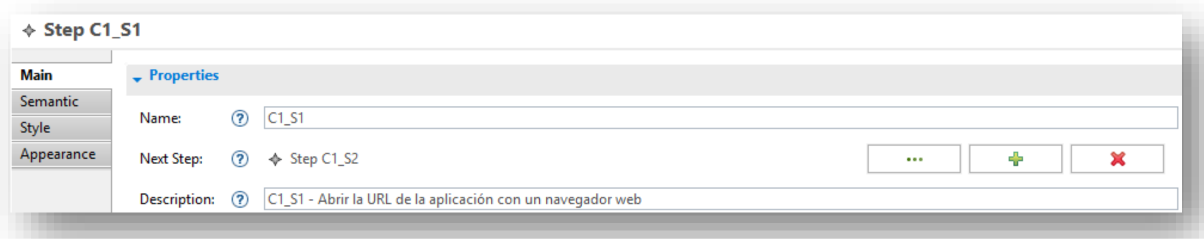

Ilustración 30 - Ejemplo (1) - Propiedades de un paso de un caso de prueba en la herramienta de modelado

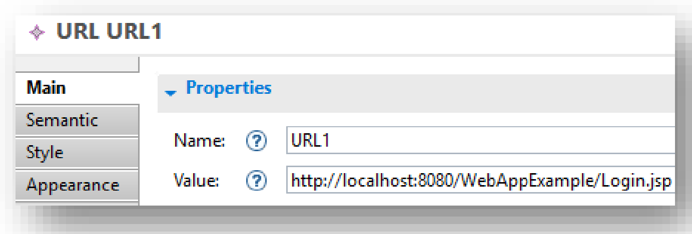

Ilustración 31 - Ejemplo (1) - Propiedades de un elemento de la GUI de la aplicación

5.5.2. Creación del modelado gráficamente en forma de árbol

Con la herramienta de modelado, se puede crear gráficamente el modelado de pruebas funcionales en forma de árbol de la siguiente manera:

- A partir de la raíz del modelado "TestingModeling", se puede crear el comienzo de la prueba (*Start*), los casos de prueba (*TestCase*) y el fin de la prueba (*Finish*).
- A partir de los casos de prueba se puede crear los pasos de la prueba.
- A partir de los pasos de los casos de prueba se puede crear los elementos de la GUI de la aplicación.

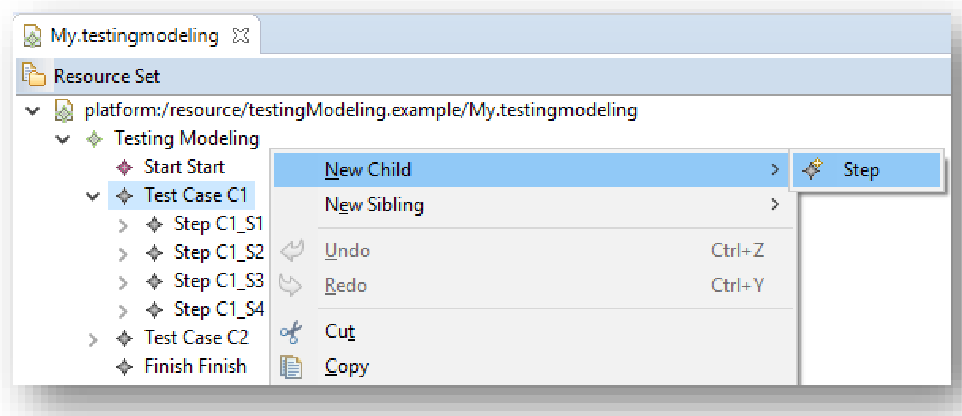

Ilustración 32 - Creación del árbol del modelado de las pruebas funcionales

Se puede proporcionar las propiedades de cada objeto del modelado como el nombre, la descripción y las relaciones con los otros objetos.

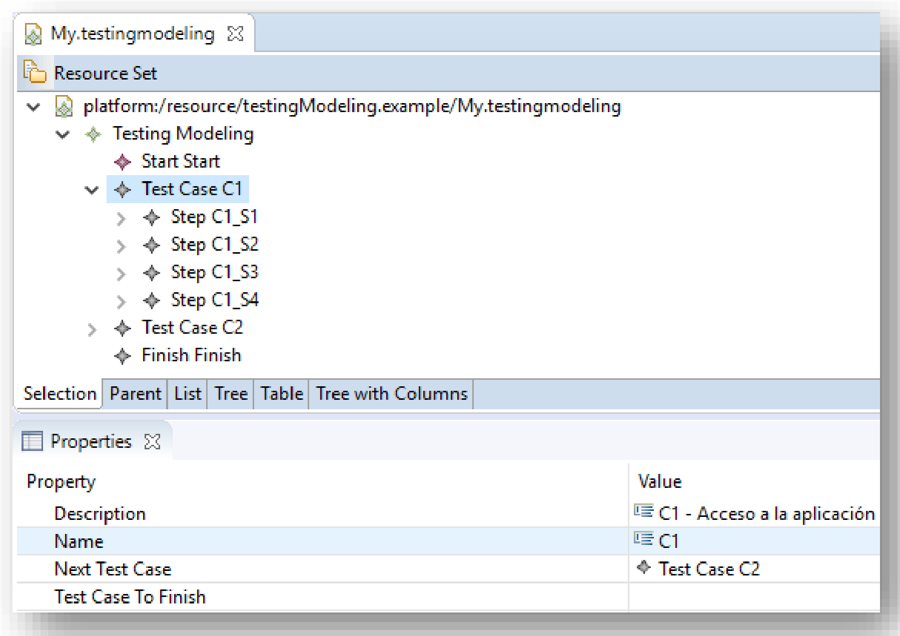

Ilustración 33 – Ejemplo (1) - Las propiedades de un objeto del árbol del modelado

En la siguiente imagen se muestra un ejemplo de una estructura de árbol del modelado de una prueba funcional, creado con la herramienta de modelado:

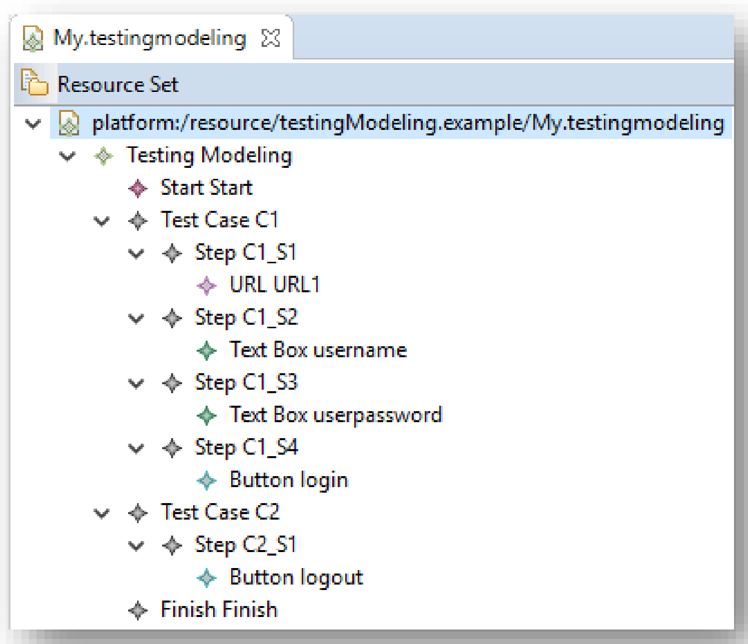

Ilustración 34 - Ejemplo (1) - La estructura de árbol del modelado de las pruebas funcionales

6. Creación de un generador de código de automatización de las pruebas funcionales

6.1. Funcionamiento del generador de código

Se ha desarrollado un generador automático de código para generar el código fuente de los scripts de automatización de las pruebas funcionales a partir del modelado creado con la herramienta de modelado.

Se puede generar un código de automatización de pruebas compatible con varios tipos de navegadores. En el piloto, el código generado es compatible con los navegadores Chrome y Firefox.

El siguiente esquema muestra el funcionamiento general del generador automático de código:

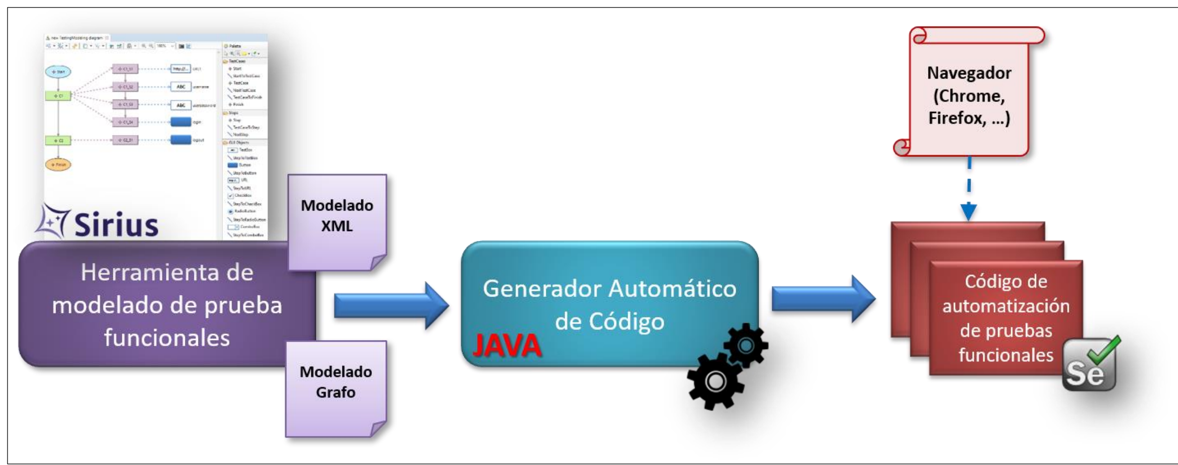

Ilustración 35 - Funcionamiento general del Generador Automático de Código

El piloto del generador automático de código proporciona una interfaz gráfica de usuario que recibe los siguientes datos de entrada:

- Ruta y nombre del fichero del código de automatización de las pruebas funcionales que se va a generar.
- Ruta y nombre del fichero XML del modelado creado con la herramienta de modelado de las pruebas de funcionales. Este fichero contiene información sobre las propiedades de cada objeto del modelado como el nombre, la descripción y el valor.

- Ruta y nombre del fichero del grafo del modelado creado con la herramienta de modelado de las pruebas de funcionales. Este fichero contiene información sobre los objetos del modelado gráfico y la relación entre ellos.

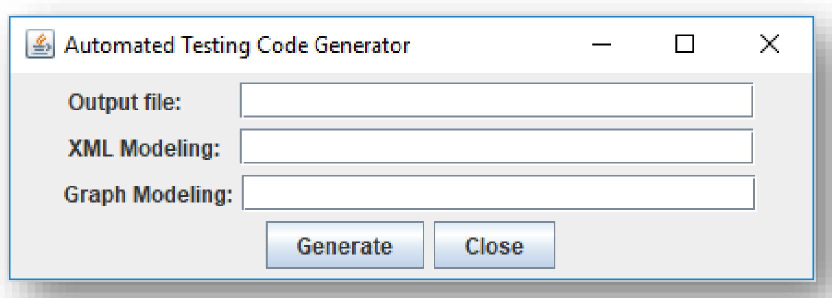

Ilustración 36 - Interfaz gráfica de usuario del Generador Automático de Código

El código fuente generado tiene las siguientes características:

- El modelado gráfico creado con el DSL visual se convierte automáticamente en instrucciones de Selenium.
- El código fuente generado tiene forma estructurada.
- Los nombres de los métodos de la clase generada, se construyen utilizando los códigos de los casos de prueba en el modelado de la prueba funcional.
- Los comentarios de los métodos de la clase generada, se construyen utilizando los códigos y las descripciones de los casos de prueba y de los pasos de los casos de prueba en el modelado de la prueba funcional.

6.2. Arquitectura del Software del generador de código

El desarrollo del Software del generador automático de código, se ha realizado con el lenguaje de programación Java en el entorno de Eclipse.

En el desarrollo del generador de código, se ha utilizado el patrón de diseño de MVC (Modelo-Vista-Controlador). MVC separa el código de la aplicación en tres componentes principales:

- Vista (*View*): Representa la interfaz gráfica de usuario.
- Controlador (*Controller*): Representa las acciones del usuario.
- Modelo (*Model*): Representa la lógica de negocio del Software.

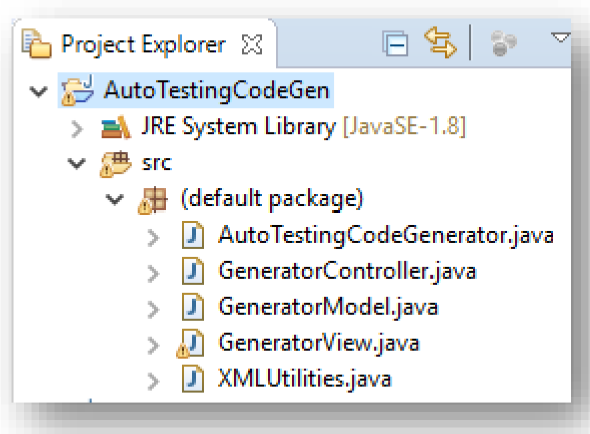

Ilustración 37 - Estructura del proyecto del generador automático de código

6.2.1. Código de la GUI (Vista de MVC)

Se ha utilizado Java Swing para crear la interfaz gráfica de usuario del generador automático de código.

A continuación, se muestra el código de la clase Java de la Vista del patrón de diseño MVC (GeneratorView.java):

```
import java.awt.*;
import javax.swing.*;


/**
 * Clase de la Vista del patrón de diseño MVC
 */
public class GeneratorView extends JFrame {

        //Etiqueta de la caja de texto del nombre del archivo a generar
        JLabel OutputFileNameLabel;

        //Caja de texto del nombre de archivo a generar
        JTextField OutputFileName;

        //Etiqueta de la caja de texto del nombre de fichero XML del modelado
        JLabel XMLFileNameLabel;

        //Caja de texto del nombre de fichero XML del modelado
        JTextField XMLFileName;

        //Etiqueta de la caja de texto del nombre del fichero del grafo del modelado
```

```
	JLabel GraphFileNameLabel;

	//Caja de texto del nombre del fichero del grafo del modelado
	JTextField GraphFileName;

	//Botón de generación del código de automatización de las pruebas funcionales
	JButton generationButton;

	//Botón de cierre del generador
	JButton closeButton;


	/**
	 * Constructor de la clase
	 */
	public GeneratorView() {

		//Se crea el contenedor y se definen el título y el tamaño de la ventana del generador
		Container container = getContentPane();
		setTitle("Automated Testing Code Generator");
		setLayout(new FlowLayout());
		setSize(450,150);

		//Se construyen los elementos de la interfaz gráfica del generador
		OutputFileNameLabel = new JLabel("Output file:        ");
		OutputFileName = new JTextField("", 25);

		XMLFileNameLabel = new JLabel("XML Modeling:  ");
		XMLFileName = new JTextField("", 25);

		GraphFileNameLabel = new JLabel("Graph Modeling:");
		GraphFileName = new JTextField("", 25);

		generationButton = new JButton("Generate");
		closeButton = new JButton("Close");

		//Se añaden al contenedor los elementos de la interfaz gráfica del generador
		container.add(OutputFileNameLabel);
		container.add(OutputFileName);
		container.add(XMLFileNameLabel);
		container.add(XMLFileName);
		container.add(GraphFileNameLabel);
		container.add(GraphFileName);
		container.add(generationButton);
		container.add(closeButton);

		setVisible(true);
	}

} //Cierre de la clase
```

6.2.2. Código de las acciones del usuario (Controlador de MVC)

En la interfaz gráfica del generador, el usuario puede realizar dos acciones:

- Generar el código de automatización de las pruebas funcionales pulsando el botón "Generate".

- Cerrar la ventana de la interfaz gráfica del generador pulsando el botón “Close”.

A continuación, se muestra el código de la clase Java del Controlador del patrón de diseño MVC (GeneratorController.java):

```
import java.awt.event.*;
import javax.swing.*;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.SAXException;
import java.io.IOException;


/**
 * Clase del Controlador del patrón de diseño MVC
 */
public class GeneratorController implements ActionListener{

	GeneratorModel model;
	GeneratorView view;

	/**
	 * Constructor de la clase
	 * @param view
	 * @param model
	 */
	public GeneratorController(GeneratorView view, GeneratorModel model) {

		this.model = model;
		this.view = view;
		actionListener(this);
	}


	/**
	 * Acción que se realiza cuando se pulsa el botón "Generate"
	 * @param event
	 */
	public void actionPerformed(ActionEvent event) {

		//Nombre del fichero generado
		String outputFileName = this.view.OutputFileName.getText();

		//Nombre del fichero XML del modelado
		String xmlFileName = this.view.XMLFileName.getText();

		//Nombre del fichero del grafo del modelado
		String graphFileName = this.view.GraphFileName.getText();

		try {
			model.generateTestingCode(outputFileName, xmlFileName, graphFileName);
		} catch (ParserConfigurationException e) {
			e.printStackTrace();
			JOptionPane.showMessageDialog(null, "Error message: " + e);
		} catch (SAXException e) {
			e.printStackTrace();
			JOptionPane.showMessageDialog(null, "Error message: " + e);
		} catch (IOException e) {
			e.printStackTrace();
```

```
                        JOptionPane.showMessageDialog(null, "Error message: " + e);
                }
        }


        /**
         * Acciones del listener de la clase GeneratorController
         * @param event
         */
        public void actionListener(ActionListener event) {

                //Añadir la acción del botón "Generate"
                view.generationButton.addActionListener(event);

                //Añadir la acción del botón "Close"
                view.closeButton.addActionListener(new Close());
        }


        /**
         * Acción del botón "Close"
         */
        private class Close implements ActionListener{
           public void actionPerformed(ActionEvent e) {
              System.exit(0);
           }
        }


} //Cierre de la clase
```

6.2.3. Código del modelo de negocio (Modelo de MVC)

En el modelo de negocio del generador, se construye el código de automatización de las pruebas funcionales a partir de los ficheros XML de entrada que contienen la información sobre los objetos del modelado y la relación entre ellos. Los nombres y las rutas de los ficheros XML de entrada y el fichero Java de salida, se proporcionan por el usuario a través de la interfaz gráfica del generador.

A continuación, se muestra el código de la clase Java del Modelo del patrón de diseño MVC (GeneratorModel.java):

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.*;
import org.xml.sax.SAXException;


/**
 * Clase del Modelo del patrón de diseño MVC
 */
```

```
public class GeneratorModel {

	XMLUtilities xmlUtilities = new XMLUtilities();

	/**
	 * @param ouputFileName
	 * @param xmlFileName
	 * @param graphFileName
	 * @throws ParserConfigurationException
	 * @throws SAXException
	 * @throws IOException
	 */
	public void generateTestingCode(String ouputFileName, String xmlFileName, String
graphFileName) throws ParserConfigurationException, SAXException, IOException {

		//Nombre de la clase Java a generar
		String className = ouputFileName.substring(ouputFileName.lastIndexOf("\\") + 1,
ouputFileName.indexOf(".java"));

		//Partes del código Java generado
		StringBuffer scriptCode1 = new StringBuffer();
		StringBuffer scriptCode2 = new StringBuffer();
		StringBuffer scriptCode3 = new StringBuffer();
		StringBuffer scriptCode4 = new StringBuffer();
		StringBuffer scriptCode4_1 = new StringBuffer();
		StringBuffer scriptCode4_2 = new StringBuffer();
		StringBuffer scriptCode4_3 = new StringBuffer();
		StringBuffer scriptCode5 = new StringBuffer();

		//Código Java generado
		StringBuffer generatedCode = new StringBuffer();

		//Parsear los archivos XML de entrada
		Document XMLDoc = xmlUtilities.XMLParser(xmlFileName);
		Document GraphDoc = xmlUtilities.XMLParser(graphFileName);

		//Se obtiene una lista de los nodos de los elementos del grafo
		NodeList graphObjectsList =
GraphDoc.getElementsByTagName("ownedDiagramElements");

		//Se obtiene del grafo una lista de los nodos de los casos de prueba
		NodeList testCaseList = XMLDoc.getElementsByTagName("testCase");

		//Se obtiene del grafo una lista de los nodos de los pasos de los casos de prueba
		NodeList stepList = XMLDoc.getElementsByTagName("testCaseToStep");

		//Identificador del comienzo de los casos de prueba
		String StartID = null;

		//Identificador del caso de prueba
		String testCaseID = null;

		//Nombre del caso de prueba
		String testCaseName = null;

		//Descripción del caso de prueba
		String testCaseDesc = null;

		//Identificador del paso del caso de prueba
		String stepID = null;
```

```
			//Nombre del paso del caso de prueba
			String stepName = null;

			//Descripción del paso del caso de prueba
			String stepDesc = null;

			//Tipo de elemento de la GUI
			String guiElementType = null;

			//Nombre del elemento de la GUI
			String guiElementName = null;

			//Valor del elemento de la GUI
			String guiElementValue = null;

			//Obtener el identificador del nodo de comienzo de los casos de prueba
			StartID = xmlUtilities.getNodeID(graphObjectsList, "Start", "Start");

			//Quitar los nodos hijos "#text" de la lista de casos de prueba
			testCaseList = xmlUtilities.removeTextFromChildNodes(testCaseList);

			//Obtener la parte 1 de 5 del código generado
			scriptCode1 = generateScriptCode1(className);

			for (int i=0; i<testCaseList.getLength(); i++) {

				if(i==0) {
					testCaseID = xmlUtilities.getNextNodeID(graphObjectsList, StartID,
"TestCase");
				} else {
					testCaseID = xmlUtilities.getNextNodeID(graphObjectsList, testCaseID,
"TestCase");
				}

				testCaseName = xmlUtilities.getNodeName(graphObjectsList, "TestCase",
testCaseID);
				testCaseDesc = xmlUtilities.getNodeDescription(testCaseList, testCaseName);

				//Obtener la parte 2 de 5 del código generado
				scriptCode2 = scriptCode2.append(generateScriptCode2(testCaseName,
testCaseDesc));

				//Obtener la parte 4_1 de 5 del código generado
				scriptCode4_1 = generateScriptCode4_1(testCaseName, testCaseDesc);

				//Obtener la parte 4 de 5 del código generado
				scriptCode4 = scriptCode4.append(scriptCode4_1);

				//Obtener la parte 4_3 de 5 del código generado
				scriptCode4_3 = generateScriptCode4_3(testCaseName);

				//Obtener la lista de nodos de los pasos del caso de prueba
				NodeList stepNodes = testCaseList.item(i).getChildNodes();

				//Quitar los nodos hijos "#text" de la lista de los pasos
				stepNodes = xmlUtilities.removeTextFromChildNodes(stepNodes);

				for (int n=0; n<stepNodes.getLength(); n++) {
```

```
					//Obtener el nodo del paso del caso de prueba
					Node stepNode = stepNodes.item(n);

					if (n==0) {
						stepID = xmlUtilities.getFirstStepID(testCaseList,
graphObjectsList, testCaseName);
					} else {
						stepID = xmlUtilities.getNextNodeID(graphObjectsList, stepID,
"Step");
					}

					stepName = xmlUtilities.getNodeName(graphObjectsList, "Step",
stepID);
					stepDesc = xmlUtilities.getNodeDescription(stepList, stepName);

					//Obtener los nodos de los elementos de la GUI del paso del caso de
prueba
					NodeList guiNodes = stepNode.getChildNodes();

					for (int j=0; j<guiNodes.getLength(); j++) {

						Node guiNode = guiNodes.item(j);

						//Obtener el elemento de la GUI
						Element guiElement = (Element) guiNode;
						guiElementType = guiNode.getNodeName().substring(6) ;
						guiElementName = guiElement.getAttribute("name");
						guiElementValue = guiElement.getAttribute("value");

						//Obtener la parte 4_2 de 5 del código generado
						scriptCode4_2 = generateScriptCode4_2(stepName, stepDesc,
guiElementType, guiElementName, guiElementValue);

						//Obtener la parte 4 de 5 del código generado
						scriptCode4 = scriptCode4.append(scriptCode4_2);
					}
				}

				//Obtener la parte 4 de 5 del código generado
				scriptCode4 = scriptCode4.append(scriptCode4_3);
			}

			//Obtener la parte 3 de 5 del código generado
			scriptCode3 = generateScriptCode3();

			//Obtener la parte 5 de 5 del código generado
			scriptCode5 = generateScriptCode5();

			//Se obtiene el código completo generado
			generatedCode =
scriptCode1.append(scriptCode2).append(scriptCode3).append(scriptCode4).append(scriptCode5);

			//Se crea el archivo del código completo generado
			generateScriptFile(ouputFileName, generatedCode);

		}


		/**
		 * @param className
```

```
	 * @return código Java de automatización de las pruebas funcionales (parte 1 de 5)
	 */
	public static StringBuffer generateScriptCode1(String className) {

		StringBuffer scriptCode1 = new StringBuffer();

		scriptCode1.append("import java.util.concurrent.TimeUnit;");
		scriptCode1.append("\n");
		scriptCode1.append("import java.util.List;");
		scriptCode1.append("\n");
		scriptCode1.append("import org.openqa.selenium.By;");
		scriptCode1.append("\n");
		scriptCode1.append("import org.openqa.selenium.WebDriver;");
		scriptCode1.append("\n");
		scriptCode1.append("import org.openqa.selenium.WebElement;");
		scriptCode1.append("\n");
		scriptCode1.append("import org.openqa.selenium.chrome.ChromeDriver;");
		scriptCode1.append("\n");
		scriptCode1.append("import org.openqa.selenium.support.ui.Select;");
		scriptCode1.append("\n");
		scriptCode1.append("import org.openqa.selenium.firefox.FirefoxDriver;");
		scriptCode1.append("\n\n\n");
		scriptCode1.append("/**");
		scriptCode1.append("\n");
		scriptCode1.append(" * Clase del script de automatización de las pruebas funcionales");
		scriptCode1.append("\n");
		scriptCode1.append(" *");
		scriptCode1.append("\n");
		scriptCode1.append(" */");
		scriptCode1.append("\n");
		scriptCode1.append("public class " + className + " {");
		scriptCode1.append("\n\n");
		scriptCode1.append("	/**");
		scriptCode1.append("\n");
		scriptCode1.append("	 * Método principal de ejecución de las pruebas funcionales");
		scriptCode1.append("\n");
		scriptCode1.append("	 * @param args");
		scriptCode1.append("\n");
		scriptCode1.append("	 * @throws Exception");
		scriptCode1.append("\n");
		scriptCode1.append("	 */");
		scriptCode1.append("\n");
		scriptCode1.append("	public static void main(String[] args) throws Exception {");

		scriptCode1.append("\n\n");
		scriptCode1.append("		//Se obtiene el nombre del navegador");
		scriptCode1.append("\n");
		scriptCode1.append("		String browser = args[0];");
		scriptCode1.append("\n\n");
		scriptCode1.append("		//Se crea el objeto del driver del navegador");
		scriptCode1.append("\n");
		scriptCode1.append("		WebDriver driver = getDriver(browser);");
		scriptCode1.append("\n\n");
		scriptCode1.append("		//Se define el tiempo máximo de espera ");
		scriptCode1.append("\n");
		scriptCode1.append("		driver.manage().timeouts().implicitlyWait(10,
TimeUnit.SECONDS);");

		return scriptCode1;
	}
```

```
	/**
	 * @param testCaseName
	 * @param testCaseDesc
	 * @return código Java de automatización de las pruebas funcionales (parte 2 de 5)
	 */
	public static StringBuffer generateScriptCode2(String testCaseName, String testCaseDesc) {

		StringBuffer scriptCode2 = new StringBuffer();

		scriptCode2.append("\n\n");
		scriptCode2.append("			//Se ejecuta el caso de prueba " + testCaseName + " : "
+ testCaseDesc);
		scriptCode2.append("\n");
		scriptCode2.append("			" + testCaseName + "(driver);");

		return scriptCode2;
	}


	/**
	 * @return código Java de automatización de las pruebas funcionales (parte 3 de 5)
	 */
	public static StringBuffer generateScriptCode3() {

		StringBuffer scriptCode3 = new StringBuffer();

		scriptCode3.append("\n\n");
		scriptCode3.append("			//Se cierran todas las ventanas del navegador abiertas
por el driver");
		scriptCode3.append("\n");
		scriptCode3.append("			driver.quit();");
		scriptCode3.append("\n");
		scriptCode3.append("	}");
		scriptCode3.append("\n\n\n");
		scriptCode3.append("	/**");
		scriptCode3.append("\n");
		scriptCode3.append("	 * Método para establecer las propiedades del driver del
navegador");
		scriptCode3.append("\n");
		scriptCode3.append("	 * @param browser");
		scriptCode3.append("\n");
		scriptCode3.append("	 * @return driver");
		scriptCode3.append("\n");
		scriptCode3.append("	 */");
		scriptCode3.append("\n");
		scriptCode3.append("	static WebDriver getDriver(String browser) {");
		scriptCode3.append("\n\n");
		scriptCode3.append("			//Objeto del driver del navegador");
		scriptCode3.append("\n");
		scriptCode3.append("			WebDriver driver = null;");
		scriptCode3.append("\n\n");
		scriptCode3.append("			//Ruta del navegador");
		scriptCode3.append("\n");
		scriptCode3.append("			String driverPath = \"\";");
		scriptCode3.append("\n\n");
		scriptCode3.append("			//Se establecen las propiedades del driver según el
navegador elegido");
		scriptCode3.append("\n");
```

```
		scriptCode3.append("		switch(browser) {");
		scriptCode3.append("\n");
		scriptCode3.append("			case \"Chrome\":");
		scriptCode3.append("\n");
		scriptCode3.append("				driverPath = \"/eclipse-web-
workspace/WebDrivers/chromedriver.exe\";");
		scriptCode3.append("\n");
		scriptCode3.append("
	System.setProperty(\"webdriver.chrome.driver\", driverPath);");
		scriptCode3.append("\n");
		scriptCode3.append("				driver = new ChromeDriver();");
		scriptCode3.append("\n");
		scriptCode3.append("				break;");
		scriptCode3.append("\n");
		scriptCode3.append("			case \"Firefox\":");
		scriptCode3.append("\n");
		scriptCode3.append("				driverPath = \"/eclipse-web-
workspace/WebDrivers/geckodriver.exe\";");
		scriptCode3.append("\n");
		scriptCode3.append("
	System.setProperty(\"webdriver.gecko.driver\", driverPath);");
		scriptCode3.append("\n");
		scriptCode3.append("				driver = new FirefoxDriver();");
		scriptCode3.append("\n");
		scriptCode3.append("				break;");
		scriptCode3.append("\n");
		scriptCode3.append("			default:");
		scriptCode3.append("\n");
		scriptCode3.append("				System.out.println(\"Navegador no
definindo\");");
		scriptCode3.append("\n");
		scriptCode3.append("		}");
		scriptCode3.append("\n\n");
		scriptCode3.append("		return driver;");
		scriptCode3.append("\n");
		scriptCode3.append("	}");

		return scriptCode3;
	}


	/**
	 * @param testCaseName
	 * @param testCaseDesc
	 * @return código Java de automatización de las pruebas funcionales (parte 4_1 de 5)
	 */
	public static StringBuffer generateScriptCode4_1(String testCaseName, String testCaseDesc) {

		StringBuffer scriptCode4_1 = new StringBuffer();

		scriptCode4_1.append("\n\n\n");
		scriptCode4_1.append("	/**");
		scriptCode4_1.append("\n");
		scriptCode4_1.append("	 * Método para ejecutar el caso de prueba " +  testCaseName +
" : " + testCaseDesc);
		scriptCode4_1.append("\n");
		scriptCode4_1.append("	 * @param driver");
		scriptCode4_1.append("\n");
		scriptCode4_1.append("	 * @throws InterruptedException");
		scriptCode4_1.append("\n");
```

```
                    scriptCode4_1.append("    */");
                    scriptCode4_1.append("\n");
                    scriptCode4_1.append("   static void " + testCaseName + "(WebDriver driver) throws
InterruptedException {");
                    scriptCode4_1.append("\n\n");

                    return scriptCode4_1;
          }


          /**
           * @param stepName
           * @param stepDesc
           * @param guiElementType
           * @param guiElementName
           * @param guiElementValue
           * @return código Java de automatización de las pruebas funcionales (parte 4_2 de 5)
           */
          public static StringBuffer generateScriptCode4_2(String stepName, String stepDesc, String
guiElementType, String guiElementName, String guiElementValue) {

                    StringBuffer scriptCode4_2 = new StringBuffer();

                    scriptCode4_2.append("               //Paso " +  stepName + " : " + stepDesc);
                    scriptCode4_2.append("\n");

                    //Se construye la acción según el tipo de elemento de la GUI
                    switch(guiElementType) {
                              case "URL":
                                        scriptCode4_2.append("               driver.get(\"" + guiElementValue +
"\");");
                                        break;
                              case "TextBox":
                                        scriptCode4_2.append("               driver.findElement(By.name(\"" +
guiElementName + "\")).sendKeys(\"" + guiElementValue + "\");");
                                        break;
                              case "Button":
                                        scriptCode4_2.append("               driver.findElement(By.name(\"" +
guiElementName + "\")).click();");
                                        break;
                              case "RadioButton":
                                        scriptCode4_2.append("       List<WebElement> " + guiElementName +
" = driver.findElements(By.name(\"" + guiElementName + "\"));");
                                        scriptCode4_2.append("\n");
                                        scriptCode4_2.append("       for (int i = 0; i < " + guiElementName +
".size(); i++) {");
                                        scriptCode4_2.append("\n");
                                        scriptCode4_2.append("          if(" + guiElementName +
".get(i).getAttribute(\"value\").equals(\"" + guiElementValue + "\")) {");
                                        scriptCode4_2.append("\n");
                                        scriptCode4_2.append("               " + guiElementName +
".get(i).click();");
                                        scriptCode4_2.append("\n");
                                        scriptCode4_2.append("          }");
                                        scriptCode4_2.append("\n");
                                        scriptCode4_2.append("       }");
                                        break;
                              case "CheckBox":
                                        scriptCode4_2.append("               driver.findElement(By.name(\"" +
guiElementName + "\")).click();");
```

```
                                        break;
                              case "ComboBox":
                                        scriptCode4_2.append("                Select "+ guiElementName + " = new
Select(driver.findElement(By.name(\"" + guiElementName + "\")));");
                                        scriptCode4_2.append("\n");
                                        scriptCode4_2.append("                " + guiElementName +
".selectByValue(\"" + guiElementValue +  "\");");
                                        break;
                              default:
                                        System.out.println("Elemento no definindo");
                    }

                    scriptCode4_2.append("\n\n");

                    return scriptCode4_2;
          }


          /**
           * @param testCaseName
           * @return código Java de automatización de las pruebas funcionales (parte 4_3 de 5)
           */
          public static StringBuffer generateScriptCode4_3(String testCaseName) {

                    StringBuffer scriptCode4_3 = new StringBuffer();

                    scriptCode4_3.append("                System.out.println(\"El caso de prueba " +
testCaseName + " ha terminado correctamente\");");
                    scriptCode4_3.append("\n");
                    scriptCode4_3.append("    }");

                    return scriptCode4_3;
          }


          /**
           * @return código Java de automatización de las pruebas funcionales (parte 5 de 5)
           */
          public static StringBuffer generateScriptCode5() {

                    StringBuffer scriptCode5 = new StringBuffer();

                    scriptCode5.append("\n\n\n");
                    scriptCode5.append("}//Cierre de la clase");

                    return scriptCode5;
          }


          /**
           * Método para generar el archivo de salida
           * @param fileName
           * @param generatedCode
           * @throws IOException
           */
          public static void generateScriptFile(String fileName, StringBuffer generatedCode) throws
IOException {

                    //Se crea el archivo
                    File file = new File(fileName);
```

```
			file.createNewFile();
			System.out.println("Se ha generado el archivo de salida");

			//Se escribe el código generado en el archivo creado
			FileWriter writer = new FileWriter(file);
			writer.write(generatedCode.toString());

			writer.close();
		}


} //Cierre de la clase
```

6.2.4. Código de la clase principal del generador de código

A continuación, se muestra el código de la clase principal del generador de código que contiene el método "*main(Sring[] args)*" (AutoTestingCodeGenerator.java):

```
/**
 * Clase principal del generador de código
 */
public class AutoTestingCodeGenerator {

	/**
	 * Método principal para ejecutar el generador de código
	 * @param args
	 */
	public static void main(String[] args) {

		GeneratorModel model = new GeneratorModel();
		GeneratorView view = new GeneratorView();

		new GeneratorController(view, model);
	}

} //Cierre de la clase
```

6.2.5. Código de la clase de utilidades

Se han desarrollado unas utilidades para obtener información de los ficheros XML de entrada sobre el modelado de las pruebas funcionales. Las utilidades se utilizan para obtener:

- Los archivos XML de entrada parseados.
- Las propiedades de cada nodo del grafo del modelado (identificador, nombre y descripción).
- El tipo de cada nodo del grafo del modelado (Start, Finish, TestCase, Step, URL, TextBox, Button, CheckBox, RadioButton, ComboBox, etc.).

- El identificador del siguiente nodo en un flujo de casos de prueba o en un flujo de pasos de un caso de prueba.
- El identificador del primer paso del caso de prueba.

A continuación, se muestra el código de la clase Java de las utilidades (XMLUtilities.java):

```
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;


public class XMLUtilities {

	/**
	 * @param fileName
	 * @return archivo parseado
	 * @throws ParserConfigurationException
	 * @throws SAXException
	 * @throws IOException
	 */
	Document XMLParser(String fileName) throws ParserConfigurationException, SAXException,
IOException {

		//Parsear el archivo de entrada
		DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
		DocumentBuilder builder = factory.newDocumentBuilder();
		Document doc = builder.parse(fileName);

		return doc;
	}


	/**
	 * @param nodesList
	 * @param nodeType
	 * @param nodeName
	 * @return el identificador del nodo
	 */
	String getNodeID(NodeList nodesList, String nodeType, String nodeName) {

		String nodeID = null;

		for (int i=0; i<nodesList.getLength(); i++) {
			Node graphNode = nodesList.item(i);
			Element graphElement = (Element) graphNode;

			if (getNodeType(graphNode).equals(nodeType) &&
graphElement.getAttribute("name").equals(nodeName)) {
				nodeID = graphElement.getAttribute("xmi:id");
			}
		}
```

```
			return nodeID;
		}


		/**
		 * @param nodesList
		 * @param nodeType
		 * @param nodeID
		 * @return el nombre del nodo
		 */
		String getNodeName(NodeList nodesList, String nodeType, String nodeID) {

			String nodeName = null;

			for (int i=0; i<nodesList.getLength(); i++) {
				Node graphNode = nodesList.item(i);
				Element graphElement = (Element) graphNode;

				if(getNodeType(graphNode).equals(nodeType) &&
graphElement.getAttribute("xmi:type").equals("diagram:DNode") &&
graphElement.getAttribute("xmi:id").equals(nodeID)) {
					nodeName = graphElement.getAttribute("name");
				}
			}

			return nodeName;
		}


		/**
		 * @param nodesList
		 * @param nodeName
		 * @return la descripción del nodo
		 */
		String getNodeDescription(NodeList nodesList, String nodeName) {

			String nodeDescripton = null;

			for (int i=0; i<nodesList.getLength(); i++) {
				Node graphNode = nodesList.item(i);
				Element graphElement = (Element) graphNode;

				if(graphElement.getAttribute("name").equals(nodeName)) {
					nodeDescripton = graphElement.getAttribute("description");
				}
			}

			return nodeDescripton;
		}


		/**
		 * @param graphNode
		 * @return el tipo del nodo
		 */
		String getNodeType(Node graphNode){

			String nodeType = null;
			NodeList childNodes = graphNode.getChildNodes();
```

```
		for (int i=0; i<childNodes.getLength(); i++) {

			if (childNodes.item(i).getNodeName().equals("target")) {
				Node childNode = childNodes.item(i);
				Element childElement = (Element) childNode;
				nodeType = childElement.getAttribute("xmi:type");
				nodeType = nodeType.substring(nodeType.indexOf(":") + 1);
			}

		}

		return nodeType;
	}


	/**
	 * @param nodesList
	 * @param nodeID
	 * @return el tipo del nodo a través del identificador del nodo
	 */
	String getNodeTypeByID(NodeList nodesList, String nodeID) {

		String nodeType = null;

		for (int i=0; i<nodesList.getLength(); i++) {
			Node graphNode = nodesList.item(i);
			Element graphElement = (Element) graphNode;

			if(graphElement.getAttribute("xmi:type").equals("diagram:DNode") &&
graphElement.getAttribute("xmi:id").equals(nodeID)) {
				nodeType = getNodeType(graphNode);
			}
		}

		return nodeType;
	}


	/**
	 * @param nodesList
	 * @param sourceNodeID
	 * @param nodeType
	 * @return el identificador del siguiente nodo
	 */
	String getNextNodeID(NodeList nodesList, String sourceNodeID, String nodeType) {

		String nextNodeID = null;

		for (int i=0; i<nodesList.getLength(); i++) {
			Node graphNode = nodesList.item(i);
			Element graphElement = (Element) graphNode;

			String nodeTypeTarget = getNodeTypeByID(nodesList,
graphElement.getAttribute("targetNode"));

			if (graphElement.getAttribute("xmi:type").equals("diagram:DEdge") &&
graphElement.getAttribute("sourceNode").equals(sourceNodeID) && nodeTypeTarget.equals(nodeType)) {
				nextNodeID = graphElement.getAttribute("targetNode");
			}
```

```
			}

			return nextNodeID;
		}


		/**
		 * @param testCaseList
		 * @param graphObjectsList
		 * @param testCaseName
		 * @return el identificador del primer paso del caso de prueba
		 */
		String getFirstStepID(NodeList testCaseList, NodeList graphObjectsList, String testCaseName) {

			NodeList stepNodes = null;

			//Quitar los nodos hijos "#text"
			testCaseList = removeTextFromChildNodes(testCaseList);

			for (int i=0; i<testCaseList.getLength(); i++) {

				Node testCaseNode = testCaseList.item(i);
				Element testCaseElement = (Element) testCaseNode;

				if(testCaseElement.getAttribute("name").equals(testCaseName)) {
					stepNodes = testCaseNode.getChildNodes();

				}

			}

			//El identificador del primer paso del caso de prueba
			String firstStepID = null;

			for (int i=0; i<stepNodes.getLength(); i++) {

				Node stepNode = stepNodes.item(i);
				Element stepElement = (Element) stepNode;

				String stepID = getNodeID(graphObjectsList, "Step",
stepElement.getAttribute("name"));

				boolean stepHasTarget = false;

				for (int n=0; n<graphObjectsList.getLength(); n++) {

					Node objectNode = graphObjectsList.item(n);
					Element objectElement = (Element) objectNode;

					String sourceNodeType = getNodeTypeByID(graphObjectsList,
objectElement.getAttribute("sourceNode"));
					String targetNodeType = getNodeTypeByID(graphObjectsList,
objectElement.getAttribute("targetNode"));

					if (objectElement.getAttribute("xmi:type").equals("diagram:DEdge") &&
sourceNodeType.equals("Step") && targetNodeType.equals("Step") &&
objectElement.getAttribute("targetNode").equals(stepID)) {
						stepHasTarget = true;
					}
```

```
			}

			if (stepHasTarget == false) {
				firstStepID = stepID;
			}

		}

		return firstStepID;
	}


	/**
	 * @param nodeList
	 * @return lista de nodos sin los nodos hijos "#text"
	 */
	NodeList removeTextFromChildNodes(NodeList nodeList) {

		//Quitar los nodos "#text"
		for (int i=0; i<nodeList.getLength(); i++) {

			Node node = nodeList.item(i);
			NodeList nodeChilds = node.getChildNodes();

			for (int n=0; n<nodeChilds.getLength(); n++) {

				Node childNode = nodeChilds.item(n);

				if (childNode.getNodeName().equals("#text")) {
					nodeList.item(i).removeChild(childNode);
				}
			}
		}

		return nodeList;
	}


}
```

7. Ejemplo de modelado y automatización de las pruebas funcionales

Se ha creado una aplicación web como ejemplo con el objetivo de modelar y automatizar sus pruebas funcionales utilizando la solución propuesta.

7.1. Funcionamiento de la aplicación web de ejemplo

Se ha desarrollado una aplicación web de Gestión de Productos utilizando el lenguaje de programación Java y una base de datos MySQL. Con esta aplicación se puede añadir o eliminar productos en la base de datos. Para acceder a la aplicación, el usuario debe proporcionar su nombre y su contraseña registrados en la base de datos.

El siguiente esquema muestra el flujo de los pasos de los casos de prueba de la aplicación web de Gestión de Productos:

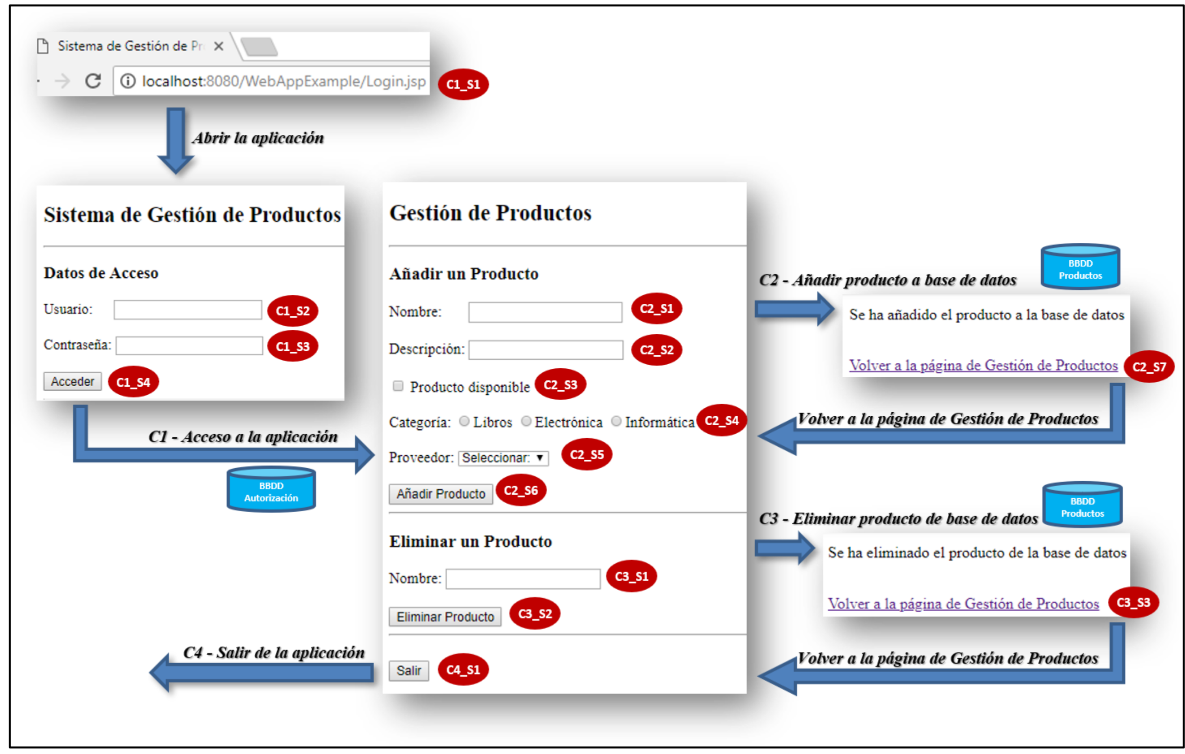

Ilustración 38 - Ejemplo (2) del flujo de los pasos de los casos de prueba de una aplicación web

7.2. Ejemplo de modelado gráfico de una prueba funcional

Vamos a modelar los siguientes casaos de prueba de la aplicación web de Gestión de Productos utilizando el DSL visual:

- Acceso a la aplicación (*login*)
- Alta de un producto en base de datos
- Baja de un producto en base de datos
- Salida de la aplicación (*logout*)

La siguiente tabla contiene el detalle del plan de prueba funcional de la aplicación web de ejemplo:

Casos de prueba		**Pasos de los casos de prueba**	
Código	**Descripción**	**Código**	**Descripción**
C1	Acceso a la aplicación	C1_S1	Abrir la URL de la aplicación con un navegador web
		C1_S2	Escribir el nombre del usuario
		C1_S3	Escribir la contraseña del usuario
		C1_S4	Pulsar el botón "Acceder"
C2	Alta de un producto	C2_S1	Escribir el nombre del producto
		C2_S2	Escribir la descripción del producto
		C2_S3	Marcar si el producto está disponible
		C2_S4	Elegir la categoría del producto
		C2_S5	Seleccionar el proveedor del producto
		C2_S6	Pulsar el botón "Añadir Producto"
		C2_S7	Pulsar sobre el enlace "Volver a la página de Gestión de Productos"
C3	Baja de un producto	C3_S1	Escribir el nombre del producto a eliminar
		C3_S2	Pulsar el botón "Eliminar Producto"

		C3_S3	Pulsar sobre el enlace "Volver a la página de Gestión de Productos"
C4	Salir de la aplicación	C4_S1	Pulsar el botón "Salir"

Tabla 6 - Ejemplo (2) de una prueba funcional

El siguiente esquema muestra el modelado de la prueba funcional de la aplicación web de Gestión de Productos utilizando la herramienta de modelado gráfico creada:

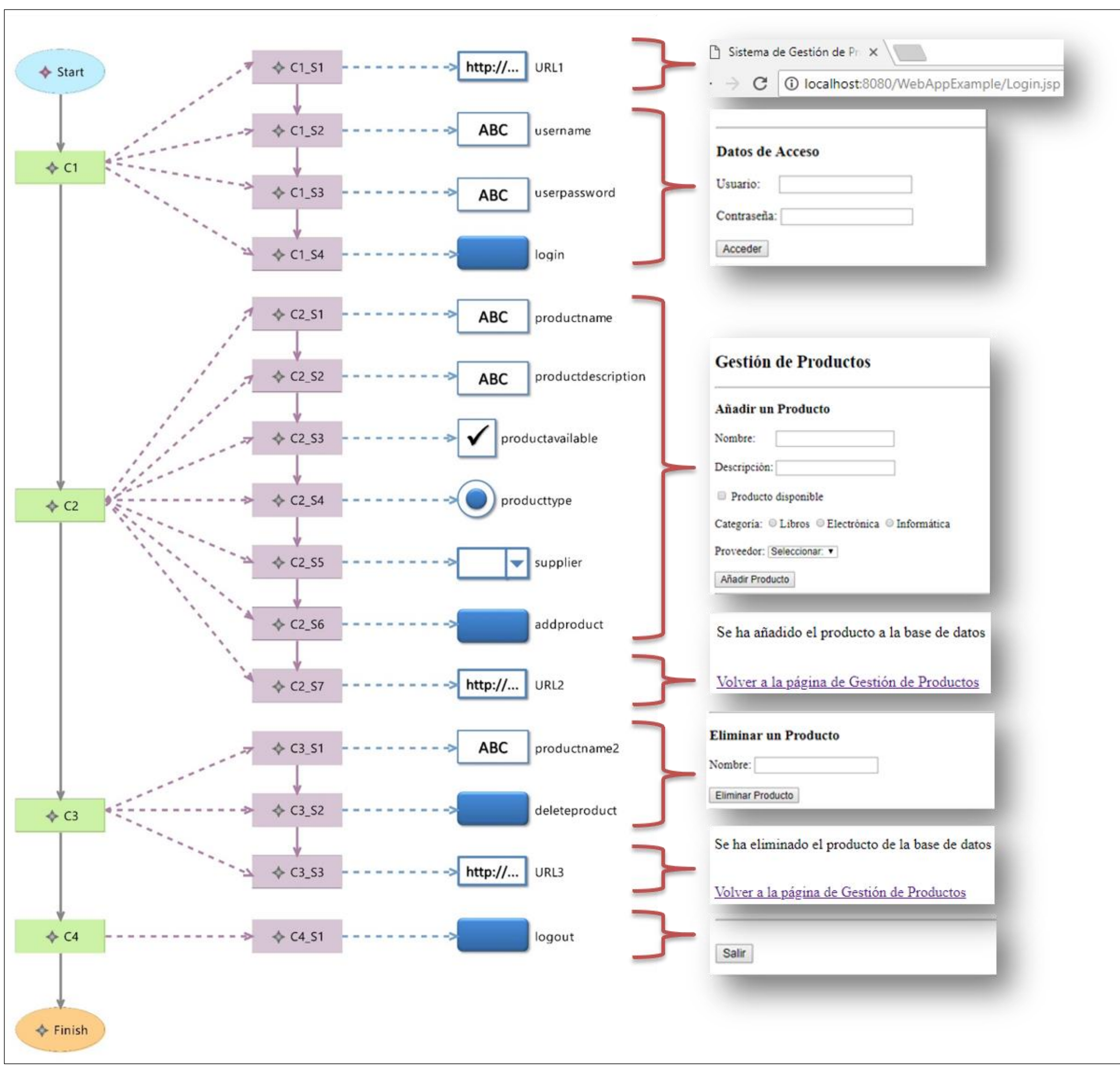

Ilustración 39 - Ejemplo (2) de modelado gráfico de una prueba funcional de una aplicación

7.3. Ejemplo de una generación automática de código de pruebas

Se puede utilizar el generador automático de código desarrollado para generar el código de automatización de las pruebas funcionales de la aplicación web de ejemplo.

El generador automático de código necesita que el usuario proporcione los siguientes datos de entrada a través de la interfaz gráfica y luego pulsar el botón "*Generate*":

- Ruta y nombre del fichero (ProductManagement.java) del código de automatización de la prueba funcional que se va a generar.
- Ruta y nombre del fichero XML (My.testingmodeling) del modelado creado con la herramienta de modelado de las pruebas de funcionales. Este fichero contiene la información sobre las propiedades de cada objeto del modelado como el nombre, la descripción y el valor.

A continuación, se muestra una parte del contenido del fichero XML (My.testingmodeling) del modelado creado con el DSL Visual:

```
<?xml version="1.0" encoding="UTF-8"?>
<testingModeling:TestingModeling xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:testingModeling="http://www.example.org/testingModeling">
  <start startToTestCase="//@testCase.0"/>
  <testCase name="C1" nextTestCase="//@testCase.1" description="C1 - Acceso a la aplicación">
    <testCaseToStep name="C1_S1" nextStep="//@testCase.0/@testCaseToStep.1" description="C1_S1 -
Abrir la URL de la aplicación con un navegador web">
      <stepToURL name="URL1" value="http://localhost:8080/WebAppExample/Login.jsp"/>
    </testCaseToStep>
    <testCaseToStep name="C1_S2" nextStep="//@testCase.0/@testCaseToStep.2" description="C1_S2 -
Escribir el nombre del usuario ">
      <stepToTextBox name="username" value="usuario01"/>
    </testCaseToStep>
    <testCaseToStep name="C1_S3" nextStep="//@testCase.0/@testCaseToStep.3" description="C1_S3 -
Escribir la contraseña del usuario">
      <stepToTextBox name="userpassword" value="password01"/>
    </testCaseToStep>
    <testCaseToStep name="C1_S4" description="C1_S4 - Pulsar el botón &quot;Acceder&quot;">
      <stepToButton name="login"/>
    </testCaseToStep>
  </testCase>
  …
  …
  <testCase name="C4" testCaseToFinish="//@finish" description="C4 - Salir de la aplicación">
    <testCaseToStep name="C4_S1" description="C4_S1 - Pulsar el botón &quot;Salir&quot;">
      <stepToButton name="logout"/>
    </testCaseToStep>
  </testCase>
  <finish/>
</testingModeling:TestingModeling>
```

➢ Ruta y nombre del fichero del grafo (representations.aird) del modelado creado con la herramienta de modelado de las pruebas de funcionales. Este fichero contiene la información sobre los objetos del modelado gráfico y la relación entre ellos.

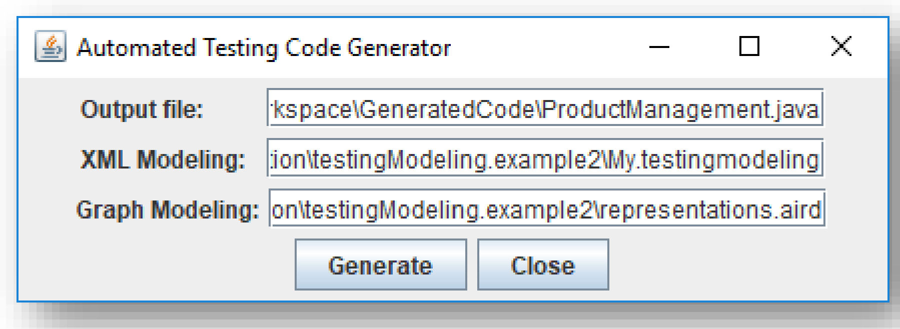

Ilustración 40 - GUI del Generador Automático de Código con los datos de entrada completados

A continuación, se muestra el código fuente generado del script de automatización de la prueba funcional (ProductManagement.java):

```
import java.util.concurrent.TimeUnit;
import java.util.List;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.support.ui.Select;
import org.openqa.selenium.firefox.FirefoxDriver;


/**
 * Clase del script de automatización de las pruebas funcionales
 *
 */
public class ProductManagement {

	/**
	 * Método principal de ejecución de las pruebas funcionales
	 * @param args
	 * @throws Exception
	 */
	public static void main(String[] args) throws Exception {

		//Se obtiene el nombre del navegador
		String browser = args[0];

		//Se crea el objeto del driver del navegador
		WebDriver driver = getDriver(browser);

		//Se define el tiempo máximo de espera
		driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
```

```
		//Se ejecuta el caso de prueba C1 : C1 - Acceso a la aplicación
		C1(driver);

		//Se ejecuta el caso de prueba C2 : C2 - Alta de un producto
		C2(driver);

		//Se ejecuta el caso de prueba C3 : C3 - Baja de un producto
		C3(driver);

		//Se ejecuta el caso de prueba C4 : C4 - Salir de la aplicación
		C4(driver);

		//Se cierran todas las ventanas del navegador abiertas por el driver
		driver.quit();
	}


	/**
	 * Método para establecer las propiedades del driver del navegador
	 * @param browser
	 * @return driver
	 */
	static WebDriver getDriver(String browser) {

		//Objeto del driver del navegador
		WebDriver driver = null;

		//Ruta del navegador
		String driverPath = "";

		//Se establecen las propiedades del driver según el navegador elegido
		switch(browser) {
			case "Chrome":
				driverPath = "/eclipse-web-workspace/WebDrivers/chromedriver.exe";
				System.setProperty("webdriver.chrome.driver", driverPath);
				driver = new ChromeDriver();
				break;
			case "Firefox":
				driverPath = "/eclipse-web-workspace/WebDrivers/geckodriver.exe";
				System.setProperty("webdriver.gecko.driver", driverPath);
				driver = new FirefoxDriver();
				break;
			default:
				System.out.println("Navegador no definindo");
		}

		return driver;
	}


	/**
	 * Método para ejecutar el caso de prueba C1 : C1 - Acceso a la aplicación
	 * @param driver
	 * @throws InterruptedException
	 */
	static void C1(WebDriver driver) throws InterruptedException {

		//Paso C1_S1 : C1_S1 - Abrir la URL de la aplicación con un navegador web
		driver.get("http://localhost:8080/WebAppExample/Login.jsp");
```

```
                //Paso C1_S2 : C1_S2 - Escribir el nombre del usuario
                driver.findElement(By.name("username")).sendKeys("usuario01");

                //Paso C1_S3 : C1_S3 - Escribir la contraseña del usuario
                driver.findElement(By.name("userpassword")).sendKeys("password01");

                //Paso C1_S4 : C1_S4 - Pulsar el botón "Acceder"
                driver.findElement(By.name("login")).click();

                System.out.println("El caso de prueba C1 ha terminado correctamente");
        }


        /**
         * Método para ejecutar el caso de prueba C2 : C2 - Alta de un producto
         * @param driver
         * @throws InterruptedException
         */
        static void C2(WebDriver driver) throws InterruptedException {

                //Paso C2_S1 : C2_S1 - Escribir el nombre del producto
                driver.findElement(By.name("productname")).sendKeys("Producto1");

                //Paso C2_S2 : C2_S2 - Escribir la descripción del producto
                driver.findElement(By.name("productdescription")).sendKeys("Descripción del
Producto1");

                //Paso C2_S3 : C2_S3 - Marcar si el producto está disponible
                driver.findElement(By.name("productavailable")).click();

                //Paso C2_S4 : C2_S4 - Elegir la categoría del producto
      List<WebElement> producttype = driver.findElements(By.name("producttype"));
      for (int i = 0; i < producttype.size(); i++) {
         if(producttype.get(i).getAttribute("value").equals("3")) {
           producttype.get(i).click();
         }
       }

                //Paso C2_S5 : C2_S5 - Seleccionar el proveedor del producto
                Select supplier = new Select(driver.findElement(By.name("supplier")));
                supplier.selectByValue("2");

                //Paso C2_S6 : C2_S6 - Pulsar el botón "Añadir Producto"
                driver.findElement(By.name("addproduct")).click();

                //Paso C2_S7 : C2_S7 - Pulsar sobre el enlace "Volver a la página de Gestión de
Productos"
                driver.get("http://localhost:8080/WebAppExample/ProductAdmin.jsp");

                System.out.println("El caso de prueba C2 ha terminado correctamente");
        }


        /**
         * Método para ejecutar el caso de prueba C3 : C3 - Baja de un producto
         * @param driver
         * @throws InterruptedException
         */
        static void C3(WebDriver driver) throws InterruptedException {
```

```
		//Paso C3_S1 : C3_S1 - Escribir el nombre del producto a eliminar
		driver.findElement(By.name("productname2")).sendKeys("Producto1");

		//Paso C3_S2 : C3_S2 - Pulsar el botón "Eliminar Producto"
		driver.findElement(By.name("deleteproduct")).click();

		//Paso C3_S3 : C3_S3 - Pulsar sobre el enlace "Volver a la página de Gestión de
Productos"
		driver.get("http://localhost:8080/WebAppExample/ProductAdmin.jsp");

		System.out.println("El caso de prueba C3 ha terminado correctamente");
	}


	/**
	 * Método para ejecutar el caso de prueba C4 : C4 - Salir de la aplicación
	 * @param driver
	 * @throws InterruptedException
	 */
	static void C4(WebDriver driver) throws InterruptedException {

		//Paso C4_S1 : C4_S1 - Pulsar el botón "Salir"
		driver.findElement(By.name("logout")).click();

		System.out.println("El caso de prueba C4 ha terminado correctamente");
	}


}//Cierre de la clase
```

7.4. Creación del script de automatización de las pruebas funcionales

Para crear el script de automatización de las pruebas funcionales, se crea un proyecto Java utilizando el código fuente generado “ProductManagement.java” y se añade al proyecto lo siguiente:

- Las librerías de Selenium:
  - *client-combined*
  - *selenium-server-standalone*
- Referencias a los *WebDrivers* que proporcionan la capacidad de navegar de forma automática por las páginas web de la aplicación:
  - *chromedriver* del navegador Chrome
  - *geckodriver* del navegador Firefox

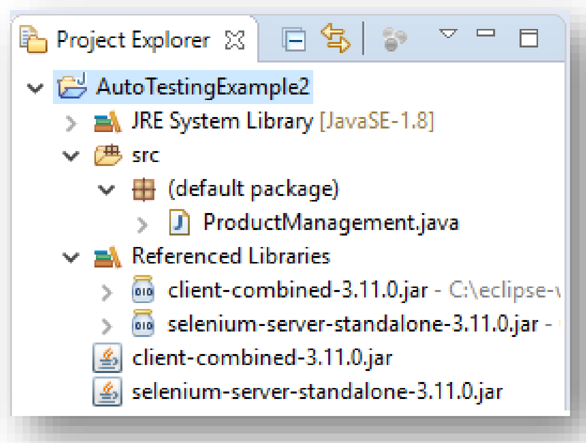

Ilustración 41 - Estructura del proyecto del script de automatización de las pruebas funcionales

Se exportan todos los recursos necesarios del proyecto Java para crear un fichero JAR ejecutable.

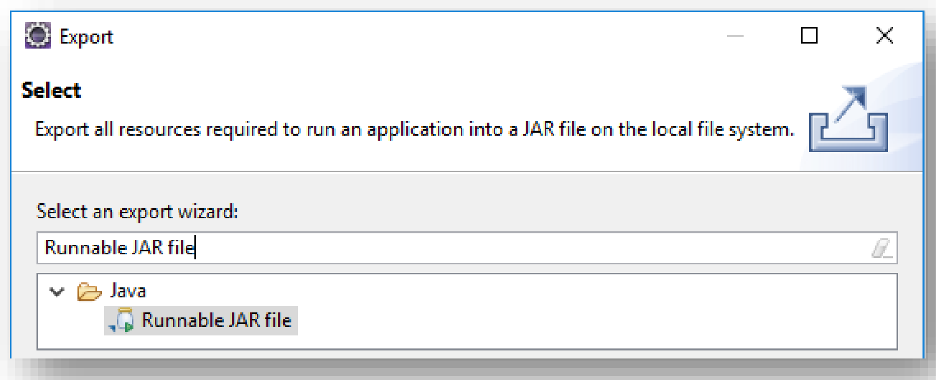

Ilustración 42 - Exportar el proyecto del script de automatización de la prueba funcional a un fichero JAR ejecutable

En el piloto, el script de automatización de las pruebas funcionales, es compatible con los navegadores Chrome y Firefox. Se puede ampliar el número de navegadores añadiendo más referencias a *WebDrivers* de otros navegadores tanto en el Software del generador de código como en el proyecto del Script de automatización de las pruebas.

Al final, se puede ejecutar el script de automatización de las pruebas funcionales utilizando la siguiente instrucción:

```
java -jar [Nombre del Script] [Tipo de Navegador (Chrome, Firefox, …)]
```

Ejemplo de la instrucción de una ejecución de una prueba funcional con el navegador Chrome:

```
java -jar AutoTestingExample2.jar Chrome
```

Ejemplo de la instrucción de una ejecución de una prueba funcional con el navegador Firefox:

```
java -jar AutoTestingExample2.jar Firefox
```

8. Conclusiones

Se ha definido un lenguaje visual específico de dominio para modelar las pruebas funcionales de las aplicaciones y se ha explicado cómo se puede implementar la solución propuesta de forma óptima en el ciclo de vida de desarrollo del Software.

La solución tiene las ventajas que se utiliza un lenguaje de modelado natural, intuitivo, fácil de aprender y se puede usar por los expertos del dominio sin necesidad de tener muchos conocimientos técnicos. El modelado gráfico de las pruebas es fácil de mantener y contiene detalles del funcionamiento de la aplicación tanto a alto nivel como a bajo nivel.

Se han creado pilotos y ejemplos utilizando Software libre para demostrar que la solución puede servir como base para desarrollar herramientas de gestión y automatización de las pruebas funcionales:

- Se ha creado una herramienta de modelado gráfico utilizando el proyecto Sirius de Eclipse para implementar el DSL visual definido.
- Se ha desarrollado un generador automático de código que genera el código de automatización de las pruebas funcionales de las aplicaciones web a partir del modelado creado con el DSL visual. El código de automatización generado tiene las ventajas que es estructurado y tiene información sobre los casos de prueba y los pasos de cada caso de prueba.
- Se ha creado un ejemplo de una aplicación web para modelar y para automatizar sus pruebas funcionales utilizando la solución propuesta.

9. Trabajos Futuros

Como trabajo futuro se propone ampliar el alcance de la solución propuesta de modelado y generación automática de código. El objetivo será analizar y desarrollar los siguientes puntos para mejorar la gestión y la generación de código de automatización de las pruebas funcionales:

- **Mejoras en la gestión de las pruebas funcionales**
    - Incluir más información en el modelado de los casos de prueba:
        - Precondiciones: es necesario informar los requerimientos previos para iniciar la prueba, que pueden ser ciertos valores en base de datos o la existencia de algunos ficheros en el sistema.
        - Resultado esperado: es para contrastar el resultado de ejecución real de la prueba con el resultado de salida esperado de forma manual o automática.
        - Criticidad: informar la criticidad (baja, media, alta) de los casos de prueba facilita el diseño de los planes de prueba de forma óptima.
        - Tiempo de ejecución: la información del tiempo previsto de ejecución del caso de prueba es importante para calcular el tiempo total necesario para realizar los planes de prueba.
        - Requisitos funcionales: relacionar los casos de prueba con los requisitos funcionales permite verificar la cobertura funcional de los planes de prueba.
    - En el DSL visual, añadir más tipos de los elementos de la interfaz gráfica de usuario de las aplicaciones como listas y menús.

- **Mejoras en la generación de código de automatización de las pruebas funcionales**
    - Creación de pilotos complejos de generadores de código de automatización de pruebas compatibles con más tipos de los elementos de la interfaz gráfica de usuario de las aplicaciones y con más tipos de navegadores (ej. Internet Explorer).
    - Diseño y desarrollo de generadores de código de automatización de las pruebas para aplicaciones no basadas en web.

10. Referencias

1. TestLink, http://testlink.org
2. Quality Center, https://software.microfocus.com/es-es/software/quality-center
3. Selenium & Selenium IDE, https://www.seleniumhq.org
4. Unified Functional Testing (UFT), https://software.microfocus.com/en-us/products/unified-functional-automated-testing/overview
5. Unified Functional Testing (UFT), Product Availability Matrix, https://software.microfocus.com/sites/default/files/resources/files/uft_14.xx_pam.pdf
6. Eggplant Functional, https://www.testplant.com/products/eggplant-functional
7. GMF, https://wiki.eclipse.org/Graphical_Modeling_Framework
8. Introducing the GMF Runtime, http://www.eclipse.org/articles/Article-Introducing-GMF/article.html
9. Sirius, https://www.eclipse.org/sirius
10. Domain-Specific Modeling: Enabling Full Code Generation, Steven Kelly and Juha-Pekka Tolvanen, 2008
11. Git, https://git-scm.com
12. Jenkins, https://jenkins.io
13. Eclipse Modeling Tools, http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/oxygen2
14. Firefox driver v0.20.0, https://github.com/mozilla/geckodriver/releases
15. ChromeDriver 2.36, WebDriver for Chrome, https://sites.google.com/a/chromium.org/chromedriver
16. MySQL Installer, https://dev.mysql.com/downloads/installer

11. Términos

Término	Descripción
DSL	Lenguaje Específico de Dominio
DSM	Modelado Específico de Dominio
EMF	Entorno de Modelado de Eclipse
GEF	Entorno de Edición Gráfica de Eclipse
GMF	Entorno de Modelado Gráfico de Eclipse
GUI	Interfaz Gráfica de Usuario
LDAP	Protocolo ligero de acceso a directorios
MVC	Patrón de arquitectura de Software: Modelo-Vista-Controlador