UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Proyecto Final de Carrera

Ingeniería Informática

Incorporación de pruebas automáticas sobre transformaciones de modelos en el entorno de integración continua de MOSKitt

Código:

DSIC-145

Autor:

Jesús Badenas Martínez

Director del Proyecto:

Vicente Pelechano Ferragud

RESUMEN

Una de las funcionalidades que ofrece la herramienta *MOSKitt* es la transformación de modelos, una parte clave en la *Ingeniería Dirigida por Modelos* (MDE). La Transformación de Modelos se define como la generación automática de un modelo a partir de otro, de acuerdo con unas reglas establecidas de transformación.

Como todo fragmento de software, en la implementación de estas transformaciones pueden producirse errores, de forma que, aunque la transformación se haya llevado a cabo de forma correcta, el resultado no sea el esperado. Es por tanto necesario el definir los casos de prueba necesarios para minimizar los errores. El mejor método de detectar dichos fallos y no desperdiciar tiempo es que estos casos de prueba puedan ejecutarse de la forma más automática posible una vez construida la transformación.

La *integración continua* es un concepto que surge a partir de la idea de realización de construcciones de software diarias. El modelo ideal permite que la construcción y ejecución de las pruebas sea realizada cada vez que el código cambia o es enviado al software de control de versiones.

Así pues, la idea principal de este proyecto es la incorporación de una serie de pruebas automáticas en el proceso de integración continua de la herramienta *MOSKitt*, de tal modo que se asegure que se está construyendo de un modo fiable y funcionalmente correcto.

Índice de contenido

1. Introducción

1.1. Ámbito

El trabajo realizado en este proyecto forma parte de **MOSKitt** (*Modeling Software KITT*), una herramienta CASE *Open Source* gratuita, desarrollada por la *Consellería de Infraestructuras y Transporte* (CIT). Está basada en *Eclipse* y su fin es dar soporte a la metodología *gvMétrica* (una adaptación de *Métrica III*). Entre otras funcionalidades, nos permite definir y transformar modelos.

Basada en una arquitectura de *plugins*, se llevan a cabo continuas iteraciones para ir obteniendo versiones estables/correctas de la herramienta, añadiendo nuevas funcionalidades y solventando errores existentes al final de cada una de ellas.

1.2. Motivación

Debido a la continua modificación del código, el avance y desarrollo de nuevas funcionalidades y la solución de errores sobre lo que ya se encontraba desarrollado, surge la necesidad de corregir versiones anteriores de nuestra herramienta, y esto nos obliga a realizar continuos test de regresión para comprobar que todo sigue funcionando correctamente.

Esta necesidad de *testing*, repetida una y otra vez, supone una gran inversión de tiempo que afecta a la productividad, reduciendo considerablemente el período dedicado al desarrollo del proyecto. Bajo estas circunstancias surge la idea de la automatización de las pruebas, un método que permitirá reducir considerablemente el tiempo dedicado a comprobar tanto que las nuevas funcionalidades dan los resultados esperados, como que las anteriores siguen trabajando de forma correcta.

En el contexto de *MOSKitt* se trabaja con modelos y metamodelos Se realizan transformaciones de modelo a modelo o de modelo a texto, y se genera código de forma automática. Realizar el proceso de pruebas sobre el código final conlleva un gran esfuerzo.

Así pues, la automatización de las pruebas sobre transformaciones entre modelos se empezó a cuestionar en el proyecto. Sin embargo, pese a darse una primera solución donde los test se lanzaban a través de la plataforma de desarrollo mediante la ejecución de una suite de test *JUnit;* seguía siendo un proceso independiente de la construcción de los componentes de las transformaciones.

La posibilidad de incluir la ejecución de esta serie de pruebas en el mismo proceso de construcción supondría un gran avance y más comodidad a la hora de probar el correcto funcionamiento de las nuevas funcionalidades.

1.3. Objetivo

Con el objetivo de mejorar la automatización de las pruebas surge este proyecto, que está centrado en la inclusión del *testing* de transformaciones modelo a modelo en el proceso de construcción del software.

Además, podemos concretar que el propósito es completar el entorno de integración de *MOSKitt*, incorporando en él una serie de test que realicen de modo automático las comprobaciones necesarias para verificar que el componente a construir esté correctamente implementado.

Por componente a construir nos referimos a un módulo de la herramienta *MOSKitt* que lo complementa en su funcionalidad. En nuestro caso sería una transformación modelo a modelo.

Se ha utilizado (y mejorado) la infraestructura que permite la definición y ejecución de casos de prueba para transformaciones modelo a modelo que había desarrollada. En concreto, se van a realizar test sobre una transformación presente en la herramienta: *Sketcher2UIM*.

Así pues, el trabajo referente a la automatización de pruebas va a consistir en la definición de un documento en el que se especifiquen un conjunto de casos de prueba que sirvan para hacer un completo testeo del correcto funcionamiento de la transformación *Sketcher2UIM,* y el diseño de los propios casos de prueba en forma de modelos de entrada y test que validen los modelos de salida.

Todo esto se debe incorporar en el proceso de integración de *MOSKitt*, de tal modo que si existe algún error sirva como referencia para comprobar que la construcción no ha sido correcta y se puedan solventar los errores y relanzar el proceso.

En conclusión, el proyecto propuesto tiene como objetivo mejorar la calidad de la herramienta *MOSKitt*, al facilitar no sólo la realización de pruebas de modo automático (con la consiguiente mejora en el tiempo de la producción de sus componentes), sino además garantizar la corrección de los productos obtenidos por el proceso de construcción, gracias a la integración de las pruebas en el mismo.

1.4. Palabras clave

- *Automated Build:* Hace referencia al proceso automático de construcción de *Eclipse*. Es un modo de compilar o construir una serie de features y/o plugins para que puedan pasar a formar parte de la plataforma.

- *Eclipse Modeling Framework (EMF)*: Es un *framework* de modelado y facilidad de generación de código para construir herramientas y otras aplicaciones basadas en un modelo de datos estructurado.

- *gvMétrica*: Es una metodología de desarrollo de software dirigido por modelos definida por la *Consellería de Infraestructura y Transportes* con el objetivo de agilizar sus proyectos de desarrollo de software. Es una adaptación de la metodología *Métrica III*.

- *Headless*: En el contexto de un entorno de desarrollo como *Eclipse*, una ejecución en modo *headless* se entiende como una ejecución de la plataforma sin necesidad de levantar la interfaz gráfica que lleva asociada.

- *Ingeniería dirigida por modelos (MDE)*: Se trata de una metodología de desarrollo de software que se centra en la creación y explotación de modelos de dominio. Tiene por objetivo aumentar la productividad mediante la maximización de la compatibilidad entre los sistemas, lo que simplifica el proceso de diseño y promueve la comunicación entre los usuarios y equipos que trabajan en el sistema.

- *Integración continua*: Es una metodología informática que consiste en hacer integraciones automáticas de un proyecto, lo más a menudo posible para así poder detectar fallos cuanto antes. Entendemos por integración la compilación (y ejecución de test) de todo un proyecto.

- *Metamodeling*: Es la construcción de una colección de “conceptos” dentro de un determinado dominio. Un modelo es una abstracción de los fenómenos en el mundo real; un metamodelo es otra abstracción que destaca las propiedades del propio modelo. En definitiva, un metamodelo es un modelo cuyas instancias son modelos.

- *MOSKitt*: Herramienta *CASE* libre, basada en *Eclipse*. Se utiliza para dar soporte a la metodología gvMétrica.

- *Object Management Group (OMG)*: Es un consorcio dedicado al establecimiento de diversos estándares de tecnologías orientadas a objetos, tales como *UML* o *XMI*. Es una organización sin ánimo de lucro que promueve el uso de tecnología orientada a objetos mediante guías y especificaciones para las mismas.

- *Plug-in*: Es una aplicación que se relaciona con otra para aportarle una función nueva y generalmente muy específica. Esta aplicación adicional es ejecutada por la principal e interactúan por medio de la *API*.

- *Repositorio p2*: Se trata de un directorio que incluye una serie de ficheros *.jar* obtenidos a través de la compilación de una serie de ficheros *Java*. Además, dispone de una serie de *meta-información* sobre los elementos que contiene que facilita su posterior instalación en una herramienta basada en *Eclipse*.

- *Uniform Resource Identifier (URI)*: Es una cadena corta de caracteres que identifica inequívocamente un recurso (en este caso un modelo).

1.5. Planificación

El proyecto ha tenido siete meses de duración, comprendiendo varias fases:

1.5.1. Aprendizaje

En esta fase se ha dedicado un buen porcentaje de tiempo, ya que hay que adaptarse al modo de trabajo, conocer la herramienta que se utiliza, aprender las tecnologías empleadas y “trastear” un poco con ellas.

Incluye el conocimiento del método que ya existía sobre el lanzamiento de pruebas automáticas, así como el consecuente aprendizaje de los lenguajes específicos para la creación de las mismas.

Tiempo total: 1'5 meses.

1.5.2. Adaptación

Para tener cierta soltura con la automatización de pruebas, se ha decidido revisar y actualizar los casos de prueba que ya estaban creados para conocer más los lenguajes y el entorno de ejecución de las pruebas.

En concreto, se han actualizado las pruebas sobre las transformaciones: *UML2DB* (UML a base de datos), *DD2CLD* (diccionario de datos a diagrama de clases) y *BPMN2UCD* (*business process modeling notation* a casos de uso).

Tiempo total: 0'5 meses.

1.5.3. Investigación

Una vez conocido el sistema de automatización de las pruebas, se ha dedicado tiempo a intentar mejorarlo para mejorar la ubicación de los modelos de entrada y se ha estudiado la posibilidad de poder ejecutarlo en modo '*headless*'; sin la necesidad de que se necesite una segunda instancia de la plataforma para lanzarlos.

La mayor parte del tiempo dedicado al proyecto ha sido empleado en investigar sobre el proceso automático de construcción de *Eclipse*. Para poder incluir la ejecución de

una suite de test en la integración de *MOSKitt*, ha sido necesario conocer el proceso completo de construcción, todos los *scripts* que lo conforman y cómo funcionan.

Tras haber adquirido esos conocimientos, se han averiguado diferentes métodos de inclusión de los test, finalmente aceptando la ejecución de una aplicación de *Eclipse* en modo *headless* que permite lanzar una serie de test *JUnit*.

Tiempo total: 2'5 meses.

1.5.4. Desarrollo

La primera tarea en la fase de desarrollo fue la creación de los casos de prueba para la transformación *Sketcher2UIM*. Lo normal es disponer de unos documentos con el diseño de las pruebas, pero esta transformación ha sido creada recientemente y no se disponía de ninguna prueba.

Así pues se ha procedido al diseño del documento del conjunto de pruebas, así como al propio desarrollo de las mismas. Se han creado los modelos de entrada, los ficheros de validación de los resultados y el conjunto de test que ejecutan las pruebas.

En segundo lugar, al disponer de las pruebas ya diseñadas y construidas, se ha procedido a configurar el entorno donde se construirá el componente que pertenece a la transformación mencionada y al cual se le asociarán los test.

El último paso ha sido emplear todos los conocimientos adquiridos en la fase de investigación sobre la integración de los test en el proceso de construcción.

Tiempo total: 2 meses.

1.5.5. Resultados

Una vez todo desarrollado, se han ido observando los resultados producidos a la vez que retocando pequeños detalles para perfeccionar el proceso.

Tiempo total: 0'5 meses.

1.6. Estructura del documento

En el próximo apartado indicaremos el marco conceptual que envuelve al proyecto, examinando la actualidad en lo que concierne al testeo de software, la transformación de modelos y labores actuales de pruebas de transformaciones. A su vez, comentaremos cómo funciona el proceso automático de construcción de *Eclipse* y las técnicas actuales para incorporar test automáticos a dicho proceso.

En la sección 3 analizaremos las distintas tecnologías utilizadas en este proyecto, mencionando qué son y cómo las hemos utilizado, haciendo una breve descripción de su funcionamiento y acompañándolas de algunos ejemplos.

En la sección 4 encontraremos el apartado de análisis del proyecto. Aquí se realiza una descripción detallada de la solución, paso a paso, sin hacer referencia concreta a la implementación de la misma. Se describe el *qué*.

En la sección 5 nos centraremos en el diseño del proyecto. Detallaremos uno a uno todos los casos de prueba, así como la implementación de todo el desarrollo realizado. Aquí se describe el *cómo*.

En la sección 6 observaremos un ejemplo de ejecución del proceso completo, desde los datos de entrada hasta los resultados obtenidos.

Finalmente encontramos dos secciones que contienen las conclusiones obtenidas y las referencias que han servido de gran utilidad para la elaboración de este proyecto.

2. Marco conceptual

2.1. Software testing

Las pruebas de software (*testing*), son los procesos que permiten verificar y revelar la calidad de un producto software. Son utilizadas para identificar posibles fallos de implementación, calidad o usabilidad de un programa. Básicamente es una fase en el desarrollo de software consistente en probar las aplicaciones construidas.

El *testing* se puede entender también como la ejecución de una aplicación concreta bajo una serie de condiciones o valores de entrada específicos, junto con la comprobación de que los resultados sean los que se esperaban. Cada una de las pruebas que se ejecutan se le denomina caso de prueba.

Así pues, un caso de prueba es un conjunto de condiciones o variables bajo las cuales se determina si el requisito de una aplicación es parcial o completamente satisfactorio.

El desarrollo guiado por pruebas, o *Test-driven development* (*TDD*) es una técnica de programación que implica escribir en primer lugar las pruebas y verificar que fallen, luego se implementa el código hasta que las pruebas pasen satisfactoriamente y seguidamente se refactoriza el código escrito. La idea es que los requerimientos sean traducidos a pruebas, de este modo, cuando las pruebas pasen se garantizará que los requerimientos se hayan implementado correctamente.

Actualmente existe una gran variedad de herramientas que permiten realizar de forma automática pruebas de funcionalidad sobre software, entre las cuales podemos destacar *JUnit*.

2.2. Transformación de modelos

La transformación de modelos es una parte muy importante de la *Ingeniería Dirigida por Modelos (MDE)*. Consiste en la generación automática de un modelo a partir de otro, de acuerdo a una serie de reglas de transformación. Estas reglas establecen cómo se transforman los elementos del modelo fuente al destino.

En la **figura 1** podemos observar un pequeño esquema del proceso de una transformación de modelo a modelo.

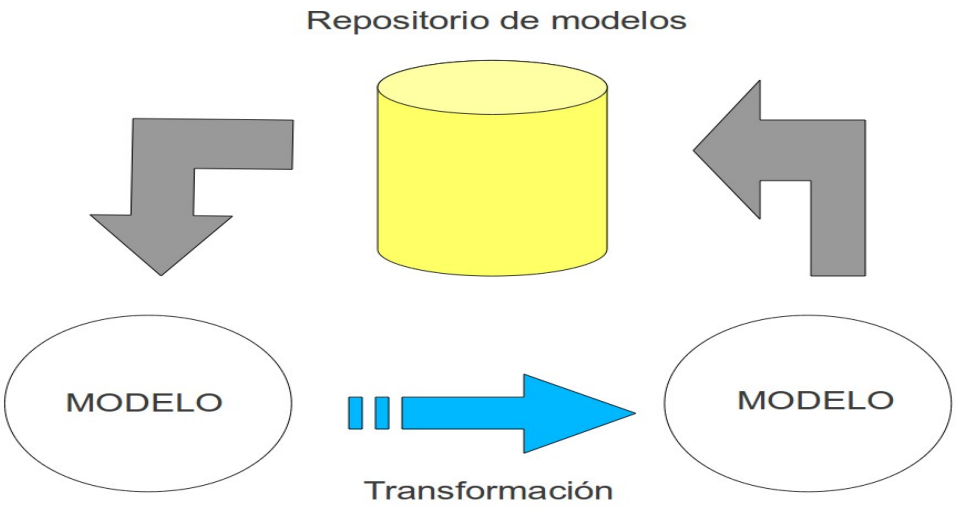


Figura 1. Transformación de modelos

Una transformación también puede tener una serie de modelos de entrada así como una lista de modelos de salida.

Existen varios tipos de transformaciones:

- *Model Merging: v*arios modelos se mezclan en uno. Mismo metamodelo.
- *Model Modification: s*e añaden elementos a los modelos. Mismo metamodelo.
- *Model Transformation: u*n modelo se transforma en otro diferente. Diferentes metamodelos.
- *Model Weaving/Linking: l*os elementos de dos o más modelos se correlacionan.
- *Model Marking: a*ñade etiquetas con información.

2.3. Testeo de transformaciones

Como se ha comentado anteriormente, las transformaciones de modelos son una parte muy destacable en la *Ingeniería Dirigida por Modelos*, y aunque éstas se realicen de modo automático, siguen estando definidas por las personas, lo que implica que éstas puedan contener posibles errores.

Una transformación puede llevarse a cabo, es decir, puede realizarse sin generar ningún error durante su ejecución aunque esto no quiere decir que el resultado sea el correcto o el esperado. Si estos errores no son detectados, todos los modelos que resulten de las transformaciones estarían generándose erróneamente y a medida que

avanza el tiempo del proyecto cada vez es más complejo solucionar este tipo de errores.

Aunque existan formas de realizar pruebas sobre transformaciones para comprobar su correcta aplicación, no es un campo en el que se haya investigado mucho.

El principal problema es que los metamodelos están sometidos a constantes cambios, lo que obliga a que los modelos de prueba diseñados tengan que ser modificados a su vez, y esto no es una tarea trivial.

Actualmente, en nuestro ambiente de trabajo, las pruebas de las transformaciones se realizan de modo semi-automático. Se diseñan una serie de pruebas que consisten en una lista de modelos de entrada que se corresponden con una lista de modelos de salida. Se lanzan las transformaciones y posteriormente se comparan los resultados con los esperados.

Gracias al esfuerzo de un compañero de *MOSKitt*, gran parte de este automatismo se encontraba realizado, por lo que sólo ha sido necesario mejorar ciertos aspectos en lo que concierne a este apartado en concreto.

2.4. Automated Build

Cuando hablamos de construcción automática nos referimos al proceso mediante el cual se origina un producto software a través de una serie de *scripts* o tareas que se ejecutan de modo automático; es decir, sin la necesidad de realizar manualmente una compilación y empaquetado del producto, si no que el proceso es lanzado una sola vez y como resultado obtenemos el producto listo para su uso.

En este proyecto nos vamos a centrar en el proceso de construcción automática que dispone *Eclipse*, ya que es la herramienta sobre la que se asienta todo el montaje.

Como todo proceso de construcción, se siguen una serie de pasos para determinar el software a elaborar:

1. Descarga del código fuente
2. Compilación
3. Ejecución de test
4. Empaquetado

El primero de los pasos consiste en la obtención de todo el código fuente que deberá ser posteriormente compilado y ensamblado. Normalmente lo encontramos en repositorios de tipo *SVN* o *CVS*, los cuales disponen de comandos personalizados que facilitan la descarga de todos ellos.

Una vez obtenidos los fuentes, se procede a compilarlos. Así pues, se generarán una serie de ficheros ejecutables que conforman el producto.

Si no ha habido ningún problema en el proceso de compilación, puede que lo haya en su ejecución. Para ello se definen (opcionalmente) una serie de test o pruebas que el producto debe ejecutar para garantizar su correcto funcionamiento.

Si todos los test son superados, el último de los pasos consiste en el empaquetado (normalmente compresión) del producto para facilitar su distribución.

Estos son los pasos que se siguen en el proceso de construcción de *Eclipse* y, por tanto, en el que utilizamos para generar *MOSKitt* y/o cualquiera de sus componentes.

2.5. Automated Tests

La automatización de los test consiste en el uso de software para el control de la ejecución de los test, la comparación de los resultados obtenidos con los esperados, la configuración de precondiciones, generación de informes...

En el proceso de construcción descrito anteriormente existe la posibilidad de incluir un subproceso que se encargue de la ejecución de una serie de test que validen el producto que se está construyendo. Esto resulta una técnica muy útil para conocer de antemano que nuestra aplicación funcionará correctamente y evitará el tener que configurar un entorno de pruebas específico sólo para los test.

Actualmente existen bastantes herramientas especializadas en la realización de test automáticos, pero esto implica disponer físicamente de la herramienta, por lo que la opción de incluirlo en el proceso de construcción es irreproducible.

La idea de nuestro proyecto es aprovechar el montaje de ejecución de test mediante *Eclipse* e integrarlo en el proceso de construcción, permitiendo así obtener de una sentada el producto *testeado* y, por consiguiente, funcionalmente correcto.

En la siguiente sección obtendremos una visión de las tecnologías más importantes que se han utilizado en el desarrollo del proyecto, a través de las cuales se ha podido realizar el montaje del automatismo de los test y su integración en el proceso de construcción.

3. Tecnologías utilizadas

En este proyecto se han empleado una serie de tecnologías, cada una de ellas para realizar una función en concreto. En este apartado las analizaremos, acompañándolas con algunos ejemplos.

3.1. Eclipse

Es un entorno de desarrollo integrado de código abierto multiplataforma para desarrollar "aplicaciones de cliente enriquecido". Típicamente ha sido utilizada para crear entornos de desarrollo integrados (*IDE*).

La base para *Eclipse* es la "plataforma de cliente enriquecido" (*Rich Client Platform*, RCP). Los siguientes componentes la constituyen:

- Plataforma principal - inicio de *Eclipse*, ejecución de *plugins*
- *OSGi* - una plataforma para *bundling* estándar
- *Standard Widget Toolkit* (*SWT*) – Un *widget toolkit* portable
- *JFace* - manejo de archivos, manejo de texto, editores de texto
- *Workbench* - vistas, editores, perspectivas, asistentes

El entorno de desarrollo integrado (*IDE*) de *Eclipse* emplea módulos (*plugins*) para proporcionar toda su funcionalidad al frente de la plataforma de cliente enriquecido, a diferencia de otros entornos monolíticos donde las funcionalidades están todas incluidas, las necesite el usuario o no. Este mecanismo de módulos es una plataforma ligera para componentes de software.

Eclipse es la base sobre la que se asienta *MOSKitt*. Es más, se puede afirmar que *MOSKitt* es un conjunto de *plugins* desarrollados con *Eclipse* y posteriormente integrados en una plataforma minimalista de *Eclipse*.

3.2. HUTN

Es un lenguaje que define una sintaxis concreta para la construcción de instancias de metamodelos. Estas instancias son modelos con las características deseadas. El motivo de utilizar este tipo de lenguaje para definir los modelos se explicará más adelante. Se ha utilizado la implementación que se proporciona en el proyecto *Epsilon*

Human-Usable Textual Notation (*HUTN*) es una especificación de la *OMG* que se enmarca en la arquitectura dirigida por modelos. La especificación de *HUTN* nos proporciona tres beneficios principales:

- Genérico, es decir, se puede utilizar con cualquier modelo *MOF*
- Completamente automatizado
- *Human-Usable*: diseñado para ajustarse al criterio de uso de las personas

Un fichero *HUTN* se puede separar en 2 partes:

- Referencia al metamodelo: Es como un encabezamiento. En él se especifica el identificador del metamodelo, así como su situación en el proyecto.

- Instancia de los elementos del modelo: Aquí se proporcionan los valores a la instancia del modelo. Se especifican los elementos que se quiere que contenga el modelo, asignando también valores a sus propiedades correspondientes.

Referencia al metamodelo

```
@Spec {
        metamodel "nameM" {
                nsUri: "uriMetamodel"
        }
        ...
}
```

donde

- ***nameM***: nombre que se le quiera dar al metamodelo instanciado. Puede ser cualquier nombre, aunque se recomienda que haga referencia al nombre del metamodelo.

- ***uriMetamodel***: la *URI* del metamodelo referenciado. La podemos conocer si navegamos hasta donde se encuentre el metamodelo y vemos sus propiedades. Se trata de un campo obligatorio.

Se puede hacer referencia a más de un metamodelo, simplemente añadiendo más elementos *metamodel* debajo. Esto es por si necesitamos referenciar elementos de metamodelos distintos en nuestro modelo.

Instancia de los elementos del modelo

```
namePackage {
        nameElement "nameE" {
                [propertyName: value]
                ...
        }
        ...
}
```

donde

- ***namePackage***: nombre del paquete que queremos utilizar del metamodelo.
- ***nameElement***: nombre del elemento del paquete que queramos instanciar.
- ***nameE***: identificador del elemento.
- ***propertyName***: nombre de la propiedad del elemento a la que se le quiera dar un valor distinto al de por defecto. El valor se especificará en *value*.

Una vez definido el código en *HUTN*, se genera un modelo con la extensión *".model"*. Pese a tener esta extensión, se puede observar que en el fondo se trata de un fichero *XML* con otra extensión. Al guardar el fichero se generará automáticamente el modelo.

Como se ha comentado anteriormente, una de las ventajas de emplear este lenguaje es que está completamente integrado en *Eclipse*, a través del proyecto *Epsilon*, que incluye todos los *plugins* necesarios.

A continuación podemos observar un ejemplo de código escrito con la sintaxis de *HUTN*. Se pueden apreciar claramente las 2 partes; una cabecera donde se establece la ubicación del metamodelo, y el bloque de código donde se instancian los elementos del modelo:

(El código se ha simplificado por cuestión de espacio)

```
@Spec {
	metamodel "0.1.0" {
		nsUri: "http:///es.cv.gvcase.mdt.sketcher/0.1.0"
	}
}

sketcher {
	Sketcher "S1" {
		id: "S1"
		elements: Window "MenuPrincipal" {
			id: "MenuPrincipal"
			widgets: TreeViewMenu "TreeViewMenuAAdministracionSistema" {
				id: "TreeViewMenuAAdministracionSistema"
				items: MenuItemAction "ManualGuiaDeEstilo" {
					id: "ManualGuiaDeEstilo"
					text: "Manual Guía de Estilo"
				}
			},
			TreeViewMenu "TreeViewMenuAHerramientasAuxiliares" {
				id: "TreeViewMenuAHerramientasAuxiliares"
				items: MenuItemAction "DebugIgep" {
					id: "DebugIgep"
					text: "Debug igep"
				},
				MenuItemAction "LimpiarPlantillas" {
					id: "LimpiarPlantillas"
					text: "Limpiar plantillas"
				},
				...
			},
			...
			Panel "PanelSuperiorMenuPrincipal" {
				id: "PanelSuperiorMenuPrincipal"
				widgets: IconButton "IconButtonAplicacionSalir" {
					id: "IconButtonAplicacionSalir"
				},
				Label "LabelUsuario" {
					id: "LabelUsuario"
					text: "Usuario"
				}
			}
		}
	}
}
```

Ejemplo de sintaxis HUTN

3.3. EOL

Es un lenguaje de programación imperativo para crear, consultar y modificar modelos *EMF*. Se puede entender como una mezcla entre *Javascript* y *OCL*, combinando lo mejor de ambos. También forma parte del proyecto *Epsilon*.

Está compuesto por módulos, y cada uno de ellos define un cuerpo (*body*) y unas operaciones. El *body* es una secuencia de instrucciones y las operaciones definen un nombre, el tipo de datos sobre el que se pueden ejecutar, parámetros y (opcionalmente) un valor de retorno.

Este lenguaje incorpora un conjunto de métodos y tipos predefinidos; además de los 4 tipos de datos que se proporcionan (*int*, *float*, *string* y *bool*), también existen las colecciones:

- *Bag*: elementos no ordenados y que pueden repetirse
- *Sequence*: elementos ordenados y que pueden repetirse
- *Set*: elementos no ordenados y que no pueden repetirse
- *OrderedSet*: elementos ordenados y que no pueden repetirse

A continuación podemos observar un ejemplo de la sintaxis de *EOL*:

```
self.select(p: PatternIU | p.name = '1C1EI
TabularRegistro').first().interactionUnits.exists(i: InformationIU |
i.name = 'Ventana Consulta')
```

Esta instrucción está evaluada sobre instancias de tipo *PatternIU*, donde *self* indica la propia instancia. Dentro de los *PatternIU* existentes, se está seleccionando aquel cuyo nombre es "*1C1EI TabularRegistro*" (el método *first()* elige el primero de la lista que devuelve la función select(), aunque sólo exista uno). *interactionUnits* es una colección de elementos contenida dentro del *PatternIU* seleccionado, en la cual se está comprobando que exista un *InformationIU* con el nombre *"Ventana Consulta"*.

En el proyecto ha sido empleado, junto con *EVL*, para definir las restricciones que deben cumplir los modelos de salida una vez han sido transformados.

3.4. EVL

Como bien dice su nombre, *Epsilon Validation Language*, se trata de un lenguaje de validación de modelos que forma parte de *EOL*, y por tanto, del proyecto *Epsilon*.

En su forma más simple, las restricciones definidas mediante *EVL* son bastante similares a las de *OCL*. Sin embargo, aporta una serie de mejoras a tener en cuenta:

- Soporta dependencias entre restricciones (por ejemplo: si una restricción *A* falla → no se evalúa la restricción *B*).
- Permite personalizar los mensajes de error que se muestran al usuario.
- Especificación de soluciones que los usuarios pueden invocar para reparar las incoherencias.
- Como está basado en *EOL*, se puede evaluar las restricciones entre modelos (a diferencia de *OCL*).
- Permite diferenciar entre restricciones y avisos, haciendo que la validación falle en los primeros pero sólo muestre un mensaje de advertencia en los segundos; indicando que, aunque el modelo sea correcto, hay cosas que deberían modificarse.

Se trata, por tanto, de una extensión de *OCL*.

La estructura básica de un fichero con extensión *".evl"* es la siguiente:

```
context nameContext {
        constraint nameConstraint {
                [guard: Bloque EOL]
                check: Bloque EOL
                [message: Bloque EOL]
                [fix: Bloque EOL]
        }
}
```

donde

- **nameContext:** nombre del tipo de instancia sobre la cual se van a evaluar las restricciones.

- **nameConstraint:** nombre de la restricción.

- **guard:** especifica si la restricción depende de otra cualquiera. En caso de que la dependencia no se resuelva correctamente, la restricción actual no se ejecuta. Es una condición que debe superarse.

- **check:** aquí va el bloque de código *EOL* que define la restricción en sí.

- **message:** mensaje de error personalizado por parte del usuario. Si la restricción se evalúa a falso, se mostrará el mensaje.

- **fix:** sirve para reparar los errores detectados en los modelos. Son bloques de código *EOL* que corrigen el modelo para que se pueda superar la restricción.

Para nuestro proyecto ha sido utilizado para definir los ficheros de validación que indican las características que deben cumplir los modelos transformados. Cada caso de prueba (o test) llevará asociado, a parte del modelo de entrada y una posible configuración, un fichero de validación.

Seguidamente podemos observar un ejemplo de fichero *EVL*:

```
context View {
	constraint viewName {
		check: self.name = 'DEMO'
		message: 'The name of the View must be "DEMO"'
	}

	constraint patternName {
		guard: self.satisfies('viewName')
		check: self.select(v: View | v.name =
'DEMO').first().interactionUnits.exists(p : PatternIU | p.name = 'MenuPrincipal')
		message: 'The view must have a PatternIU named "MenuPrincipal"'
	}
}
```

Ejemplo de sintaxis EVL

3.5. JUnit

Es un conjunto de bibliotecas utilizadas en programación para hacer pruebas unitarias de aplicaciones *Java*.

Es un *framework* que permite la ejecución de clases *Java* de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces *JUnit* devolverá éxito; en case de que el valor esperado sea diferente al que regresó el método durante la ejecución, *JUnit* devolverá un fallo.

Es también un medio de controlar las pruebas de regresión, necesarias cuando una parte del código ha sido modificado y se desea ver que el nuevo código cumple con los requerimientos anteriores y no se ha alterado su funcionalidad después de la nueva modificación.

En la actualidad la herramienta *Eclipse* cuenta con *plugins* que permiten que la generación de las plantillas necesarias para la creación de las pruebas se realice de manera automática, facilitando enfocarse en la prueba y el resultado esperado, dejando a la herramienta la creación de las clases que permiten coordinar las pruebas.

La estructura típica de una clase *JUnit* es la siguiente:

```
class nameClass {
        @BeforeClass
        static void setUpBeforeClass() throws Exception {  …  }

        @Test
        void test1() {  …  }
        @Test
        void test2() {  …  }
        …

        @AfterClass
        static void tearDownAfterClass() throws Exception {  …  }
}
```

donde

- **nameClass:** nombre de la clase. Es indiferente, aunque sea *JUnit*.

- **@BeforeClass:** etiqueta que identifica a un método que se ejecuta siempre antes del lanzamiento de todos los métodos de test.

- **@Test:** etiqueta que identifica a un método como prueba.

- **@AfterClass:** etiqueta que identifica a un método que se ejecuta siempre después del lanzamiento de todos los métodos de test.

Hay otras etiquetas a destacar como @Before y @After que identifican a los métodos que se ejecutan justo antes (o después) de cada uno de los métodos de test.

En nuestro proyecto, la herramienta *JUnit* ha sido empleada para evaluar los resultados de los diferentes casos de prueba diseñados. A cada caso de prueba se le asocia un test *JUnit* que determina si la transformación del modelo ha sido correcta y el modelo resultante cumple con las características esperadas.

A continuación se muestra un ejemplo de clase *JUnit*:

(El código se ha simplificado por cuestión de espacio)

```
public class TransformationAndValidation {
	...
	@BeforeClass
	public static void setUpBeforeClass() throws Exception {
		...
		tav = new TransformAndValidate(transformationId,
				dirValidationInputModels, dirValidations, metamodelFile,
				metamodelUri, transformationInputPath,
				transformationOutputPath, dirConfigurationModels, "uim",
				namePlugin);
	}

	@Test
	public void TFR_MAP_001_V00_001() {
		boolean error;
		error = tav.execute("TFR-MAP-001-V00-001", ol);
		...
		assertFalse(error);
	}// end TFR_MAP_001_V00_001
```

```
        ...

        @AfterClass
        public static void tearDownAfterClass() throws Exception {
                OutputLog.endLog(ol);
                ...
                // prints a little summary of test results
                System.out.println("\nTEST SUMMARY:\n");
                System.out.println("Time elapsed: " + timeTests + " seconds");
                System.out.println("Tests executed: " + nTests);
                System.out.println("Hits: " + nHits);
                System.out.print("Failures: " + nFailures);
                System.out.println("  which " + tav.nErrors + " are errors");
                ...
        }
}
```

Ejemplo de sintaxis JUnit

3.6. Ant

Es una herramienta que se utiliza para la realización de tareas mecánicas y repetitivas, normalmente durante la fase de compilación y construcción. Es, por tanto, un software para procesos de automatización de compilación, similar a *Make*, pero empleando lenguaje *Java*.

Tiene la ventaja de no depender de las órdenes de *shell* de cada sistema operativo, sino que se basa en archivos de configuración *XML* y clases *Java* para la realización de distintas tareas, siendo idónea como solución multi-plataforma.

Cada fichero *Ant* contiene un proyecto (*project*) y al menos un objetivo (*target*). Cada objetivo puede contener varias tareas (*task*) que son fragmentos de código a ejecutar. Un proyecto, además, puede constar de diversas propiedades. Cada propiedad consta de nombre y valor y son usadas para asignar valores a los atributos de los *task*.

En este proyecto, es una de las tecnologías más empleadas, ya que es la herramienta base sobre la que se asienta la construcción de los componentes.

El método de construcción de una plataforma basada en *Eclipse* (como es el caso de *MOSKitt*) se centra en la ejecución de un montón de *scripts* de *Ant* que realizan la compilación y el ensamblaje de todos los *plugins* que conformarán el futuro producto.

Cabe comentar que *Ant* también se ha utilizado en este proyecto para realizar copias de ficheros (modelos) entre directorios, a modo de agilizar un proceso que se debe repetir constantemente antes de la realización de las pruebas.

Seguidamente podemos observar un ejemplo de un fichero *Ant*, en el que se realiza una copia de un repositorio de un directorio local a uno remoto:

```
<project default="mirror">
        <target name="mirror" depends="check">
               <delete dir="${dest}"/>
                <echo>"Mirroring the repository: ${source} to ${dest}"</echo>
                <p2.mirror>
                        <destination location="${dest}"/>
                        <source>
                                <repository location="${source}"/>
                        </source>
                </p2.mirror>
        </target>
        <target name="check" unless="dest">
           <fail message="Uso: mirror -Dsource=URL_REPOSITORY -Ddest=DESTINATION_NAME"/>
        </target>
</project>
```

Ejemplo de sintaxis Ant

3.7. Shell-script

Un *script* es un programa simple, que por lo regular se almacena en un archivo de texto plano. Casi siempre son interpretados. El uso habitual de los *scripts* es realizar diversas tareas como combinar componentes, interactuar con el sistema operativo o con el usuario. Por ese motivo, es frecuente que los *shells* sean a la vez intérpretes de este tipo de programas.

Shell es el término usado para referirse a un intérprete de comandos, que tiene la capacidad de traducir las órdenes que introducen los usuarios, mediante un conjunto de instrucciones facilitadas por él mismo directamente al núcleo y al conjunto de herramientas que forman el sistema operativo.

Así pues, un *shell-script* no es más que un *script* escrito para el *shell* de un sistema operativo determinado. Es considerado también un lenguaje específico del dominio. Algunas de las operaciones típicas que realizan son manipulación de ficheros, ejecución de programas e impresión de texto.

En cuanto a la estructura de un *script*, no tiene un esqueleto básico por el cual guiarnos, ya que simplemente es un fichero de texto con una serie de órdenes de *shell* (una en cada línea). Lo único que tienen en común es la cabecera, en la que se indica el intérprete sobre el cual serán ejecutadas → **#!/bin/bash**

En nuestro proyecto se han empleado para la ejecución de varias tareas de modo automático, en la que se lanzan una serie de instrucciones u órdenes interpretables por el *shell* de *UNIX*.

Con ellos pretendemos automatizar en medida de lo posible labores como descargar fuentes de repositorios, comprimir/descomprimir ficheros, instalar repositorios, publicarlos...

Como se puede observar, se les ha dado un gran uso para el manejo de operaciones con los *repositorios p2* que se utilizan en *Eclipse*. Gracias a estos *scripts* podremos, por ejemplo, descargar la herramienta *MOSKitt*, instalar aquellos *plugins* necesarios para los test (a través de los repositorios), ejecutar las pruebas y visualizar los resultados.

A continuación podemos observar un pequeño ejemplo de *shell-script* que realiza una actualización de un directorio y de todos sus contenidos, descargándolos de un repositorio *SVN*:

```
#!/bin/bash
dir="$HOME/moskitt-galileo-dev/build/es.cv.gvcase.releng.builder"
svnurl="https://svn.gvcase.org/svn/gvcase-releng/branches/moskittbase"

rm -rf ${dir}
echo "Updating releng.builder"
mkdir ${dir}
svn export ${svnurl}/es.cv.gvcase.releng.builder ${dir} –force
```

Ejemplo de sintaxis shell-script

3.8. Subversion (SVN)

Es un sistema de control de versiones, diseñado para reemplazar a *CVS*. Se encarga de automatizar las tareas de guardar, recuperar, registrar, identificar y mezclar versiones de archivos. Además, permite acceder al repositorio a través de la red, lo que facilita ser utilizado por personas que se encuentran en distintas máquinas.

En este proyecto, la herramienta *Subversion* juega un papel muy importante en cuanto al tema de construcción se refiere. Todo el código fuente de los *plugins* que conforman la plataforma *MOSKitt* se encuentra en distintos repositorios; por lo que constantemente se realizan operaciones de consulta *svn* sobre estos repositorios.

3.9. Hudson

Es una herramienta de integración continua escrita en *Java*, que se ejecuta en contenedores de *servlets*, como *Apache Tomcat*. Trabaja con herramientas de control de versiones como *SVN* y puede ejecutar proyectos basados en *Apache Ant*, así como también *shell-scripts*.

Hudson permite monitorizar ejecuciones de tareas repetitivas, como la construcción de un producto software o tareas ejecutadas con *cron*. Principalmente se centra en 2 tareas:

1. Building/testing de proyectos software

Provee de un sistema de integración continua sencillo de utilizar, haciendo más fácil a los desarrolladores integrar cambios en el proyecto y facilitar a los usuarios la obtención de una versión actualizada.

2. Seguimiento de ejecuciones de trabajos de gestión externa

Incluso aquellas que se ejecutan en un equipo remoto. Con *Hudson*, el resultado obtenido se mantiene almacenado en la tarea, haciendo así más sencilla la notificación de los errores.

Hudson es una aplicación web en la cual destacan los siguientes elementos:

- **Tareas o *jobs*:** son las encargadas de realizar el trabajo. El usuario las crea, las configura y las parametriza para ponerlas en ejecución cuando desee. Aceptan una gran cantidad de configuración tanto para la entrada como para la salida del proceso. Las principales opciones de ejecución son:

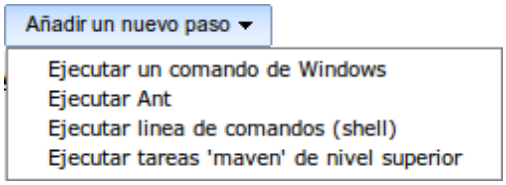


Figura 2. Opciones de ejecución de una tarea en Hudson

- **Vistas:** son agrupaciones de tareas. Hacen más sencilla la visualización por parte del usuario de las tareas que hay creadas.

- **Nodos:** coincide con el número de tareas que se pueden lanzar a ejecución en paralelo.

En nuestro proyecto es la pieza fundamental para que sea posible su ejecución. Es el software que permite el lanzamiento de todos los *scripts* que construyen los módulos necesarios para completar *MOSKitt* y poder lanzar los test contra él. Reúne a todas las tecnologías nombradas anteriormente, ya que se encarga de provocar la puesta en marcha de las mismas.

Centrándonos un poco más en este proyecto en concreto, podemos resumir diciendo que gracias a *Hudson* es posible la construcción de *MOSKitt* y del módulo que contiene la transformación *Sketcher2UIM;* así como del *plugin* que contiene los test a ejecutar. Una vez construidos, se instalan en la plataforma y se lanzan las pruebas en modo *headless*, observando el resultado final en la salida de consola de la tarea de *Hudson*.

En la **figura 3** podemos observar un ejemplo del aspecto de *Hudson*, donde se aprecia una lista de tareas creadas para la construcción de algunos módulos funcionales de *MOSKitt*:

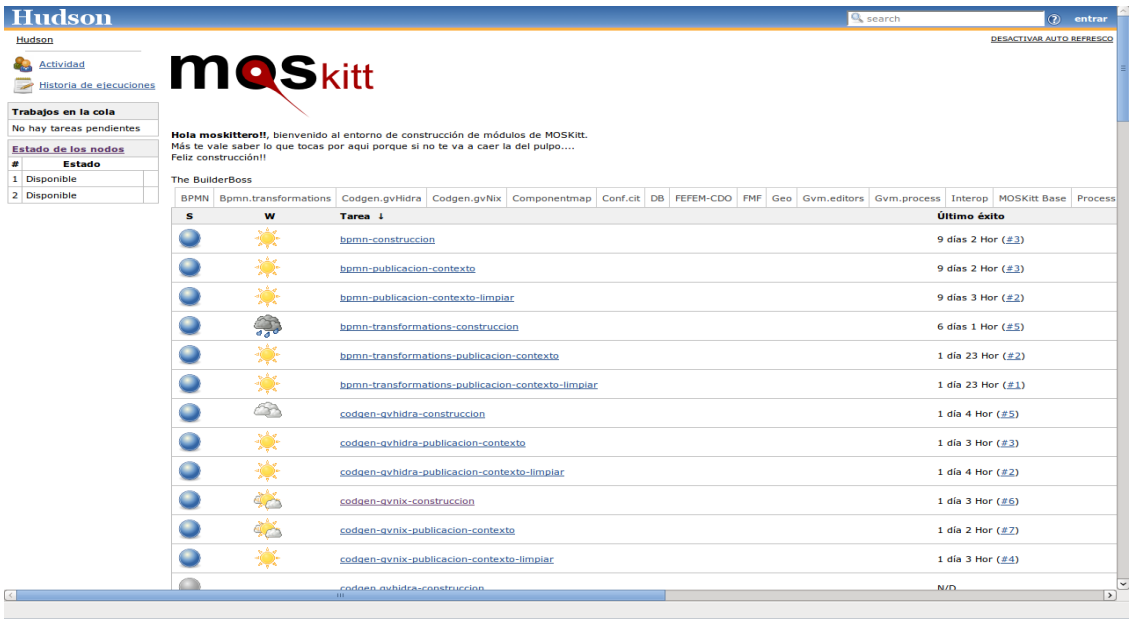


Figura 3. Interfaz de Hudson

Una vez vistas todas las tecnologías y herramientas que se van a utilizar para la implementación del proyecto, pasaremos a detallar la parte de análisis.

En el siguiente apartado podremos obtener una visión general de toda la estructura del proyecto, así como de todas las fases por las que transcurre.

4. Análisis

Este apartado está dedicado a la descripción de lo que se realiza en nuestro proyecto, centrándose en *qué* consiste, sin entrar en detalle de *cómo* lo hace (aspecto que trataremos en el apartado de diseño).

Iremos paso por paso comentando las diferentes fases o secciones en las que se divide el proyecto, desde la automatización de las pruebas, pasando por el proceso de construcción de los componentes de la herramienta *MOSKitt* hasta llegar a la ejecución de las pruebas en dicho entorno.

Hay que tener claro que las pruebas forman parte de una transformación modelo a modelo que se realiza en *MOSKitt* (*Sketcher2UIM*). Además, para poder lanzarlas, debemos tener construido el módulo que contiene la implementación de las pruebas (de eso se encarga el proceso de construcción) y al final de su ensamblaje se podrán ejecutar para detectar posibles fallos (integración de los test en la construcción).

4.1. Proceso de automatización de las pruebas

En este apartado comentaremos en qué consiste el proceso de automatización de las pruebas sobre transformaciones de modelos que se ha reutilizado y mejorado en ciertos aspectos.

¿En qué consiste el proceso automático de pruebas?

Muy sencillo. Como hemos comentado anteriormente, existe una necesidad de probar todo aquello que se implementa nuevo para una herramienta. Para eso, existen las pruebas o test.

Se diseñan una serie de casos de prueba, en los que, a partir de una serie de datos de entrada, se producen unos de salida, y luego se comprueba que todo marcha correctamente.

Así pues, para este proyecto en concreto, ya que se va a probar una transformación modelo a modelo, se deben crear una serie de modelos de entrada, que producirán sendos de salida; y luego comprobaremos que la transformación ha tenido éxito.

El método o proceso a seguir desde la creación de los modelos de entrada, de las validaciones o comprobaciones a realizar, y el modo de integrarlo todo en nuestra

herramienta para lanzarlo de forma automática es lo que denominamos *proceso de automatización de las pruebas*.

Este montaje se implementó recientemente para empezar a utilizarlo en el proyecto *MOSKitt*, aunque no se ha llegado a utilizar debido a que no llega a ser un proceso tan automático como se desea.

Así pues, mediante el diseño propuesto en este proyecto, se pretende mejorar ciertos aspectos que mejoren su estructura, funcionamiento y sobre todo su adaptación al lanzamiento en el proceso de construcción; para que se pueda empezar a utilizar y así reducir el coste dedicado a probar las nuevas implementaciones.

La **figura 4** nos muestra una representación gráfica del proceso completo de automatización, el cual procederemos a explicar más detalladamente en los siguientes apartados.

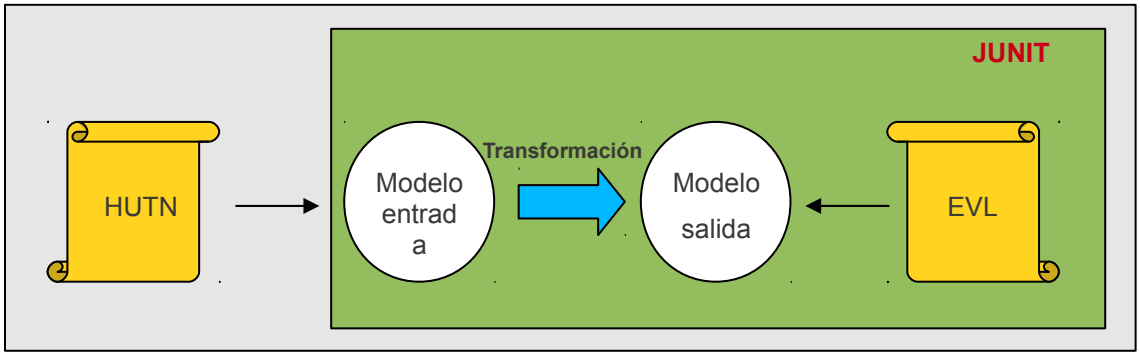


Figura 4. Proceso completo de automatización de las pruebas

A modo de resumen, podemos comentar que el proceso consiste en la definición de una serie de ficheros con *HUTN*, con los que se generan los modelos de entrada (*sketcher*) con las propiedades deseadas. A estos modelos se les aplica la transformación correspondiente (*Sketcher2UIM*) produciendo así modelos de salida (*UIM*). Estos modelos de salida son los que hay que validar. Para ello se definen ficheros de restricciones en *EVL* que validarán dichos modelos. Tanto la transformación como la validación son subprocesos integrados en *JUnit* mediante test unitarios, ya que ambas partes son las que pueden dar errores durante el proceso. Si ambos subprocesos funcionan correctamente el test será válido, por tanto, la transformación se habrá realizado con éxito.

Hay que destacar también que dicho proceso se puede separar en 2 subprocesos independientes. El primero de ellos consiste en la ejecución de un *script ANT* que

copia los modelos de configuración (en caso de que hiciesen falta) y algunos modelos auxiliares, junto con la generación de los modelos de entrada con *HUTN*. El otro de los subprocesos sería la transformación y la validación integradas con *JUnit*.

Así pues, en los siguientes apartados procederemos a explicar los diferentes pasos de los que está compuesto el proceso entero de automatización.

4.1.1. Generación de los modelos de entrada

Este paso conlleva a la vez 2 pequeñas acciones:

1. Redacción de los ficheros mediante el lenguaje *HUTN*, en los cuales se especifican todos los elementos y propiedades que los modelos deben contener. A través de estos ficheros obtendremos los modelos de entrada.

2. En caso de que hubiesen modelos de configuración de la transformación (aunque no es el caso para *Sketcher2UIM*) o algunos modelos intermedios o auxiliares que requiera la transformación; se tiene que ejecutar un *script ANT* que realiza una copia de esos ficheros a un directorio que pasará a formar parte del *workspace* de la plataforma en la que se ejecuten los test.

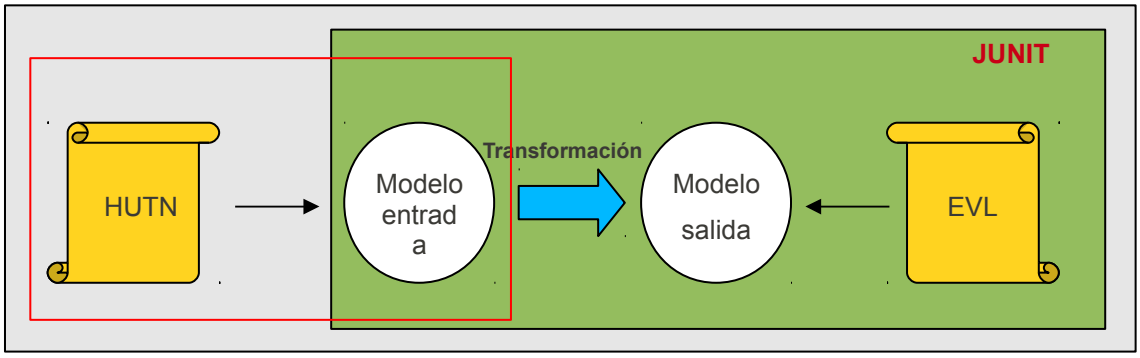


Figura 5. Generación de los modelos de entrada

Los modelos de entrada generados son de tipo *sketcher*. La herramienta *Sketcher* es un *diseñador* presente en *MOSKitt* de bocetos de interfaces de usuario (páginas web, formularios, informes, ventanas, etc.), que puedan utilizarse para validar el análisis en fases tempranas y diseñar el aspecto que la interfaz deba cumplir.

Los metamodelos de los cuales se instancian estos modelos de entrada pueden estar sometidos a frecuentes cambios, por lo que generar los modelos mediante editores de

forma manual es una técnica no muy recomendable; ya que cualquier cambio del metamodelo no garantiza que siga siendo un modelo válido.

Este es el motivo por el cual se ha escogido *HUTN* como lenguaje para generar instancias de los metamodelos. Como en las plantillas definidas con *HUTN* se especifica el metamodelo concreto con el cual estamos trabajando, simplemente con modificar esa ruta con el metamodelo actualizado sabremos en todo momento si nuestro modelo sigue siendo válido. Así también sabremos en cada instante qué propiedades o elementos han cambiado y podremos modificarlos fácilmente.

En cuanto a la copia de los modelos auxiliares, cabe comentar que el motivo por el cual no se dejan junto con el resto de modelos de entrada es porque estos modelos contienen información que puede ser sobreescrita en el proceso de la transformación, con lo cual, al tenerlos en el propio *plugin* instalado en la plataforma, esa opción de reescribir sobre el *plugin* no estaría permitida. Por eso deben estar separados en otro directorio, en este caso, en el *workspace* correspondiente desde el cual se lancen los test.

4.1.2. Transformación de los modelos

Una vez obtenidos los modelos de entrada específicos para cada prueba diseñada, se procede a la ejecución de la transformación de los mismos.

El objetivo de este paso es que a partir de un modelo de entrada se genere uno de salida. Hay que tener en cuenta que el proceso de *testing* es de *caja negra*, no entramos en lo que es el código de la transformación, sino que debemos comprobar que la salida es la esperada para los datos de entrada.

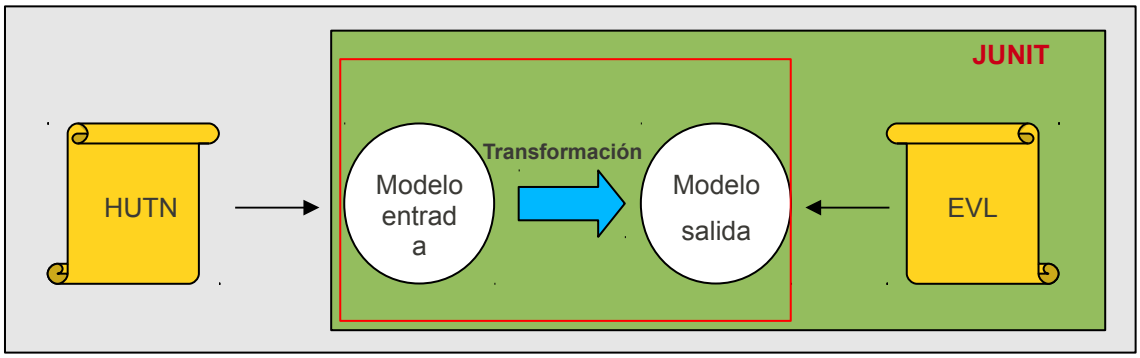


Figura 6. Transformación de los modelos

Las transformaciones en *MOSKitt* se invocan a través de un menú contextual de forma gráfica. Se selecciona un modelo a transformar y al pulsar la acción de transformar del menú se inicia un asistente para configurar la transformación y ejecutarla.

Como es de esperar, este modo de realizar la transformación no es adecuado para poder automatizar el proceso, ya que no disponemos de herramientas que realicen de un modo fiable la serie de acciones para lanzarla con la interfaz gráfica.

Así pues, las transformaciones serán lanzadas a través de código, introduciendo los parámetros necesarios y llamando al gestor de forma textual.

4.1.3. Validación de los modelos de salida

En este punto, nos encontramos con una serie de modelos de salida los cuales tendrán que ser validados para confirmar que la transformación ha funcionado correctamente, si no es que ha habido un error previo.

Para ello se definen una serie de ficheros *EVL* (uno por modelo) que contendrán las restricciones que deben ser cumplidas por los modelos de salida.

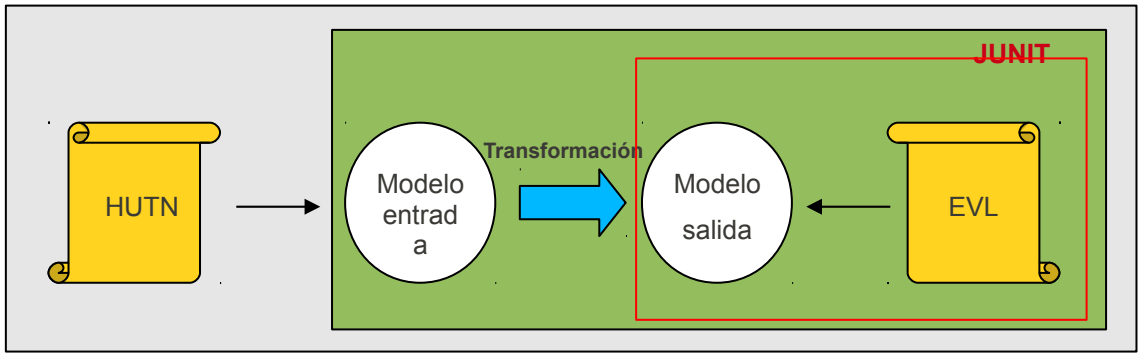


Figura 7. Validación de los modelos de salida

Existen diversas formas de validar un modelo, entre ellas:

- Podemos crear a mano el modelo esperado y compararlo con el resultante
- Comprobar que existen los elementos esperados
- Verificar el valor de determinadas propiedades

En nuestro caso se ha optado por escoger una mezcla entre las 2 últimas opciones descritas, ya que se pueden llevar a cabo por medio de consultas. En esto consiste

EVL, un lenguaje mediante el cual podemos definir restricciones en forma de consultas sobre el modelo que verifiquen que se contienen los elementos esperados y que las propiedades poseen los valores que deben tener.

El modo de lanzar las validaciones es también de forma programática, ya que no se dispone de ningún gestor que compruebe las restricciones. Así pues, se dispondrá de una jerarquía de clases mediante las cuales sabremos si un modelo ha resultado válido o no, junto con los mensajes resultantes asociados.

En el siguiente apartado se explicará cómo se ha integrado tanto la realización de la transformación como la ejecución de las validaciones contra los modelos.

4.1.4. Integración en JUnit

Llegados a este nivel, ya disponemos de toda la información necesaria para confirmar si el proceso ha sido correcto o no. Disponemos de los resultados de la transformación así como de los devueltos por las validaciones. Con todos esos datos, simplemente nos queda programar los test que nos dirán si se han pasado o no las pruebas.

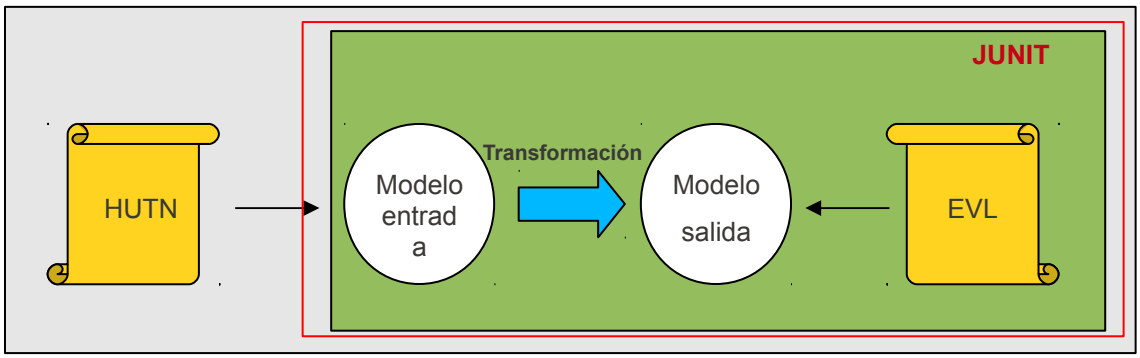


Figura 8. Integración en JUnit

Gracias a la herramienta *JUnit* somos capaces de componer una serie de test unitarios en los que se hará una llamada a la transformación *Sketcher2UIM* para un modelo de entrada concreto; posteriormente también se hará otra llamada a la jerarquía de clases de *EVL* que nos devolverá el resultado de la validación del modelo de *UIM*.

Estos pasos se realizarán una y otra vez por cada prueba diseñada, obteniendo al final un pequeño resumen del resultado de todos los test: número de test ejecutados, cuántos han fallado y cuántos han pasado, tiempo que han tardado en ejecutarse y una lista con los errores de las pruebas que hayan resultado erróneas.

4.2. Proceso de construcción

Cuando hablamos de proceso de construcción nos referimos a aquel en el que, a partir de un conjunto de código fuente, éste se compila y se ensambla para obtener un "producto". Todo esto empleando el método proporcionado por el *builder* de *Eclipse*.

El producto resultante siempre es un repositorio p2. Los repositorios p2 son los componentes con los que se maneja *Eclipse* para añadir, actualizar o eliminar *plugins*.

Dependiendo del tipo de construcción, pueden aparecer 2 tipos de repositorios:

➔ RCP

La *RCP* es lo que conocemos como la herramienta *MOSKitt*. Está dotada de un entorno de trabajo (vistas, editores, perspectivas, *wizards*...), un núcleo, un conjunto de *plugins*, un gestor de actualizaciones, etc. Además, se puede extender con módulos funcionales.

En resumidas cuentas, se trata de un repositorio p2 que además se puede ejecutar y es capaz de levantar una interfaz gráfica.

➔ Módulo funcional

Conjunto de *plugins* que sirven para extender a la *RCP*. Añaden nuevas funcionalidades a *MOSKitt*. Un repositorio p2 puro y duro.

Nuestro proyecto se centra en la obtención del segundo tipo de producto, el módulo funcional. Se parte de la existencia de una *RCP* de *MOSKitt* ya construida.

De este modo, nos encargamos de construir los módulos referentes tanto a la transformación que vayamos a testear, como al propio módulo que contiene el conjunto de pruebas a lanzar.

Este proceso está automatizado con *Hudson*, aunque actualmente no se aprovecha para realizar construcciones programadas (por ejemplo, diarias); ya que sólo cuando hay necesidad de construir se invoca. Aun así, hay que destacar que cabe la posibilidad de programar las construcciones de un modo sencillo.

Entrando ya en detalle, dicho proceso de construcción es realizado en modo *headless* para su automatización. Así pues, durante todo el proceso no es necesario en ningún momento levantar una interfaz gráfica de la herramienta.

La base sobre la que se realiza la compilación y mediante la cual se genera el producto es una versión minimalista de *Eclipse*, proporcionada por *org.eclipse.releng,basebuilder* (aunque cualquier *SDK* de *Eclipse* sirve). Para la construcción de componentes o módulos que complementan a *Eclipse* disponemos de *org.eclipse.releng.eclipsebuilder*.

La construcción está basada en la ejecución de una serie de ficheros *Ant*. Aunque el *build.xml* de *org.eclipse.releng.eclipsebuilder* sea un único fichero de construcción, hay una serie de tareas *Ant* requeridas que son proporcionadas por el *PDE* (un *plugin* contenido dentro de *org.eclipse.releng,basebuilder*). Por ese motivo el *script* debe ejecutarse en un entorno de desarrollo *Eclipse*. Como además se lanza la instancia de *Eclipse* en modo *headless* para ejecutar el *build.xml*, se debe de arrancar como una aplicación de tipo *AntRunner*.

En la siguiente tabla podemos encontrar los diferentes ficheros implicados en la construcción, quién los proporciona y una pequeña descripción de su contenido:

Fichero	Localización	Descripción
build.xml	basebuilder	Es el fichero principal y proporciona el esqueleto para el proceso de construcción
genericTargets.xml	basebuilder	Contiene *targets* como fetchElement, generateScript, processElement, assembleElement...
customTargets.xml	eclipsebuilder	Define las features a construir y descarga los ficheros *.map*
build.properties	eclipsebuilder	Contiene propiedades para obtener los fuentes y diversas propiedades sobre la compilación y construcción

Tabla 1. Ficheros que controlan el proceso de construcción

En la siguiente imagen se observa la interacción que hay entre los diversos ficheros que realizan el proceso entero:

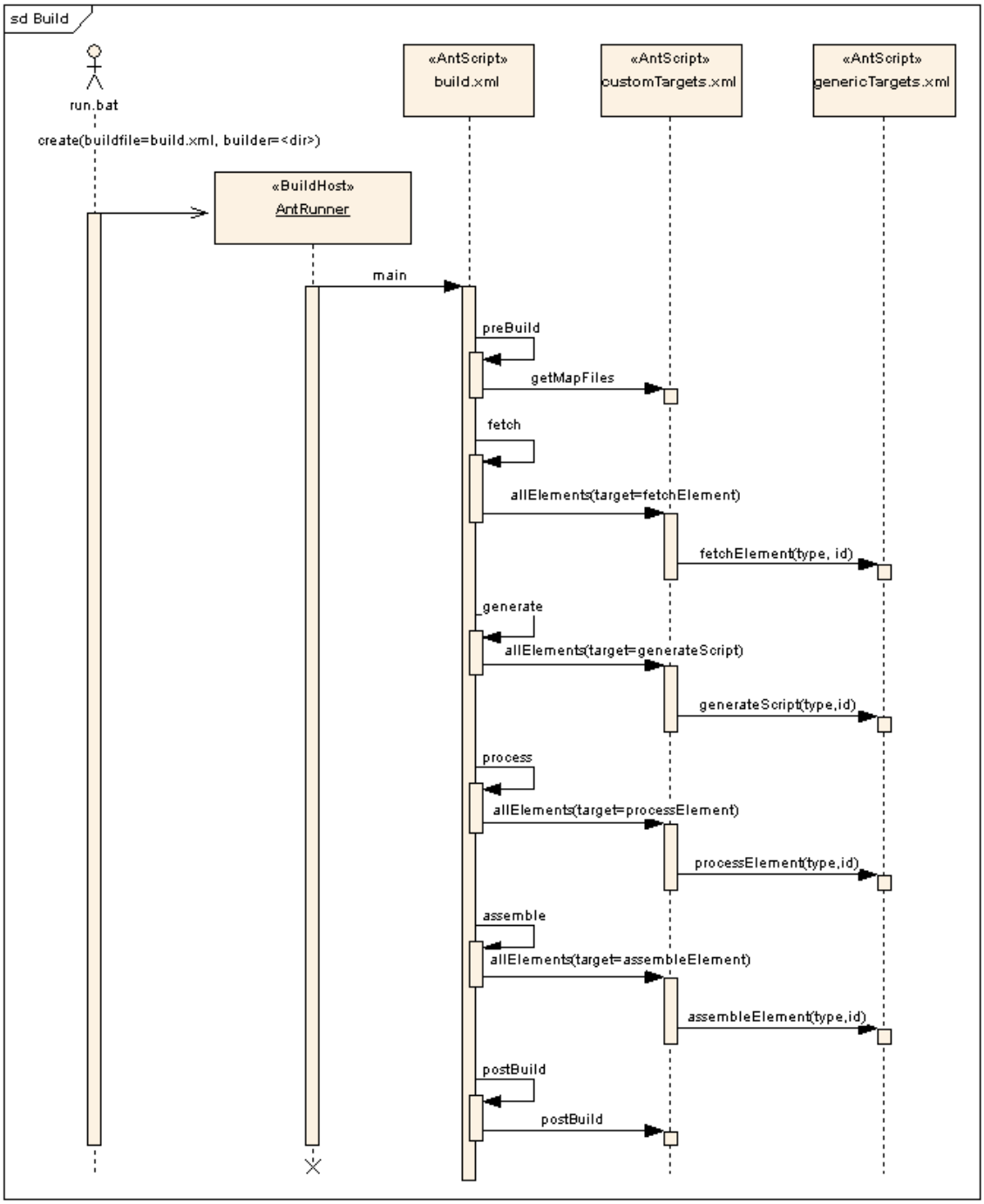


Figura 9. Interacción de los *scripts* en el proceso de construcción

La **figura 10** nos da una visión de las actividades principales que sirven para completar el proceso de construcción:

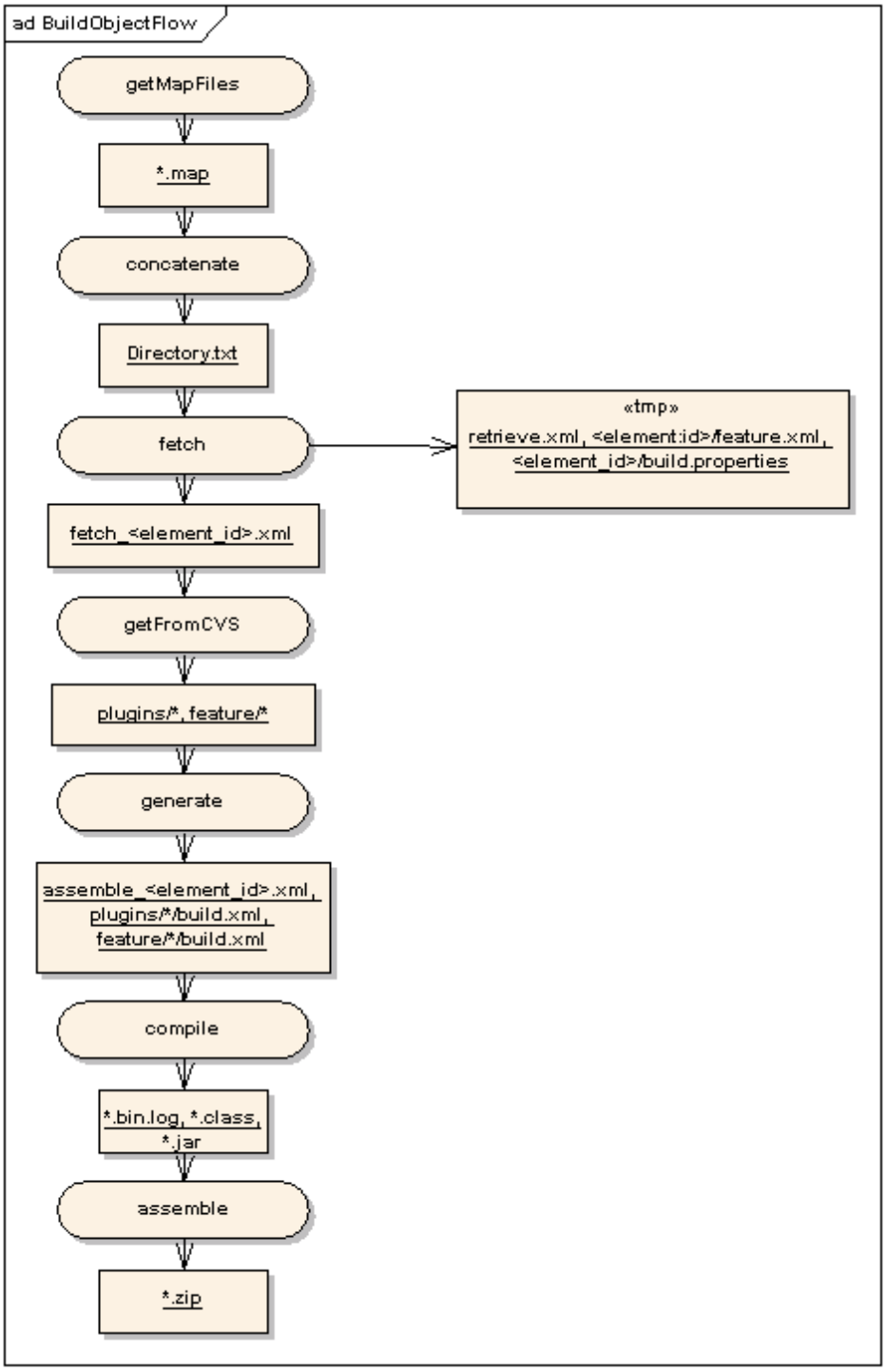


Figura 10. Procesos principales de la construcción

El proceso empieza con la tarea *getMapFiles*, que consiste en descargar los *.map* que servirán para descargar los fuentes del repositorio. Una vez todos obtenidos, se genera un *directory.txt* que agrupa el contenido de todos los ficheros *map* en un único fichero.

Posteriormente el proceso seguirá con la creación de un fichero *Ant* llamado *retrieve.xml*, que contiene un *target* para obtener la *feature.xml*. Se leerá el contenido de la *feature* para saber los *plugins* incluidos en la misma y así poder generar un fichero *fetch_<element_id>.xml* que será el encargado de descargar todo el contenido referente a un elemento en concreto del repositorio.

Una vez descargados todos los fuentes, el siguiente paso es la creación de 3 ficheros: un *build.xml* por cada *plugin*, otro por cada *feature* y un *assemble_<element_id>.all.xml*.

A esta altura del proceso, la compilación es lanzada con una llamada a *processElement* en el fichero *genericTargets.xml*. Aquí empiezan a aparecer todos los mensajes de warning o errores resultantes de la compilación. El resultado es un *.jar* por cada *feature* y *plugin*.

En el caso de que hubiesen test de prueba, éste sería el momento en el que se ejecutarían, de modo que si fallasen debería detener todo el proceso, ya que se estaría construyendo un producto funcionalmente inválido.

El último de todos los pasos consiste en generar los *.zip* que engloban el producto final. En nuestro caso, al tratarse de un repositorio que contiene un módulo funcional, el resultado es un directorio con una serie de archivos:

- **features:** subdirectorio que contiene los *.jar* de las *features* construidas.
- **plugins:** subdirectorio que contiene los *.jar* de los *plugins* construidos.
- **artifacts.xml:** indica la localización de los *plugins* en el repositorio.
- **content.xml:** indica el contenido del repositorio.

Conocido ya el proceso de construcción, pasamos a comentar la introducción de los casos de prueba en el mismo, cosa que veremos en el siguiente apartado.

4.3. Integración de las pruebas en la construcción

Antes de entrar más en detalle acerca de lo que se realiza para incorporar los test en la construcción, debemos conocer cuándo se lanzan dichas pruebas.

En este proyecto nos hemos centrado en la transformación *Sketcher2UIM*, sin embargo, en la planificación se ha comentado que también han sido actualizadas las diferentes pruebas que habían para algunas otras transformaciones presentes en *MOSKitt*.

De esta manera, podemos integrar cualquiera de este tipo de pruebas a la construcción de sus módulos correspondientes. En el instante en el que el módulo que contenga a la transformación y el *plugin* que contiene las pruebas son construidos, podremos lanzar el proceso de *testing* que comprobará el correcto funcionamiento de las transformaciones.

Hasta ahora, debíamos disponer de la herramienta *MOSKitt* con todos los *plugins* de la transformación a probar instalados, así como los diferentes modelos de entrada y el *plugin* de test en el *workspace*. Se lanzaba de forma manual una segunda instancia, que comprobaba la transformación mediante los test y te devolvía el resultado por la consola de *MOSKitt*.

Con la nueva implementación realizada en este proyecto, ha aumentado la automatización en todos los aspectos, ya que con tener una *RCP* de *MOSKitt* mínima, se le pueden ir instalando automáticamente los módulos requeridos y luego lanzar las pruebas, todo esto sin necesidad de levantar su interfaz gráfica, sin necesidad de segunda instancia y tan sólo ejecutando una tarea de *Hudson*.

Como comentábamos en el apartado anterior, en el proceso de construcción se ha incluido un apartado referente a la ejecución de test que comprueben la funcionalidad del producto.

Gracias a las diferentes tecnologías utilizadas en nuestro proyecto, comentadas en el capítulo anterior, hemos podido diseñar una serie de pruebas en forma de modelos de entrada que serán transformados y su resultado será validado; todo agrupado en una suite de test ejecutada en la plataforma.

La herramienta *JUnit* ofrece una serie de *plugins* para *Eclipse* que nos permite ejecutar un conjunto de test que estén contenidos dentro de un *plugin*. El hecho de que puedan incluirse los métodos de test en un *plugin* facilita mucho la tarea, (ya que *MOSKitt* es un conjunto de *plugins)* por lo que simplemente con hacer depender al *plugin* de test con aquellos que sean necesarios para realizar la transformación y validar los modelos ya estaremos en condiciones de lanzarlos.

En concreto, la aplicación que nos permite lanzar los test contenidos en una clase de un *plugin* se le conoce como *JUnit Plugin Test*.

Se trata de una de las tantas aplicaciones que *Eclipse* puede ejecutar, y por tanto, es posible una configuración de la misma. Los parámetros más importantes a configurar son:

1. Nombre del proyecto (*plugin*)
2. Situación del workspace
3. Nombre de la clase que contiene los test
4. Versión de los test de *JUnit*

En el siguiente capítulo detallaremos más a fondo los valores que toman estos parámetros para nuestro proyecto.

Hasta el momento, hemos comentado cómo es posible lanzar los test desde la plataforma, pero ¿cómo lo hacemos sin necesidad de levantar la interfaz gráfica de la misma?

Para ello, se ha definido un *script* que realiza una llamada a la plataforma en modo *headless* y que se encarga de ejecutar la aplicación *org.eclipse.pde.junit.runtime.uitestapplication*, que coincide con la mencionada anteriormente, junto con todos sus parámetros configurados.

Este *script* es ejecutado en el momento oportuno durante el proceso de construcción, en el que se dispondrá de una versión ejecutable del producto (con los *plugins* que necesite instalados); pudiendo así lanzar el conjunto de test y observar el resultado en la salida del proceso.

En la sección referente al diseño de la automatización de los test conoceremos con más detalle la implementación de todo este proceso.

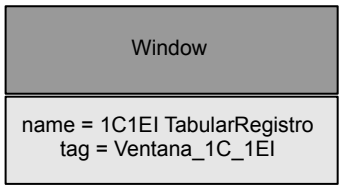
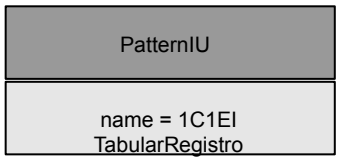
5. Diseño

5.1. Pruebas para la transformación Sketcher2UIM

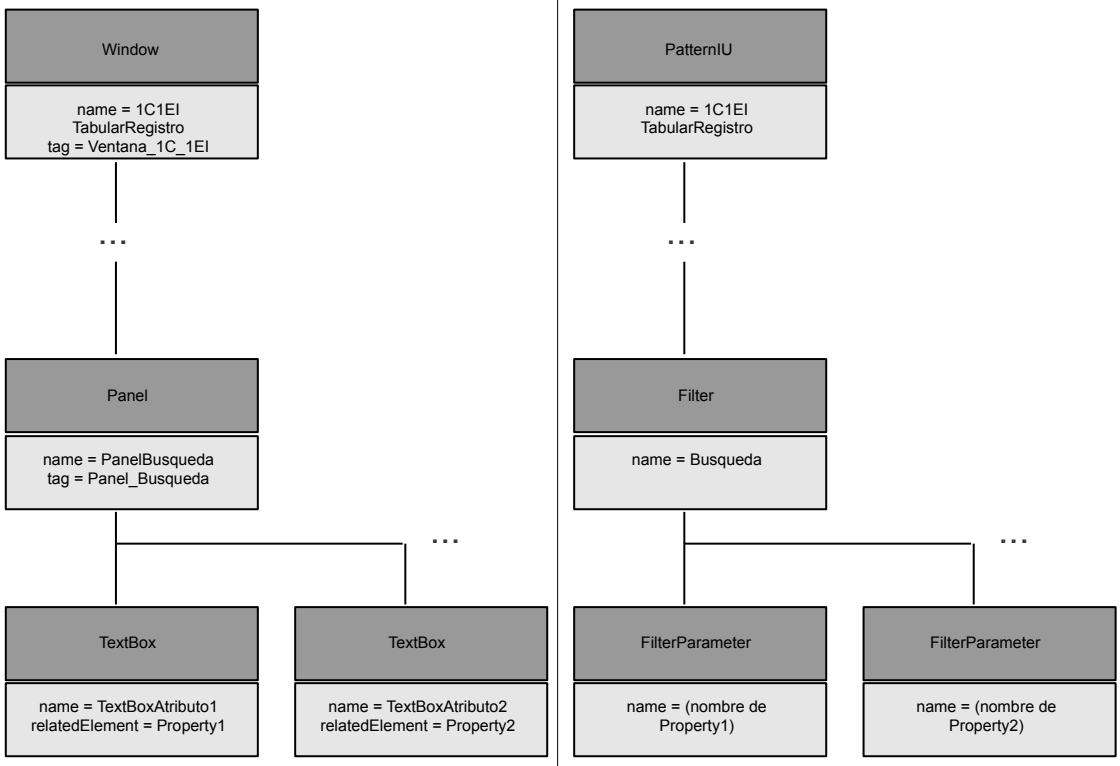
Todas las pruebas que deben realizarse para comprobar la transformación *Sketcher2UIM* han sido diseñadas desde cero. Se decidió que, debido a que *Sketcher* trabaja con patrones o plantillas prediseñadas, y que la transformación a *UIM* no tiene sentido si no se parte de alguna de estas plantillas; cada uno de los distintos casos de prueba coincidiría con uno de esos patrones.

A continuación representaremos en forma de tabla cada uno de las distintos casos de pruebas diseñados para cada tipo de patrón.

5.1.1. Patrón 1C1EI TabularRegistro

Identificador	Descripción
TFR-001-V00-001	Un elemento *Window* '1C1EI TabularRegistro' genera un elemento *PatternIU*
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**
	
Pruebas realizadas	
- Elemento *PatternIU* con *name* = “1C1EI TabularRegistro” - Elemento *InformationIU* con *name* = “Ventana Consulta” - Elemento *VisualizationSet* con *name* = “Ventana Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular - Elemento *ClassView* con *name* = “ClassView1” → contiene 6 *visibleFeature* → 5 *VisibleAttribute* y 1 *VisibleOperation* - Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter* - Elemento *EditableInformationIU* con *name* = “Ventana Edición-Inserción” - Elemento *VisualizationSet* con *name* = “Ventana Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro - Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute* - Elemento *Operations* → contiene 2 *Operation*	

Identificador	Descripción
TFR-001-V01-001	Elementos *Property* procedentes de un modelo UML relacionados en los elementos *TextBox* del Panel de Búsqueda
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**



Pruebas realizadas

- Elemento *PatternIU* con *name* = “1C1EI TabularRegistro”

  - Elemento *InformationIU* con *name* = “Ventana Consulta”

    - Elemento *VisualizationSet* con *name* = “Ventana Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

      - Elemento *ClassView* con *name* = “ClassView1” → contiene 6 *visibleFeature* → 5 *VisibleAttribute* y 1 *VisibleOperation*

      - Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter* (los nombres coinciden con las *Property* asociadas)

  - Elemento *EditableInformationIU* con *name* = “Ventana Edición-Inserción”

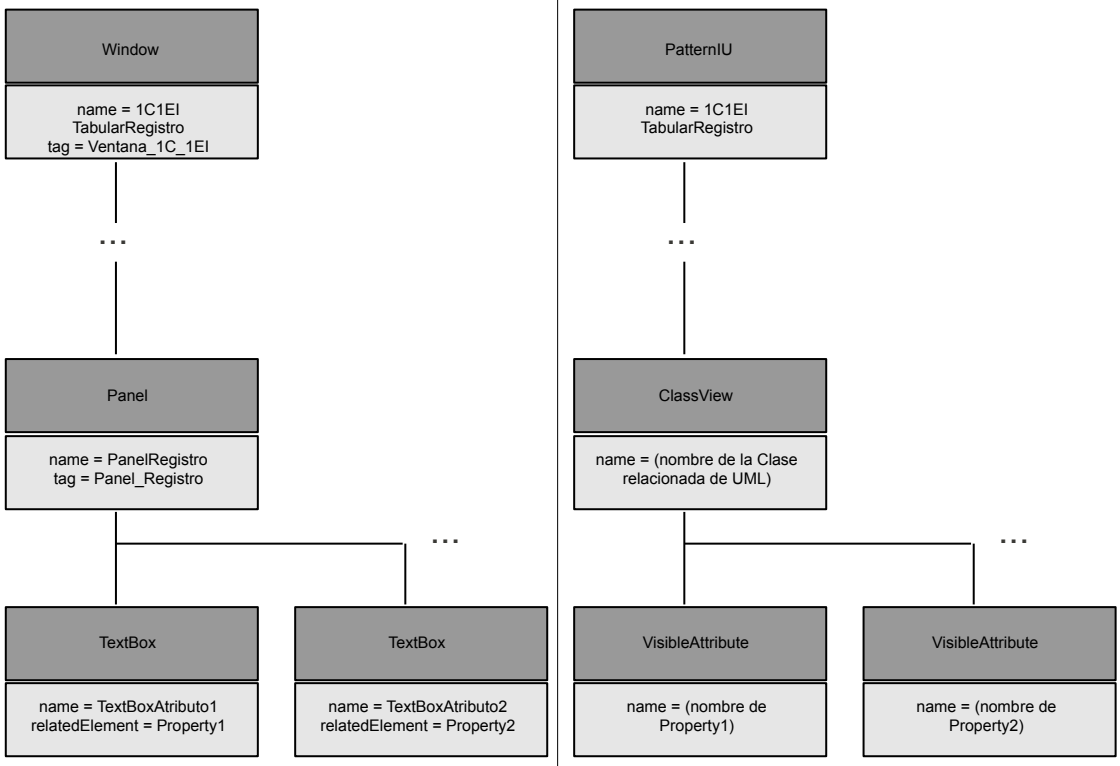
    - Elemento *VisualizationSet* con *name* = “Ventana Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

      - Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

    - Elemento *Operations* → contiene 2 *Operation*

El atributo *relatedProperty* de los *FilterParameter* apunta al *Property* relacionado

Identificador	Descripción
TFR-001-V02-001	Elementos *Property* procedentes de un modelo UML relacionados en los elementos *TextBox* del Panel de Registro
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**



Pruebas realizadas

- Elemento *PatternIU* con *name* = “1C1EI TabularRegistro”

- Elemento *InformationIU* con *name* = “Ventana Consulta”

- Elemento *VisualizationSet* con *name* = “Ventana Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- Elemento *ClassView* con *name* = “ClassView1” → contiene 6 *visibleFeature* → 5 *VisibleAttribute* y 1 *VisibleOperation*

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*

- Elemento *EditableInformationIU* con *name* = “Ventana Edición-Inserción”

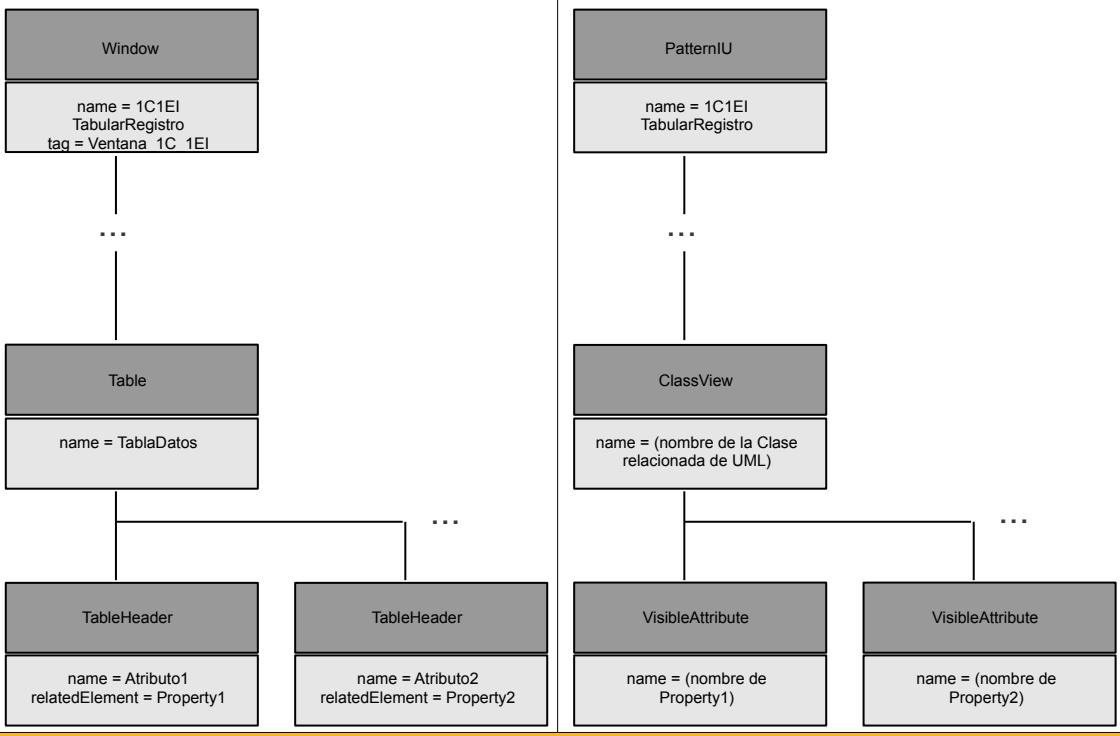
- Elemento *VisualizationSet* con *name* = “Ventana Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = (nombre clase UML) → contiene 4 *visibleFeature* → 4 *VisibleAttribute* (nombre de las *Property* relacionadas)

- Elemento *Operations* → contiene 2 *Operation*

El atributo *attribute* de los *VisibleAttribute* apunta al *Property* relacionado

Identificador	Descripción
TFR-001-V03-001	Elementos *Property* procedentes de un modelo UML relacionados en los elementos *TableHeader* del Panel Tabular
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**



Pruebas realizadas

- Elemento *PatternIU* con *name* = “1C1EI TabularRegistro”

  - Elemento *InformationIU* con *name* = “Ventana Consulta”

    - Elemento *VisualizationSet* con *name* = “Ventana Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

      - Elemento *ClassView* con *name* = (nombre clase UML) → contiene 6 *visibleFeature* → 5 *VisibleAttribute* (nombre de las *Property* relacionadas) y 1 *VisibleOperation*

      - Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*

  - Elemento *EditableInformationIU* con *name* = “Ventana Edición-Inserción”

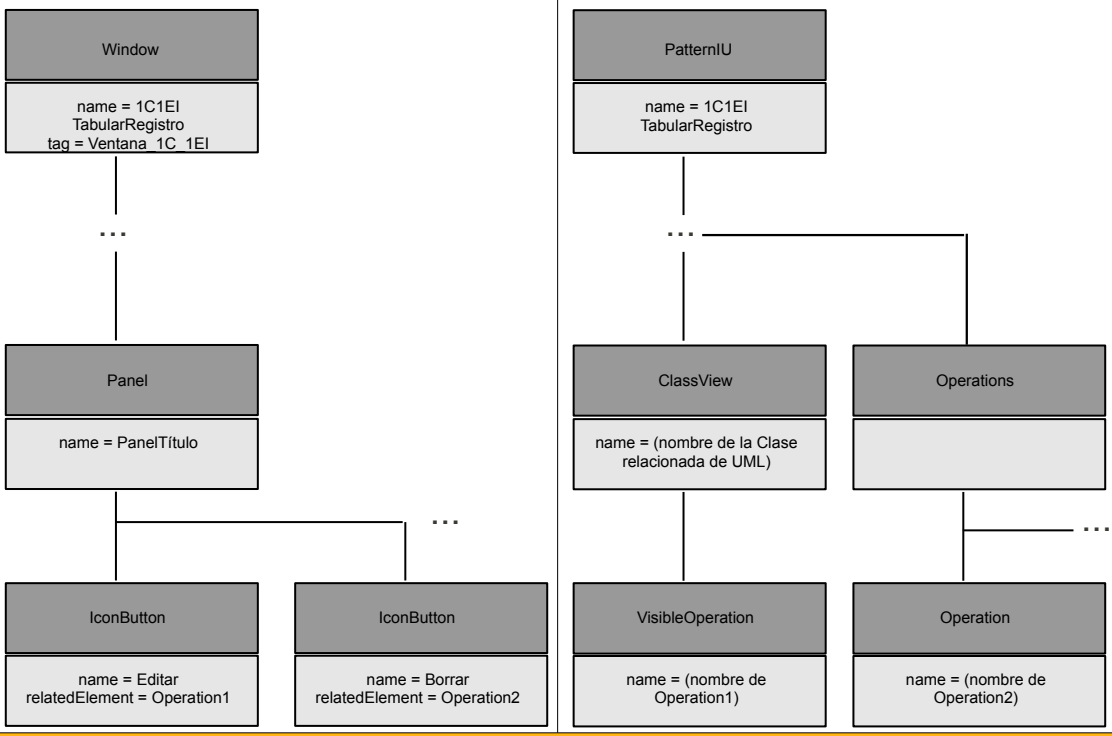
    - Elemento *VisualizationSet* con *name* = “Ventana Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

      - Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

    - Elemento *Operations* → contiene 2 *Operation*

El atributo *attribute* de los *VisibleAttribute* apunta al *Property* relacionado

Identificador	Descripción
TFR-001-V04-001	Elementos *Operation* procedentes de un modelo UML relacionados en los elementos *IconButton* del Panel Título
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**



Pruebas realizadas

- Elemento *PatternIU* con *name* = “1C1EI TabularRegistro”

- Elemento *InformationIU* con *name* = “Ventana Consulta”

- Elemento *VisualizationSet* con *name* = “Ventana Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- Elemento *ClassView* con *name* = (nombre clase UML) → contiene 1 *visibleFeature* → 1 *VisibleOperation* (nombre de la *Operation* relacionada)

- Elemento *ClassView* con *name* = “ClassView1” → contiene 5 *visibleFeature* → 5 *VisibleAttribute*

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*

- Elemento *EditableInformationIU* con *name* = “Ventana Edición-Inserción”

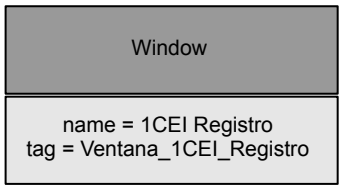
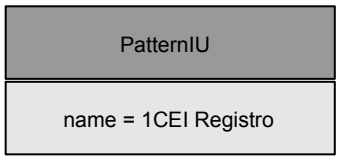
- Elemento *VisualizationSet* con *name* = “Ventana Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

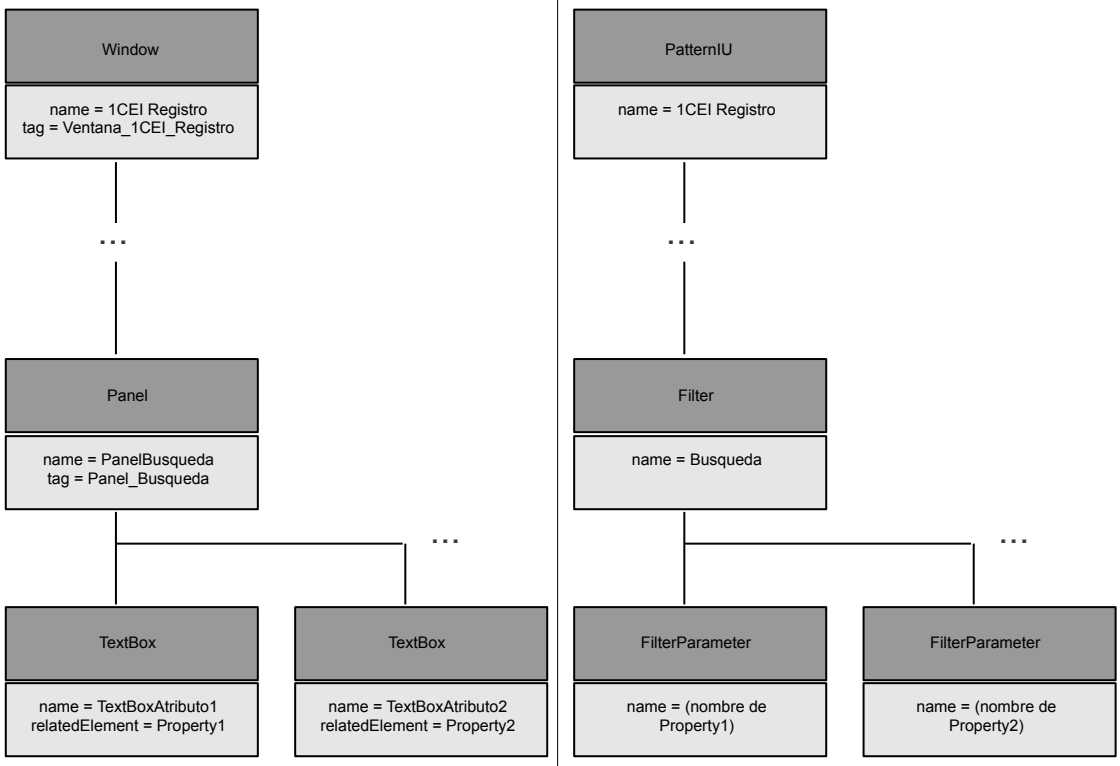
- Elemento *Operations* → contiene 2 *Operation* (nombre de las *Operation* relacionadas)

El atributo *operation* de los *VisibleOperation* apunta al *Operation* relacionado

5.1.2. Patrón 1CEI Registro

<table>
<tr><th>Identificador</th><th>Descripción</th></tr>
<tr><td>TFR-002-V00-001</td><td>Un elemento Window '1CEI Registro' genera un elemento PatternIU</td></tr>
<tr><th>Severidad</th><th>Tipo</th></tr>
<tr><td>Bloqueante</td><td>Funcional</td></tr>
<tr><th>Datos de entrada</th><th>Datos de salida esperados</th></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><th colspan="2">Pruebas realizadas</th></tr>
<tr><td colspan="2">- Elemento PatternIU con name = “1CEI Registro”
- Elemento EditableInformationIU con name = “Ventana Consulta-Edición-Inserción”
- Elemento VisualizationSet con name = “Ventana Consulta-Edición-Inserción” → contiene un visualizationPattern y un defaultVisualizationPattern de tipo Registro
- Elemento ClassView con name = “ClassView1” → contiene 5 visibleFeature → 4 VisibleAttribute y 1 VisibleOperation
- Elemento Filter con name = “Búsqueda” → contiene 4 filterParameter
- Elemento Operations → contiene 2 Operation</td></tr>
</table>

Identificador	Descripción
TFR-002-V01-001	Elementos *Property* procedentes de un modelo UML relacionados en los elementos *TextBox* del Panel de Búsqueda
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**



Pruebas realizadas

- Elemento *PatternIU* con *name* = “1CEI Registro”

- Elemento *EditableInformationIU* con *name* = “Ventana Consulta-Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

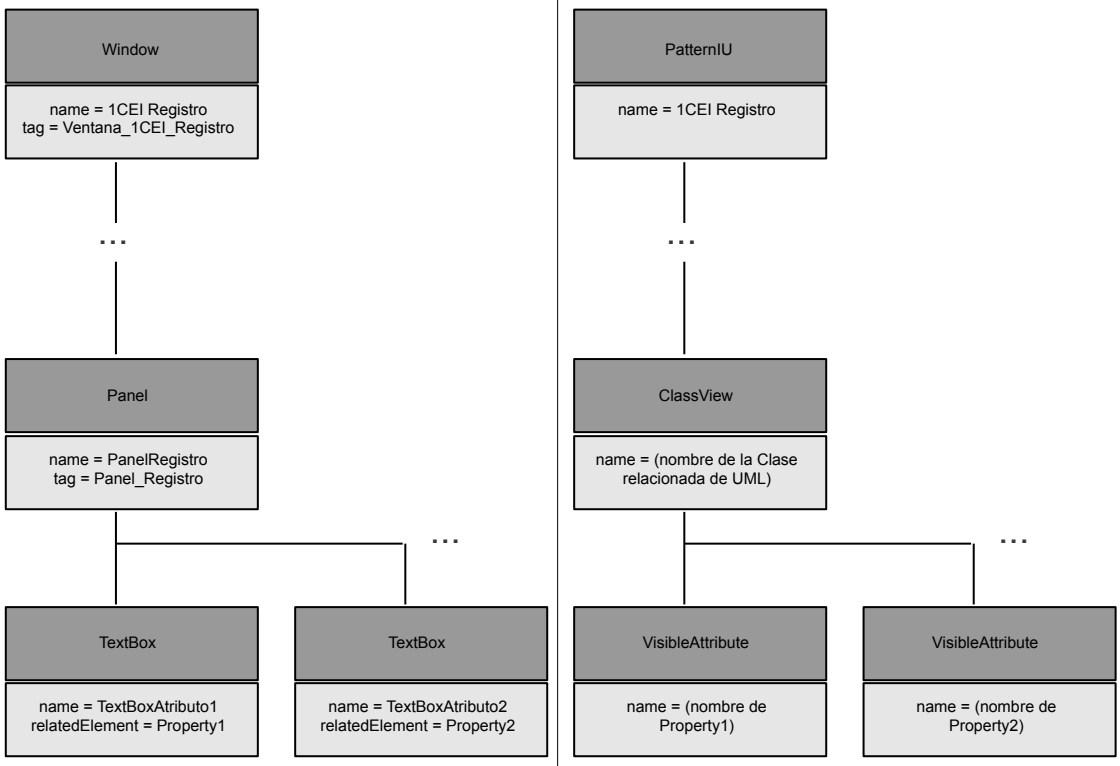
- Elemento *ClassView* con *name* = “ClassView1” → contiene 5 *visibleFeature* → 4 *VisibleAttribute* y 1 *VisibleOperation*

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter* (los nombres coinciden con las *Property* asociadas)

- Elemento *Operations* → contiene 2 *Operation*

El atributo *relatedProperty* de los *FilterParameter* apunta al *Property* relacionado

Identificador	Descripción
TFR-002-V02-001	Elementos *Property* procedentes de un modelo UML relacionados en los elementos *TextBox* del Panel de Registro
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**



Pruebas realizadas

- Elemento *PatternIU* con *name* = “1CEI Registro”

- Elemento *EditableInformationIU* con *name* = “Ventana Consulta-Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

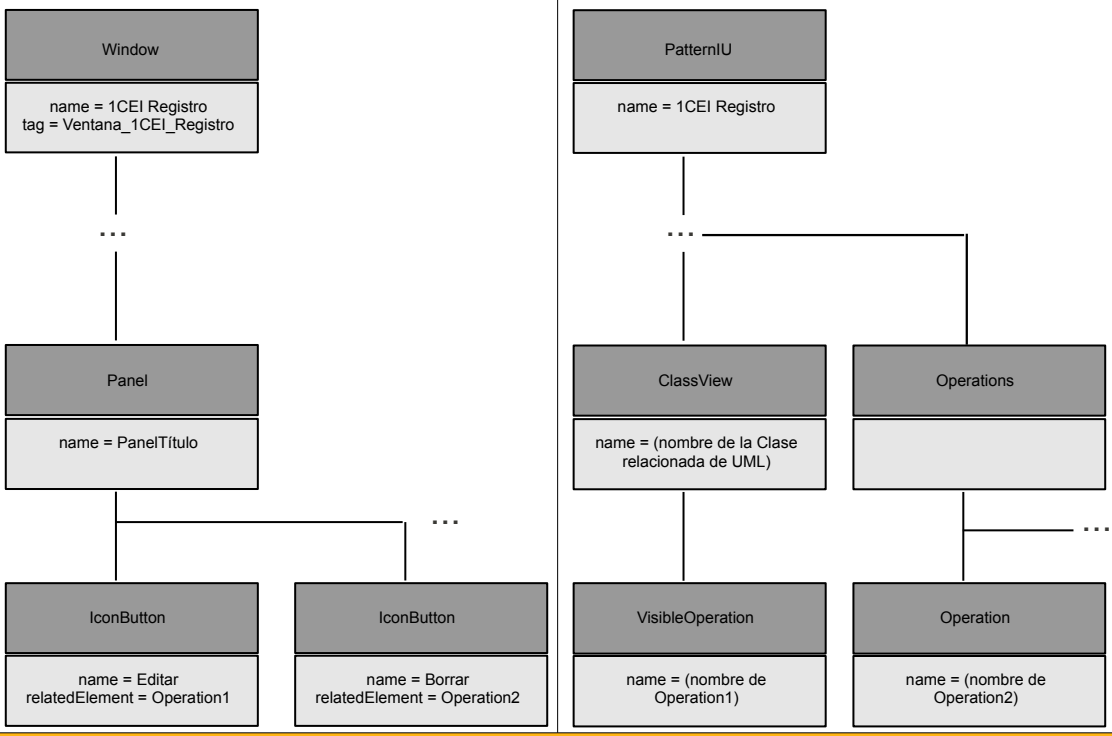
- Elemento ClassView con name = (nombre clase UML) → contiene 4 visibleFeature → 4 VisibleAttribute (nombre de las *Property* relacionadas)

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*

- Elemento *Operations* → contiene 2 *Operation*

El atributo *attribute* de los *VisibleAttribute* apunta al *Property* relacionado

Identificador	Descripción
TFR-002-V04-001	Elementos *Operation* procedentes de un modelo UML relacionados en los elementos *IconButton* del Panel Título
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**



Pruebas realizadas

- Elemento *PatternIU* con *name* = “1CEI Registro”

  - Elemento *EditableInformationIU* con *name* = “Ventana Consulta-Edición-Inserción”

    - Elemento *VisualizationSet* con *name* = “Ventana Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

      - Elemento *ClassView* con *name* = (nombre clase UML) → contiene 1 *visibleFeature* → 1 *VisibleOperation* (nombre de la *Operation* relacionada)

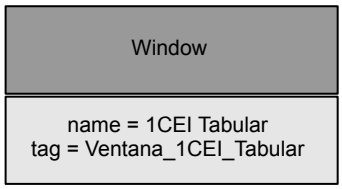
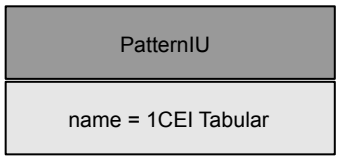
      - Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

      - Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*

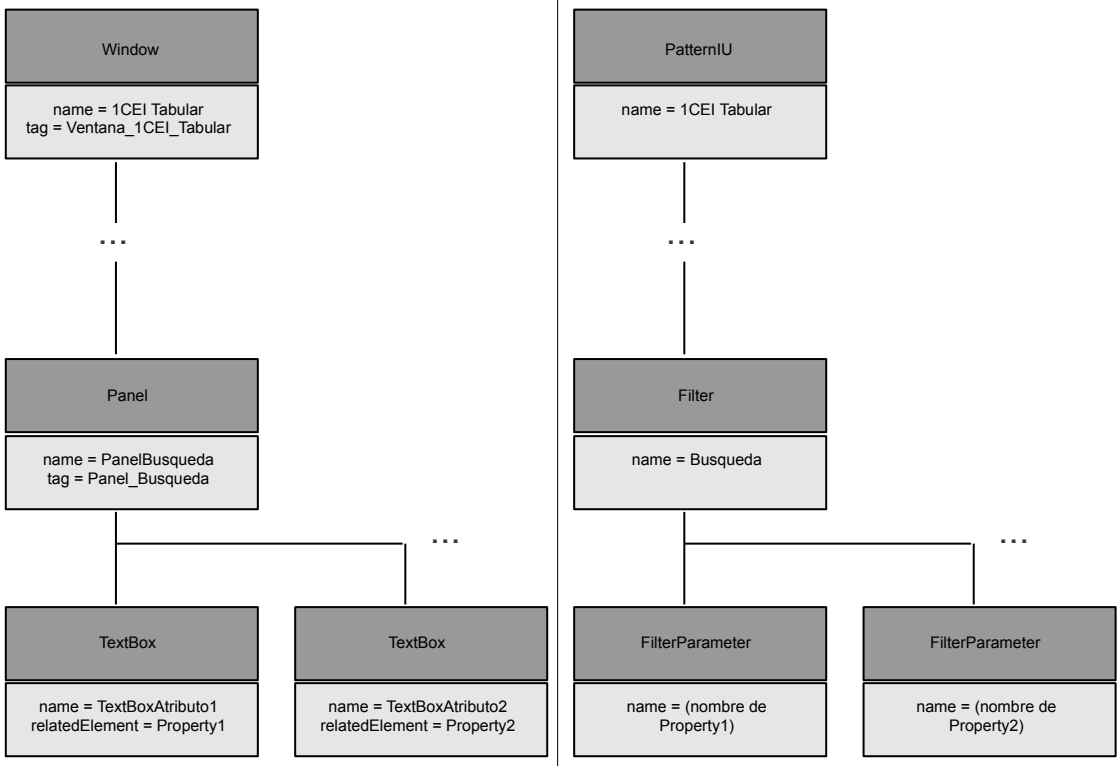
    - Elemento *Operations* → contiene 2 *Operation* (nombre de las *Operation* relacionadas)

El atributo *operation* de los *VisibleOperation* apunta al *Operation* relacionado

5.1.3. Patrón 1CEI Tabular

<table>
<tr><th>Identificador</th><th>Descripción</th></tr>
<tr><td>TFR-003-V00-001</td><td>Un elemento Window '1CEI Tabular' genera un elemento PatternIU</td></tr>
<tr><th>Severidad</th><th>Tipo</th></tr>
<tr><td>Bloqueante</td><td>Funcional</td></tr>
<tr><th>Datos de entrada</th><th>Datos de salida esperados</th></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><th colspan="2">Pruebas realizadas</th></tr>
<tr><td colspan="2">- Elemento PatternIU con name = “1CEI Tabular”
- Elemento EditableInformationIU con name = “Ventana Consulta-Edición-Inserción”
- Elemento VisualizationSet con name = “Ventana Consulta-Edición-Inserción” → contiene un visualizationPattern y un defaultVisualizationPattern de tipo Tabular
- Elemento ClassView con name = “ClassView1” → contiene 6 visibleFeature → 5 VisibleAttribute y 1 VisibleOperation
- Elemento Filter con name = “Búsqueda” → contiene 4 filterParameter
- Elemento Operations → contiene 2 Operation</td></tr>
</table>

Identificador	Descripción
TFR-003-V01-001	Elementos *Property* procedentes de un modelo UML relacionados en los elementos *TextBox* del Panel de Búsqueda
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**



Pruebas realizadas

- Elemento *PatternIU* con *name* = “1CEI Tabular”

- Elemento *EditableInformationIU* con *name* = “Ventana Consulta-Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

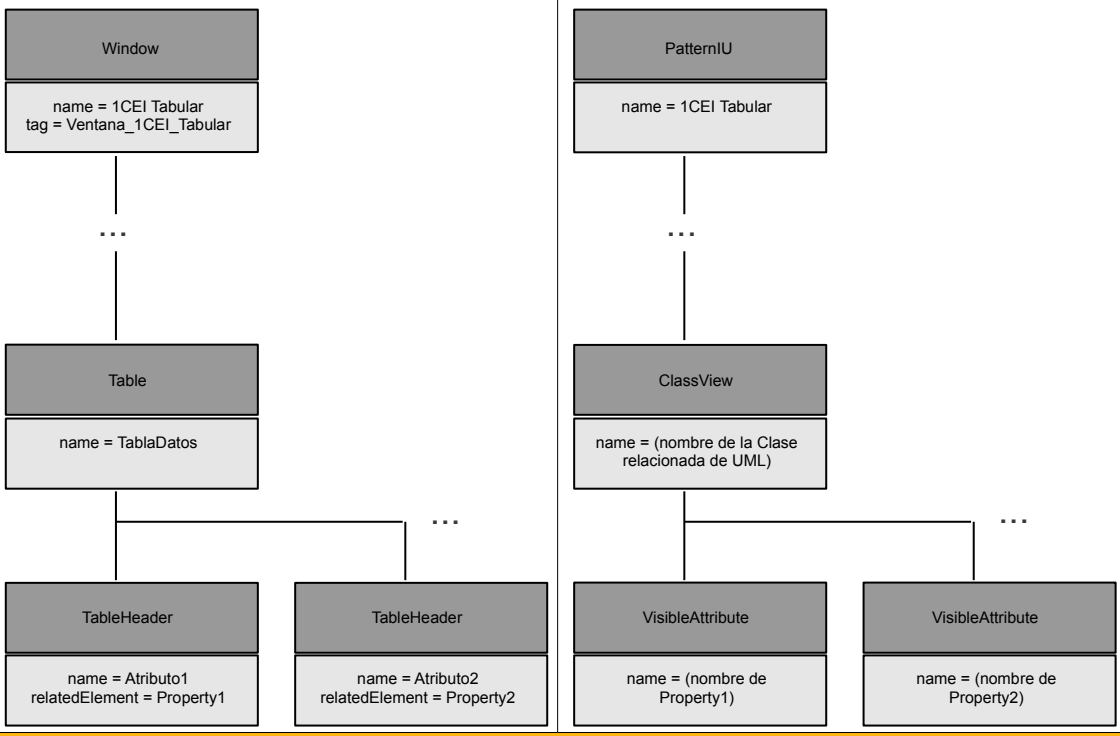
- Elemento *ClassView* con *name* = “ClassView1” → contiene 6 *visibleFeature* → 5 *VisibleAttribute* y 1 *VisibleOperation*

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter* (los nombres coinciden con las *Property* asociadas)

- Elemento *Operations* → contiene 2 *Operation*

El atributo *relatedProperty* de los *FilterParameter* apunta al *Property* relacionado

Identificador	Descripción
TFR-003-V03-001	Elementos *Property* procedentes de un modelo UML relacionados en los elementos *TableHeader* del Panel Tabular
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**



Pruebas realizadas

- Elemento *PatternIU* con *name* = “1CEI Tabular”

- Elemento *EditableInformationIU* con *name* = “Ventana Consulta-Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- Elemento *ClassView* con *name* = (nombre clase UML) → contiene 4 *visibleFeature* → 4 *VisibleAttribute* (nombre de las *Property* relacionadas)

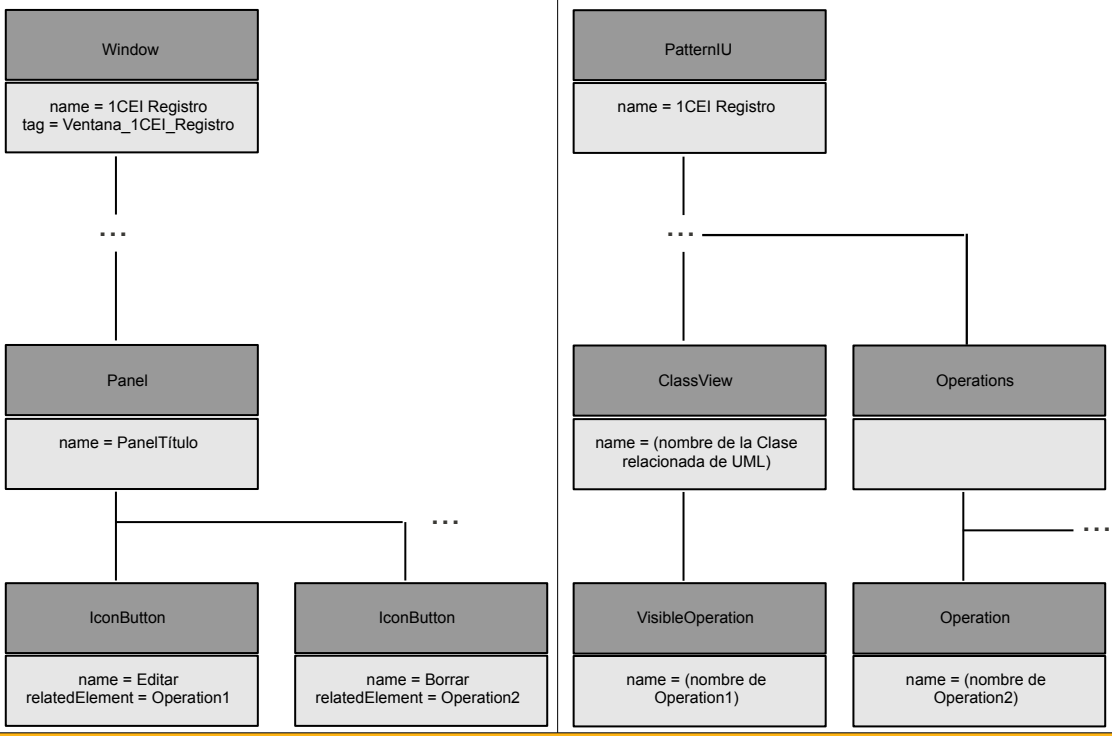
- Elemento *ClassView* con *name* = “ClassView1” → contiene 2 *visibleFeature* → 1 *VisibleAttribute* y 1 *VisibleOperation*

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*

- Elemento *Operations* → contiene 2 *Operation*

El atributo *attribute* de los *VisibleAttribute* apunta al *Property* relacionado

Identificador	Descripción
TFR-003-V04-001	Elementos *Operation* procedentes de un modelo UML relacionados en los elementos *IconButton* del Panel Título
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**



Pruebas realizadas

- Elemento *PatternIU* con *name* = “1CEI Registro”

  - Elemento *EditableInformationIU* con *name* = “Ventana Consulta-Edición-Inserción”

    - Elemento *VisualizationSet* con *name* = “Ventana Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

      - Elemento *ClassView* con *name* = (nombre clase UML) → contiene 1 *visibleFeature* → 1 *VisibleOperation* (nombre de la *Operation* relacionada)

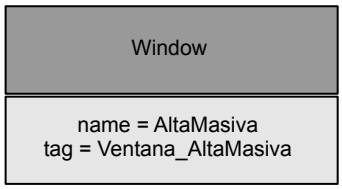
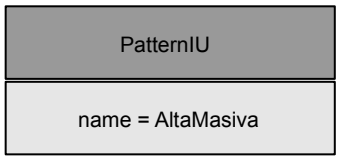
      - Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

      - Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*

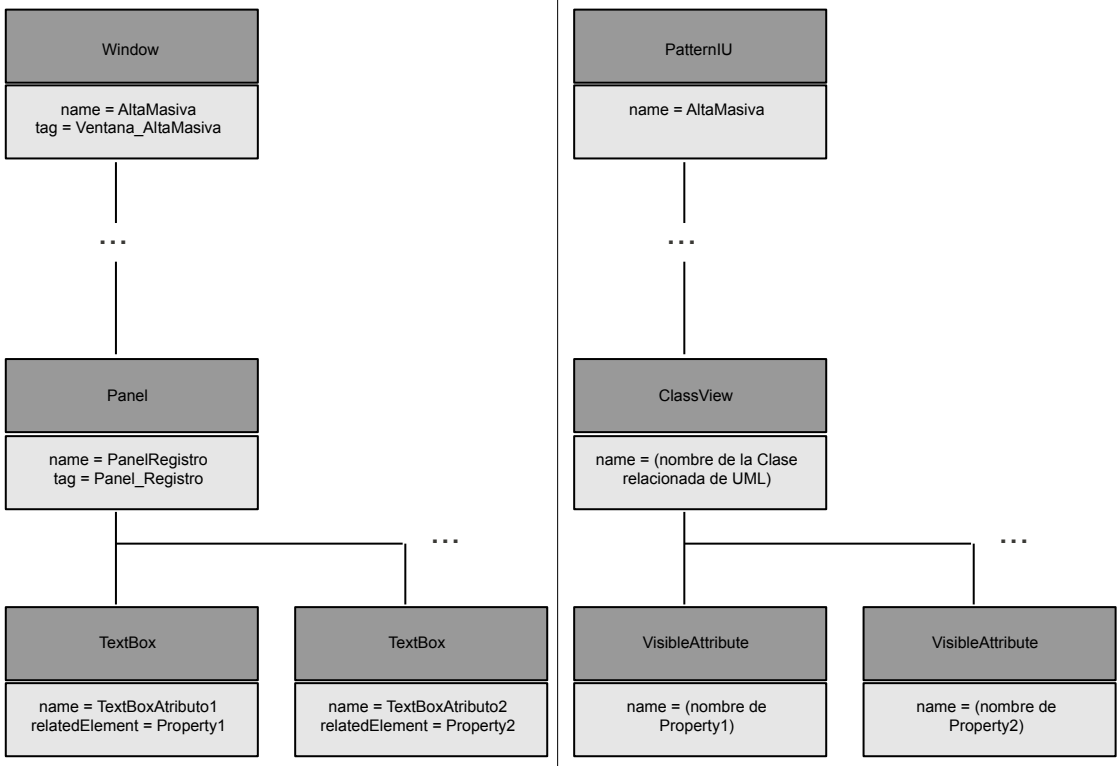
    - Elemento *Operations* → contiene 2 *Operation* (nombre de las *Operation* relacionadas)

El atributo *operation* de los *VisibleOperation* apunta al *Operation* relacionado

5.1.4. Patrón Alta Masiva

Identificador	Descripción
TFR-004-V00-001	Un elemento *Window* 'AltaMasiva' genera un elemento *PatternIU*
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**
	
Pruebas realizadas	
- Elemento *PatternIU* con *name* = “AltaMasiva” - Elemento *EditableInformationIU* con *name* = “Ventana Inserción” - Elemento *VisualizationSet* con *name* = “Ventana Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro - Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute* - Elemento *Operations* → contiene 1 *Operation*	

Identificador	Descripción
TFR-004-V02-001	Elementos *Property* procedentes de un modelo UML relacionados en los elementos *TextBox* del Panel de Registro
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**



Pruebas realizadas

- Elemento *PatternIU* con *name* = “AltaMasiva”

- Elemento *InformationIU* con *name* = “Ventana Inserción”

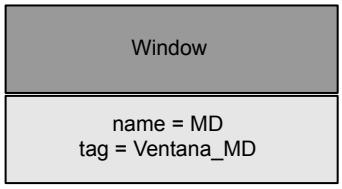
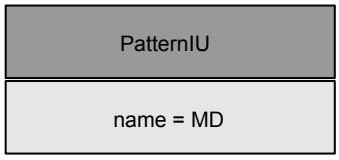
- Elemento *VisualizationSet* con *name* = “Ventana Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = (nombre clase UML) → contiene 4 *visibleFeature* → 4 *VisibleAttribute* (nombre de las *Property* relacionadas)

- Elemento *Operations* → contiene 1 *Operation*

El atributo *attribute* de los *VisibleAttribute* apunta al *Property* relacionado

5.1.5. Patrón MD

Identificador	Descripción
TFR-005-V00-001	Un elemento *Window* 'MD' genera dos elementos *PatternIU* (Maestro y Detalle) **Maestro 1CEI (Registro) – Detalle 1C (Tabular) 1EI (Registro)**
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**
	

Pruebas realizadas

- Elemento *PatternIU* con *name* = “MD”

- Elemento *Role* con *name* = “Maestro”

- Elemento *PatternIU* con *name* = “MD”

- Elemento *EditableInformationIU* con *name* = “Ventana Maestro Consulta-Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Maestro Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 5 *visibleFeature* → 4 *VisibleAttribute* y 1 *VisibleOperation*

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

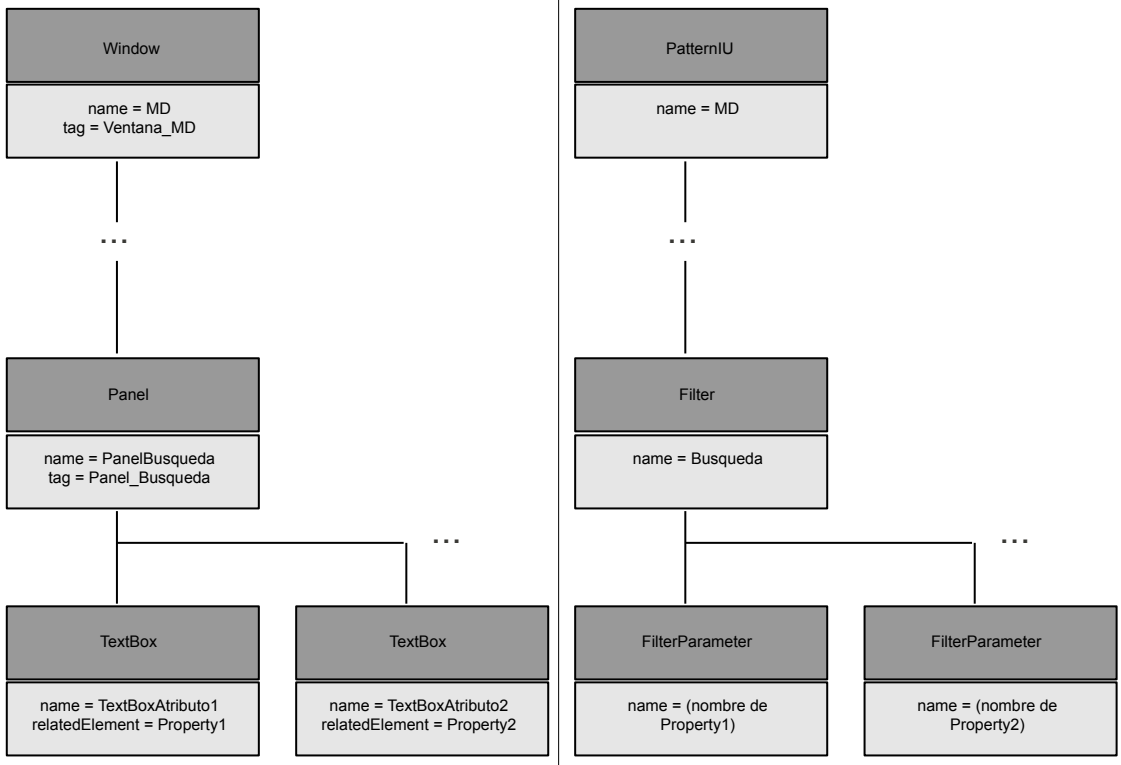
- Elemento *Operations* → contiene 2 *Operation*

- Elemento *InformationIU* con *name* = “Ventana Detalle Consulta”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- Elemento *ClassView* con *name* = “ClassView1” → contiene 6 *visibleFeature* → 5 *VisibleAttribute* y 1 *VisibleOperation*

Identificador	Descripción
TFR-005-V01-001	Elementos *Property* procedentes de un modelo UML relacionados en los elementos *TextBox* del Panel de Búsqueda del Maestro **Maestro 1CEI (Registro) – Detalle 1C (Tabular) 1EI (Registro)**
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**



Pruebas realizadas

- Elemento *PatternIU* con *name* = “MD”

- Elemento *Role* con *name* = “Maestro”

- Elemento *PatternIU* con *name* = “MD”

- Elemento *EditableInformationIU* con *name* = “Ventana Maestro Consulta-Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Maestro Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 5 *visibleFeature* → 4 *VisibleAttribute* y 1 *VisibleOperation*

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter* (los nombres coinciden con las *Property* asociadas)

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

- Elemento *Operations* → contiene 2 *Operation*

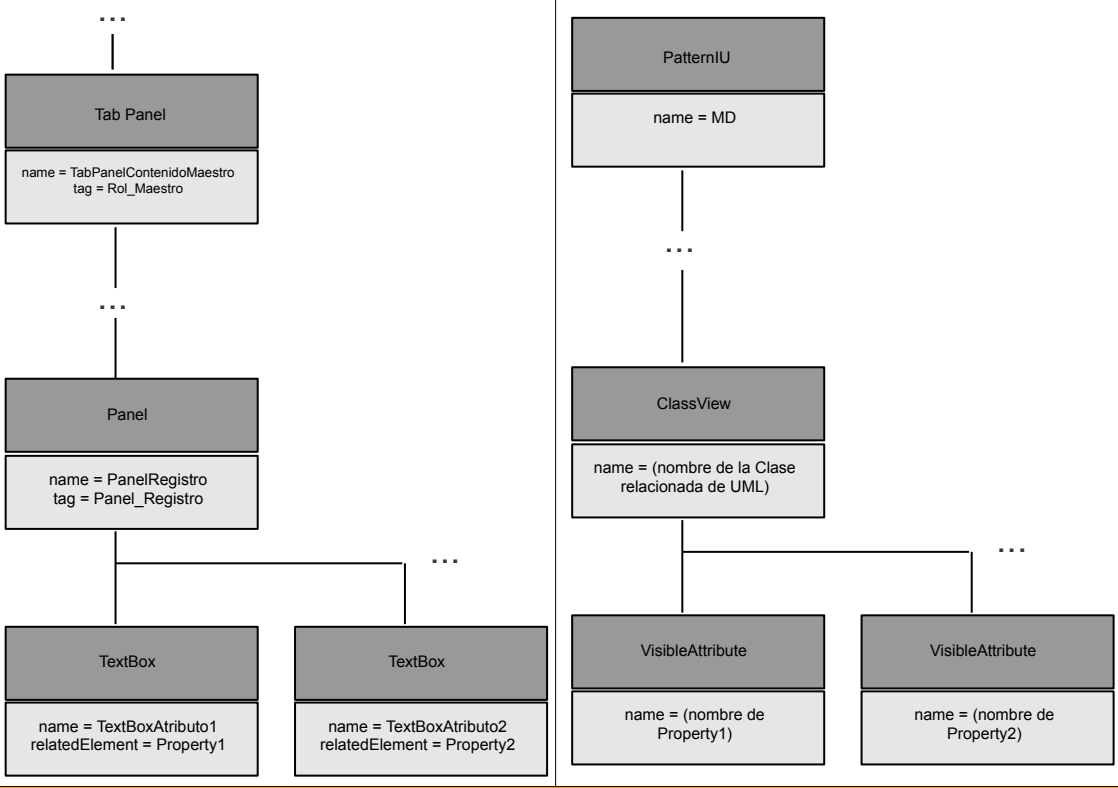
- Elemento *InformationIU* con *name* = “Ventana Detalle Consulta”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- Elemento *ClassView* con *name* = “ClassView1” → contiene 6 *visibleFeature* → 5 *VisibleAttribute* y 1 *VisibleOperation*

El atributo *relatedProperty* de los *FilterParameter* apunta al *Property* relacionado

Identificador	Descripción
TFR-005-V02-001	Elementos *Property* procedentes de un modelo UML relacionados en los elementos *TextBox* del Panel de Registro del Maestro **Maestro 1CEI (Registro) – Detalle 1C (Tabular) 1EI (Registro)**
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**



Pruebas realizadas

- Elemento *PatternIU* con *name* = “MD”

- Elemento *Role* con *name* = “Maestro”

- Elemento *PatternIU* con *name* = “MD”

- Elemento *EditableInformationIU* con *name* = “Ventana Maestro Consulta-Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Maestro Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- *Elemento ClassView con name = (nombre clase UML) → contiene 4 visibleFeature → 4 VisibleAttribute* (nombre de las *Property* relacionadas)

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

- Elemento *Operations* → contiene 2 *Operation*

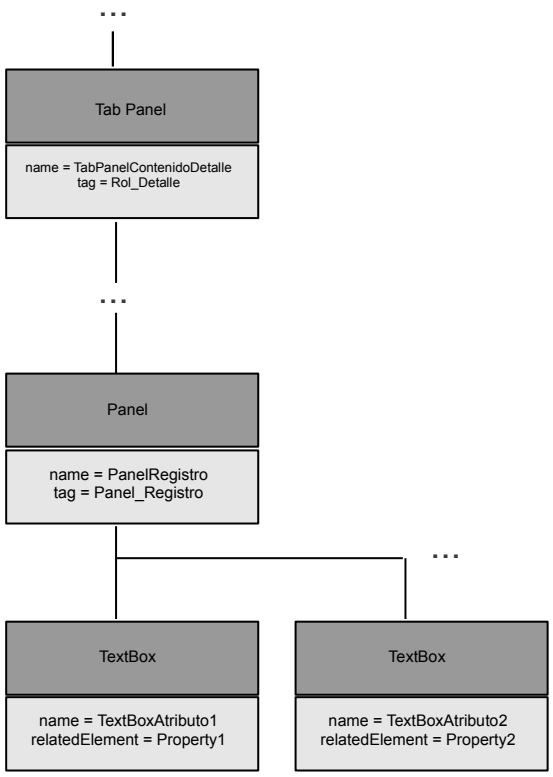
- Elemento *InformationIU* con *name* = “Ventana Detalle Consulta”

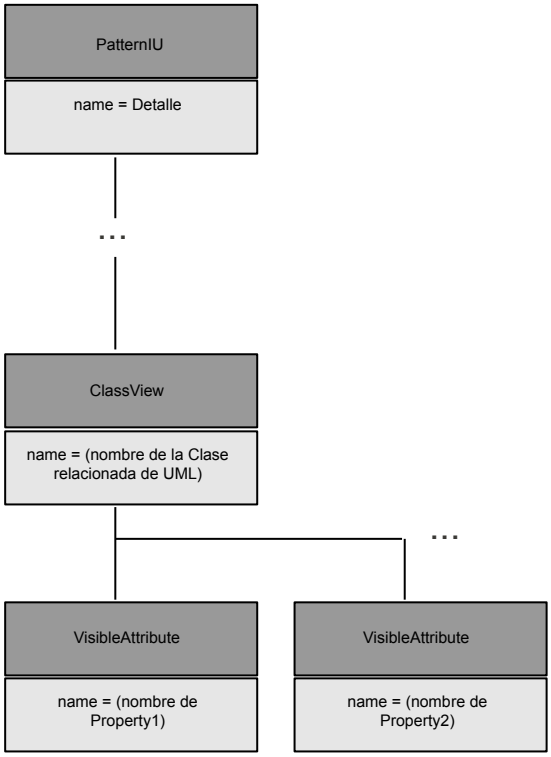
- Elemento *VisualizationSet* con *name* = “Ventana Detalle Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- Elemento *ClassView* con *name* = “ClassView1” → contiene 6 *visibleFeature* → 5 *VisibleAttribute* y 1 *VisibleOperation*

El atributo *attribute* de los *VisibleAttribute* apunta al *Property* relacionado

Identificador	Descripción
TFR-005-V02-002	Elementos *Property* procedentes de un modelo UML relacionados en los elementos *TextBox* del Panel de Registro del Detalle **Maestro 1CEI (Registro) – Detalle 1C (Tabular) 1EI (Registro)**
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**





Pruebas realizadas

- Elemento *PatternIU* con *name* = “MD”
  - Elemento *Role* con *name* = “Maestro”
    - Elemento *PatternIU* con *name* = “MD”
      - Elemento *EditableInformationIU* con *name* = “Ventana Maestro Consulta-Edición-Inserción”
        - Elemento *VisualizationSet* con *name* = “Ventana Maestro Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro
          - *Elemento ClassView con name* = “ClassView1” → *contiene 4 visibleFeature → 4 VisibleAttribute*
          - Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*
        - Elemento *Operations* → contiene 2 *Operation*

- Elemento *Role* con *name* = “Detalle”
  - Elemento *PatternIU* con *name* = “Detalle”
    - Elemento *EditableInformationIU* con *name* = “Ventana Detalle Edición-Inserción”
      - Elemento *VisualizationSet* con *name* = “Ventana Detalle Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = (nombre clase UML) → contiene 4 *visibleFeature* → 4 *VisibleAttribute* (nombre de las *Property* relacionadas)

- Elemento *Operations* → contiene 2 *Operation*

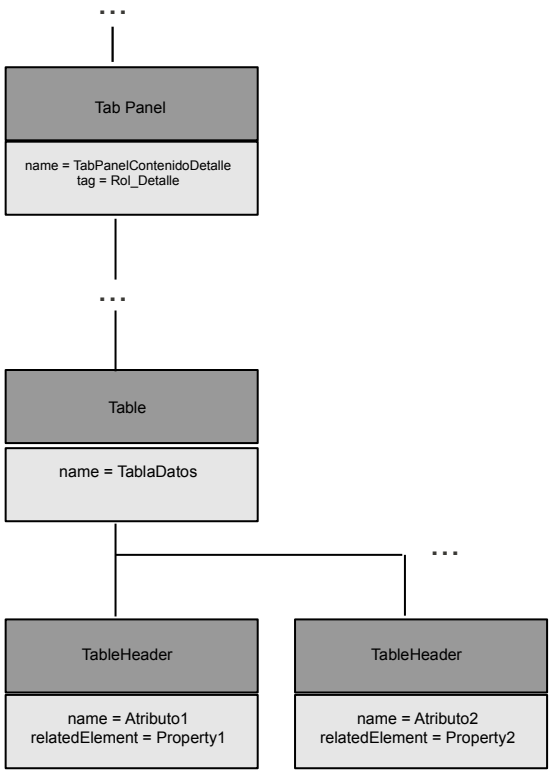
- Elemento *InformationIU* con *name* = “Ventana Detalle Consulta”

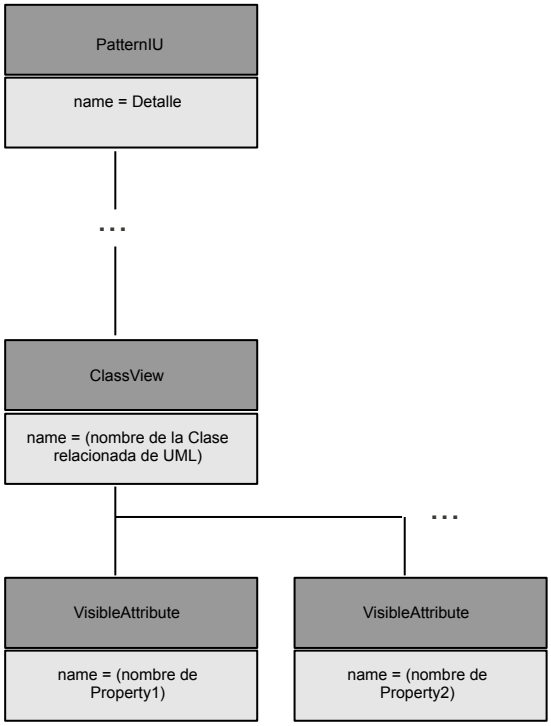
- Elemento *VisualizationSet* con *name* = “Ventana Detalle Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- Elemento *ClassView* con *name* = “ClassView1” → contiene 6 *visibleFeature* → 5 *VisibleAttribute* y 1 *VisibleOperation*

El atributo *attribute* de los *VisibleAttribute* apunta al *Property* relacionado

Identificador	Descripción
TFR-005-V03-001	Elementos *Property* procedentes de un modelo UML relacionados en los elementos *TableHeader* del Panel Tabular del Detalle **Maestro 1CEI (Registro) – Detalle 1C (Tabular) 1EI (Registro)**
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**





Pruebas realizadas

- Elemento *PatternIU* con *name* = “MD”

- Elemento *Role* con *name* = “Maestro”

- Elemento *PatternIU* con *name* = “MD”

- Elemento *EditableInformationIU* con *name* = “Ventana Maestro Consulta-Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Maestro Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- *Elemento ClassView con name* = “ClassView1” → *contiene 5 visibleFeature → 4 VisibleAttribute* y 1 *VisibleOperation*

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *InformationIU* con *name* = “Ventana Detalle Consulta”

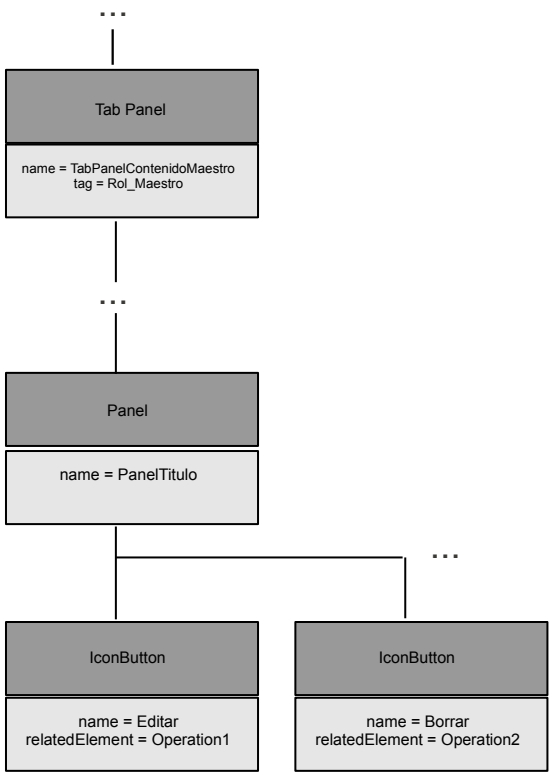
- Elemento *VisualizationSet* con *name* = “Ventana Detalle Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

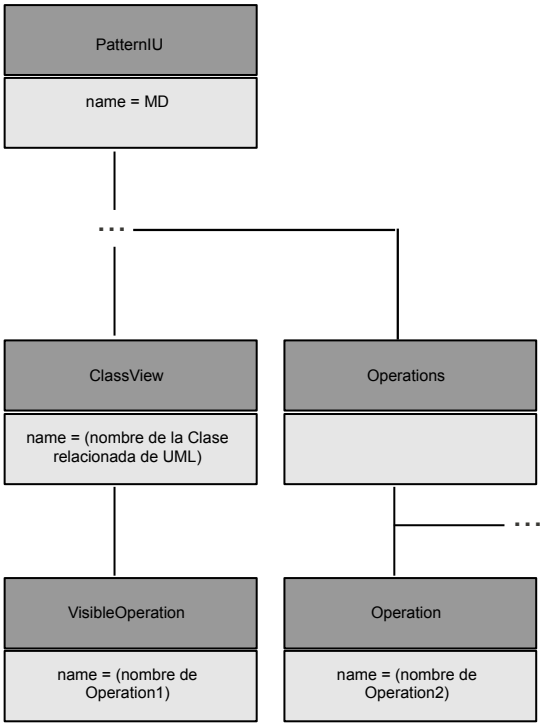
- *Elemento ClassView con name = (nombre clase UML) → contiene 4 visibleFeature → 4 VisibleAttribute* (nombre de las *Property* relacionadas)

- Elemento *ClassView* con *name* = “ClassView1” → contiene 1 *visibleFeature* → 1 *VisibleOperation*

El atributo *attribute* de los *VisibleAttribute* apunta al *Property* relacionado

Identificador	Descripción
TFR-005-V04-001	Elementos *Operation* procedentes de un modelo UML relacionados en los elementos *IconButton* del Panel Título del Maestro **Maestro 1CEI (Registro) – Detalle 1C (Tabular) 1EI (Registro)**
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**





Pruebas realizadas

- Elemento *PatternIU* con *name* = “MD”

- Elemento *Role* con *name* = “Maestro”

- Elemento *PatternIU* con *name* = “MD”

- Elemento *EditableInformationIU* con *name* = “Ventana Maestro Consulta-Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Maestro Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- *Elemento ClassView con name* = “ClassView1” → *contiene 4 visibleFeature → 4 VisibleAttribute*

- Elemento *ClassView* con *name* = (nombre clase UML) → contiene 1 *visibleFeature* → 1 *VisibleOperation* (nombre de la *Operation* relacionada)

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*

- Elemento *Operations* → contiene 2 *Operation* (nombre de las *Operation* relacionadas)

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle Edición-Inserción” → contiene un *visualizationPattern* y

un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

- Elemento *Operations* → contiene 2 *Operation*

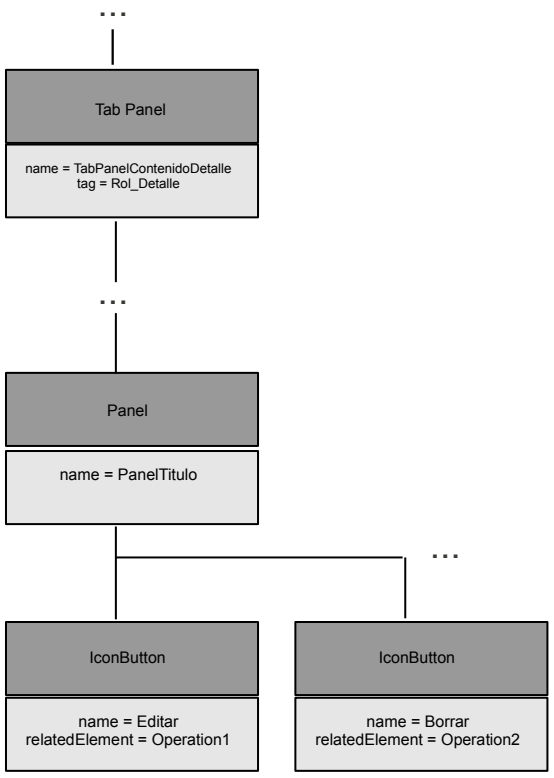
- Elemento *InformationIU* con *name* = “Ventana Detalle Consulta”

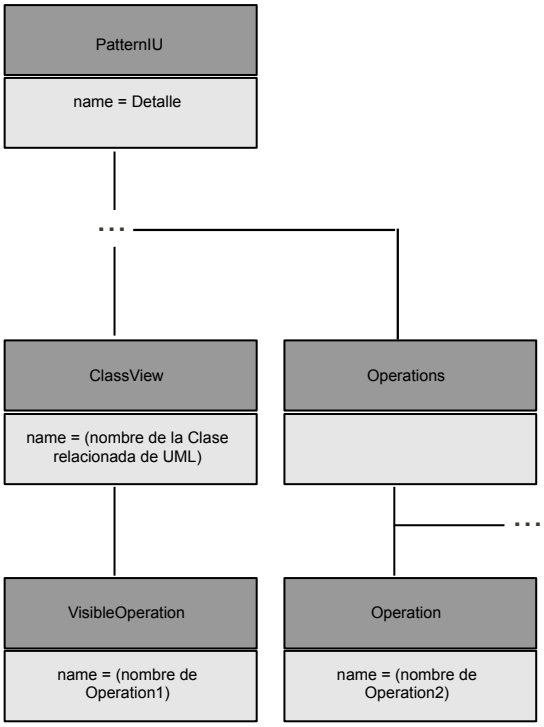
- Elemento *VisualizationSet* con *name* = “Ventana Detalle Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- *Elemento ClassView con name* = “ClassView1” → *contiene 6 visibleFeature → 5 VisibleAttribute* y 1 *VisibleOperation*

El atributo *operation* de los *VisibleOperation* apunta al *Operation* relacionado

Identificador	Descripción
TFR-005-V04-002	Elementos *Operation* procedentes de un modelo UML relacionados en los elementos *IconButton* del Panel Título del Detalle **Maestro 1CEI (Registro) – Detalle 1C (Tabular) 1EI (Registro)**
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**





Pruebas realizadas

- Elemento *PatternIU* con *name* = “MD”

- Elemento *Role* con *name* = “Maestro”

- Elemento *PatternIU* con *name* = “MD”

- Elemento *EditableInformationIU* con *name* = “Ventana Maestro Consulta-Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Maestro Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- *Elemento ClassView con name* = “ClassView1” → *contiene 5 visibleFeature → 4 VisibleAttribute* y 1 *VisibleOperation*

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

- Elemento *Operations* → contiene 2 *Operation* (nombre de las *Operation* relacionadas)

- Elemento *InformationIU* con *name* = “Ventana Detalle Consulta”

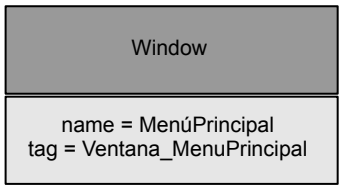
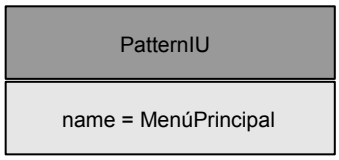
- Elemento *VisualizationSet* con *name* = “Ventana Detalle Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- *Elemento ClassView con name* = “ClassView1” → *contiene 5 visibleFeature → 5 VisibleAttribute* y

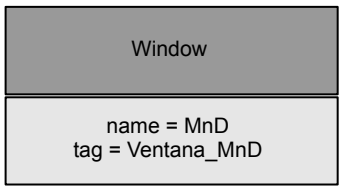
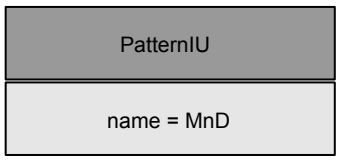
- Elemento *ClassView* con *name* = (nombre clase UML) → contiene 1 *visibleFeature* → 1 *VisibleOperation* (nombre de la *Operation* relacionada)

El atributo *operation* de los *VisibleOperation* apunta al *Operation* relacionado

5.1.6. Patrón MenúPrincipal

Identificador	Descripción
TFR-006-V00-001	Un elemento *Window* 'MenúPrincipal' genera un elemento *PatternIU*
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**
	
Pruebas realizadas	
- Elemento *PatternIU* con *name* = “MenúPrincipal” - Elemento *NavigationIU* con *name* = “MenúPrincipal” - Elemento *NavigationIU* con *name* = “Menu_AAdministracionSistema” (*visualizationOrder* 2) - Elemento *NavigationIU* con *name* = “Menu_AHerramientasAuxiliares” (*visualizationOrder* 1) - Elemento *NavigationIU* con *name* = “Menu_AModulosPrincipales” (*visualizationOrder* 0) → contiene 3 *Navigation*	

5.1.7. Patrón MnD

Identificador	Descripción
TFR-007-V00-001	Un elemento *Window* 'MnD' genera *n*+1 elementos *PatternIU* (Maestro y *n* Detalles) **Maestro 1CEI (Registro) – N Detalle 1C (Tabular) 1EI (Registro)** **N = 2**
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**
	

Pruebas realizadas

- Elemento *PatternIU* con *name* = “MnD”

- Elemento *Role* con *name* = “Maestro”

- Elemento *PatternIU* con *name* = “MnD”

- Elemento *EditableInformationIU* con *name* = “Ventana Maestro Consulta-Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Maestro Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 5 *visibleFeature* → 4 *VisibleAttribute* y 1 *VisibleOperation*

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle 1”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle 1 Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 1 Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *InformationIU* con *name* = “Ventana Detalle 1 Consulta”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 1 Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- Elemento *ClassView* con *name* = “ClassView1” → contiene 6 *visibleFeature* → 5 *VisibleAttribute* y 1 *VisibleOperation*

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle 2”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle 2 Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 2 Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

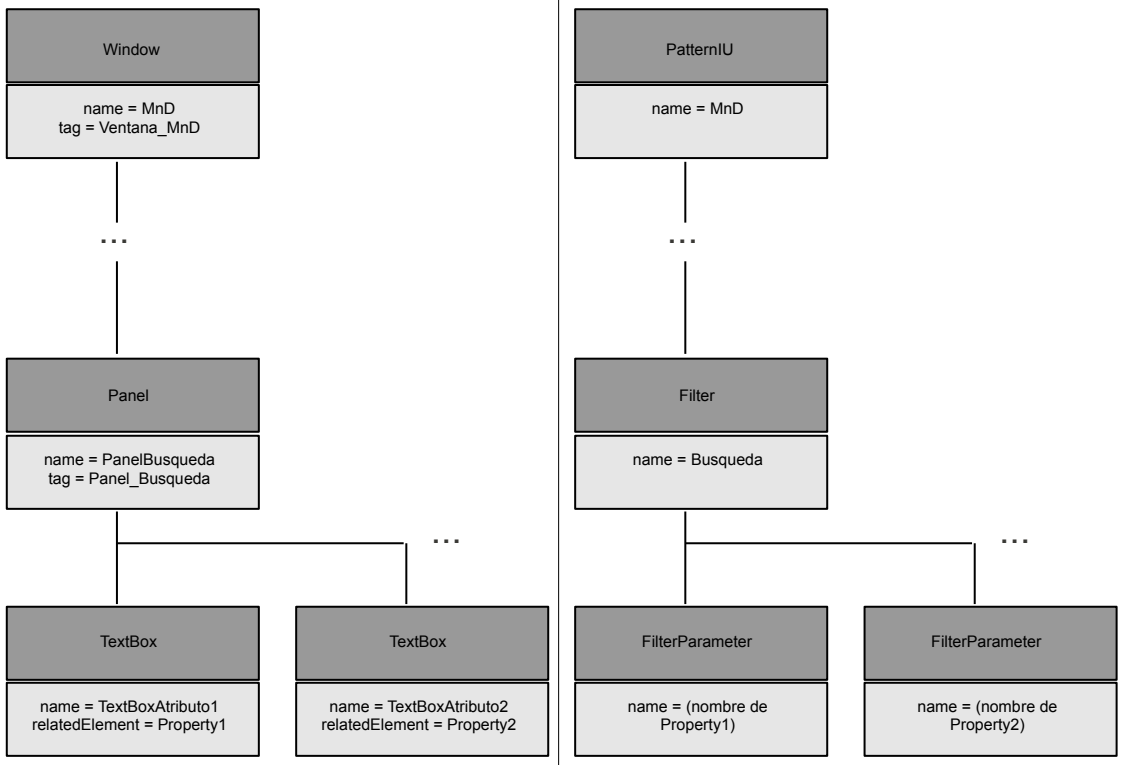
- Elemento *Operations* → contiene 2 *Operation*

- Elemento *InformationIU* con *name* = “Ventana Detalle 2 Consulta”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 2 Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- *Elemento ClassView con name = “ClassView1” → contiene 6 visibleFeature → 5 VisibleAttribute* y 1 *VisibleOperation*

Identificador	Descripción
TFR-007-V01-001	Elementos *Property* procedentes de un modelo UML relacionados en los elementos *TextBox* del Panel de Búsqueda del Maestro **Maestro 1CEI (Registro) – N Detalle 1C (Tabular) 1EI (Registro)** **N = 2**
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**



Pruebas realizadas

- Elemento *PatternIU* con *name* = “MnD”

- Elemento *Role* con *name* = “Maestro”

- Elemento *PatternIU* con *name* = “MnD”

- Elemento *EditableInformationIU* con *name* = “Ventana Maestro Consulta-Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Maestro Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 5 *visibleFeature* → 4 *VisibleAttribute* y 1 *VisibleOperation*

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter* (los nombres coinciden con las *Property* asociadas)

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle 1”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle 1 Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 1 Edición-Inserción” → contiene un *visualizationPattern*

y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *InformationIU* con *name* = “Ventana Detalle 1 Consulta”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 1 Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- Elemento *ClassView* con *name* = “ClassView1” → contiene 6 *visibleFeature* → 5 *VisibleAttribute* y 1 *VisibleOperation*

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle 2”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle 2 Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 2 Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

- Elemento *Operations* → contiene 2 *Operation*

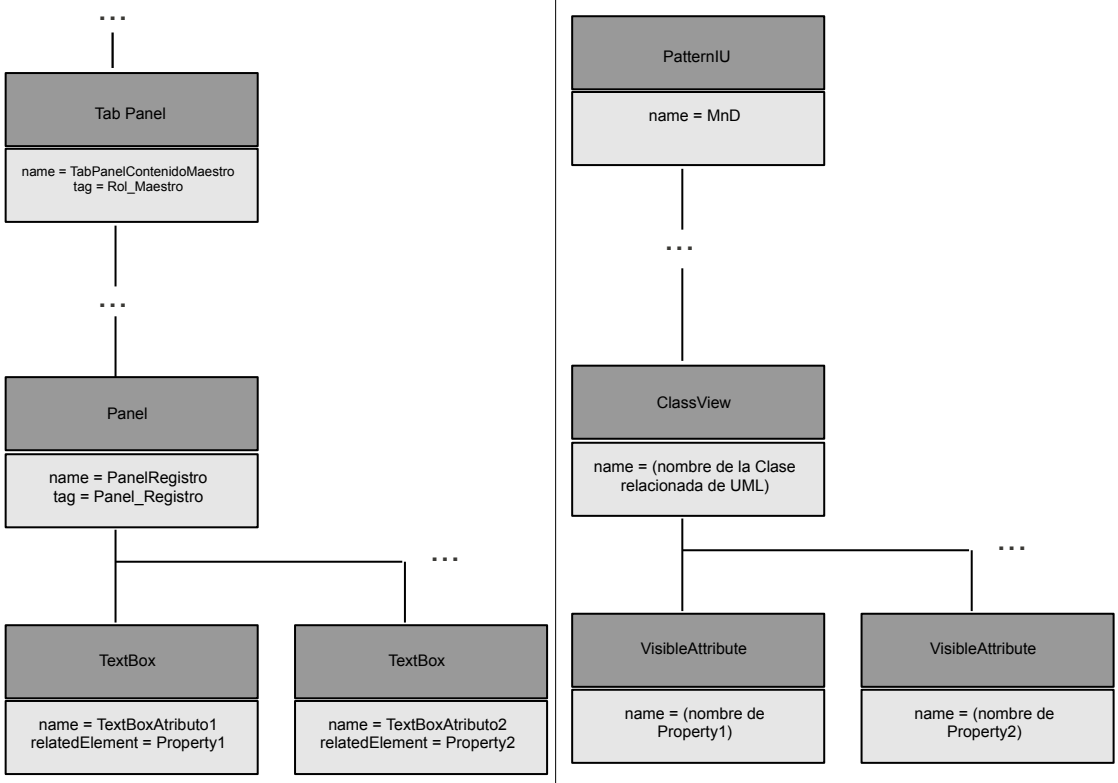
- Elemento *InformationIU* con *name* = “Ventana Detalle 2 Consulta”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 2 Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- Elemento *ClassView* con *name* = “ClassView1” → contiene 6 *visibleFeature* → 5 *VisibleAttribute* y 1 *VisibleOperation*

El atributo *relatedProperty* de los *FilterParameter* apunta al *Property* relacionado

Identificador	Descripción
TFR-007-V02-001	Elementos *Property* procedentes de un modelo UML relacionados en los elementos *TextBox* del Panel de Registro del Maestro **Maestro 1CEI (Registro) – N Detalle 1C (Tabular) 1EI (Registro)** **N = 2**
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**



Pruebas realizadas

- Elemento *PatternIU* con *name* = “MnD”

- Elemento *Role* con *name* = “Maestro”

- Elemento *PatternIU* con *name* = “MnD”

- Elemento *EditableInformationIU* con *name* = “Ventana Maestro Consulta-Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Maestro Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- *Elemento ClassView con name = (nombre clase UML) → contiene 4 visibleFeature → 4 VisibleAttribute* (nombre de las *Property* relacionadas)

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle 1”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle 1 Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 1 Edición-Inserción” → contiene un *visualizationPattern*

y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *InformationIU* con *name* = “Ventana Detalle 1 Consulta”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 1 Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- Elemento *ClassView* con *name* = “ClassView1” → contiene 6 *visibleFeature* → 5 *VisibleAttribute* y 1 *VisibleOperation*

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle 2”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle 2 Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 2 Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

- Elemento *Operations* → contiene 2 *Operation*

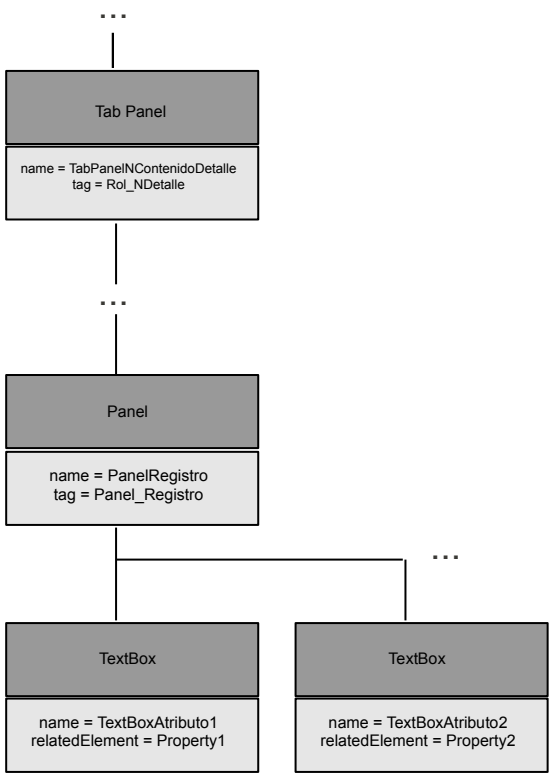
- Elemento *InformationIU* con *name* = “Ventana Detalle 2 Consulta”

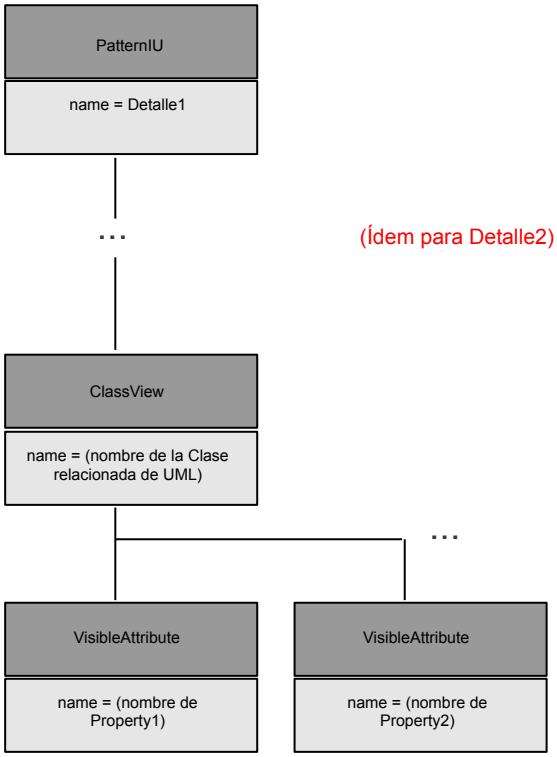
- Elemento *VisualizationSet* con *name* = “Ventana Detalle 2 Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- Elemento *ClassView* con *name* = “ClassView1” → contiene 6 *visibleFeature* → 5 *VisibleAttribute* y 1 *VisibleOperation*

El atributo *attribute* de los *VisibleAttribute* apunta al *Property* relacionado

Identificador	Descripción
TFR-007-V02-002	Elementos *Property* procedentes de un modelo UML relacionados en los elementos *TextBox* del Panel de Registro de los N Detalles **Maestro 1CEI (Registro) – N Detalle 1C (Tabular) 1EI (Registro)** **N = 2**
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**





Pruebas realizadas

- Elemento *PatternIU* con *name* = “MnD”
  - Elemento *Role* con *name* = “Maestro”
    - Elemento *PatternIU* con *name* = “MnD”
      - Elemento *EditableInformationIU* con *name* = “Ventana Maestro Consulta-Edición-Inserción”
        - Elemento *VisualizationSet* con *name* = “Ventana Maestro Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro
          - *Elemento ClassView con name* = “ClassView1” → *contiene 4 visibleFeature → 4 VisibleAttribute*
          - Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*
        - Elemento *Operations* → contiene 2 *Operation*

- Elemento *Role* con *name* = “Detalle”
  - Elemento *PatternIU* con *name* = “Detalle 1”
    - Elemento *EditableInformationIU* con *name* = “Ventana Detalle 1 Edición-Inserción”
      - Elemento *VisualizationSet* con *name* = “Ventana Detalle 1 Edición-Inserción” → contiene un *visualizationPattern*

y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = (nombre clase UML) → contiene 4 *visibleFeature* → 4 *VisibleAttribute* (nombre de las *Property* relacionadas)

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *InformationIU* con *name* = “Ventana Detalle 1 Consulta”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 1 Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- Elemento *ClassView* con *name* = “ClassView1” → contiene 6 *visibleFeature* → 5 *VisibleAttribute* y 1 *VisibleOperation*

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle 2”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle 2 Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 2 Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = (nombre clase UML) → contiene 4 *visibleFeature* → 4 *VisibleAttribute* (nombre de las *Property* relacionadas)

- Elemento *Operations* → contiene 2 *Operation*

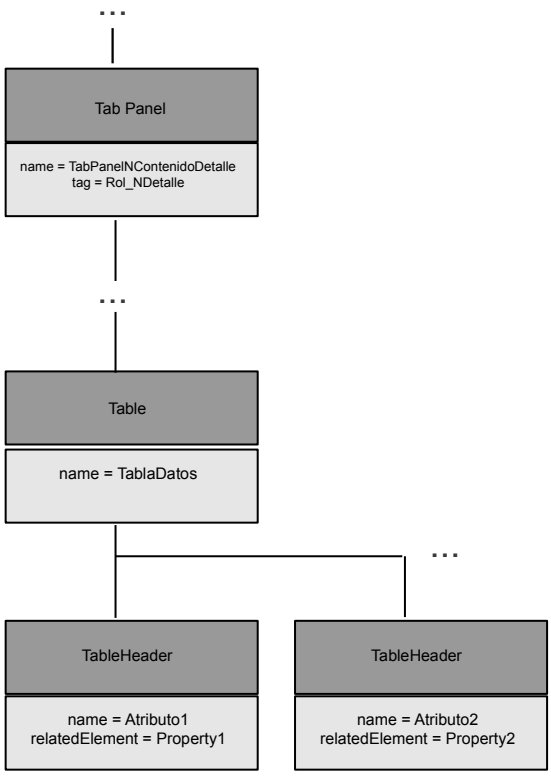
- Elemento *InformationIU* con *name* = “Ventana Detalle 2 Consulta”

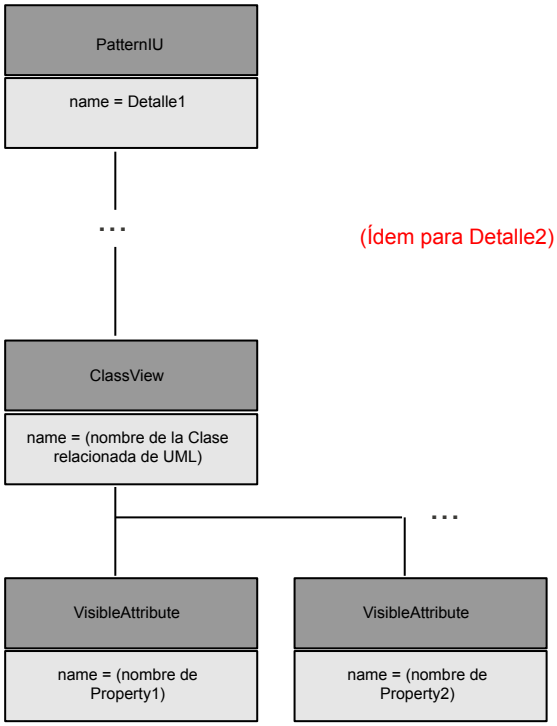
- Elemento *VisualizationSet* con *name* = “Ventana Detalle 2 Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- Elemento *ClassView* con *name* = “ClassView1” → contiene 6 *visibleFeature* → 5 *VisibleAttribute* y 1 *VisibleOperation*

El atributo *attribute* de los *VisibleAttribute* apunta al *Property* relacionado

Identificador	Descripción
TFR-007-V03-001	Elementos *Property* procedentes de un modelo UML relacionados en los elementos *TableHeader* del Panel Tabular de los N Detalles **Maestro 1CEI (Registro) – N Detalle 1C (Tabular) 1EI (Registro)** **N = 2**
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**





Pruebas realizadas

- Elemento *PatternIU* con *name* = “MnD”

- Elemento *Role* con *name* = “Maestro”

- Elemento *PatternIU* con *name* = “MnD”

- Elemento *EditableInformationIU* con *name* = “Ventana Maestro Consulta-Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Maestro Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- *Elemento ClassView con name* = “ClassView1” → *contiene 5 visibleFeature → 4 VisibleAttribute* y 1 *VisibleOperation*

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle 1”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle 1 Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 1 Edición-Inserción” → contiene un *visualizationPattern*

y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *InformationIU* con *name* = “Ventana Detalle 1 Consulta”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 1 Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- *Elemento ClassView con name = (nombre clase UML) → contiene 4 visibleFeature → 4 VisibleAttribute* (nombre de las *Property* relacionadas)

- Elemento *ClassView* con *name* = “ClassView1” → contiene 1 *visibleFeature* → 1 *VisibleOperation*

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle 2”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle 2 Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 2 Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *InformationIU* con *name* = “Ventana Detalle 2 Consulta”

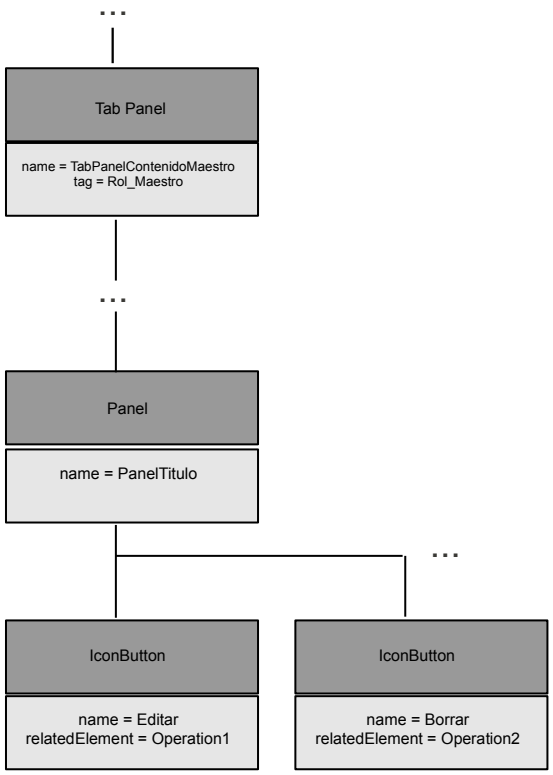
- Elemento *VisualizationSet* con *name* = “Ventana Detalle 2 Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

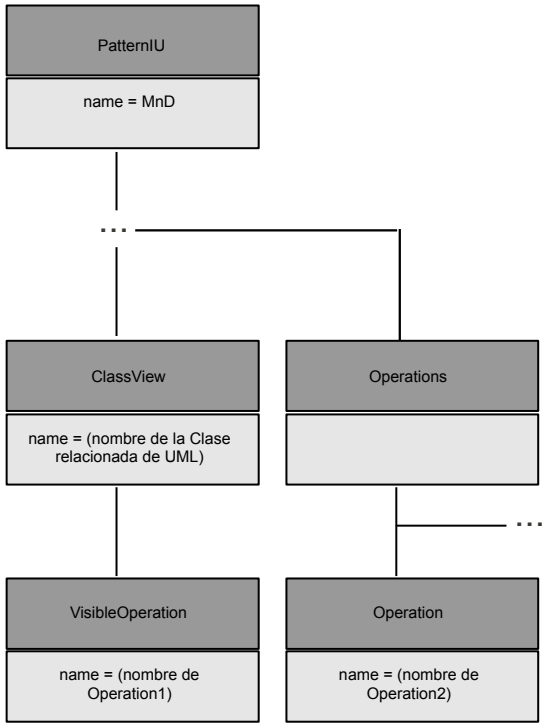
- *Elemento ClassView con name = (nombre clase UML) → contiene 4 visibleFeature → 4 VisibleAttribute* (nombre de las *Property* relacionadas)

- Elemento *ClassView* con *name* = “ClassView1” → contiene 1 *visibleFeature* → 1 *VisibleOperation*

El atributo *attribute* de los *VisibleAttribute* apunta al *Property* relacionado

Identificador	Descripción
TFR-007-V04-001	Elementos *Operation* procedentes de un modelo UML relacionados en los elementos *IconButton* del Panel Título del Maestro **Maestro 1CEI (Registro) – N Detalle 1C (Tabular) 1EI (Registro)** **N = 2**
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**





Pruebas realizadas

- Elemento *PatternIU* con *name* = “MnD”

- Elemento *Role* con *name* = “Maestro”

- Elemento *PatternIU* con *name* = “MnD”

- Elemento *EditableInformationIU* con *name* = “Ventana Maestro Consulta-Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Maestro Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- *Elemento ClassView con name* = “ClassView1” → *contiene 4 visibleFeature → 4 VisibleAttribute*

- Elemento *ClassView* con *name* = (nombre clase UML) → contiene 1 *visibleFeature* → 1 *VisibleOperation* (nombre de la *Operation* relacionada)

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*

- Elemento *Operations* → contiene 2 *Operation* (nombre de las *Operation* relacionadas)

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle 1”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 1 Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *InformationIU* con *name* = “Ventana Detalle 1 Consulta”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 1 Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- *Elemento ClassView con name* = “ClassView1” → *contiene 6 visibleFeature → 5 VisibleAttribute* y 1 *VisibleOperation*

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle 2”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 1 Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

- Elemento *Operations* → contiene 2 *Operation*

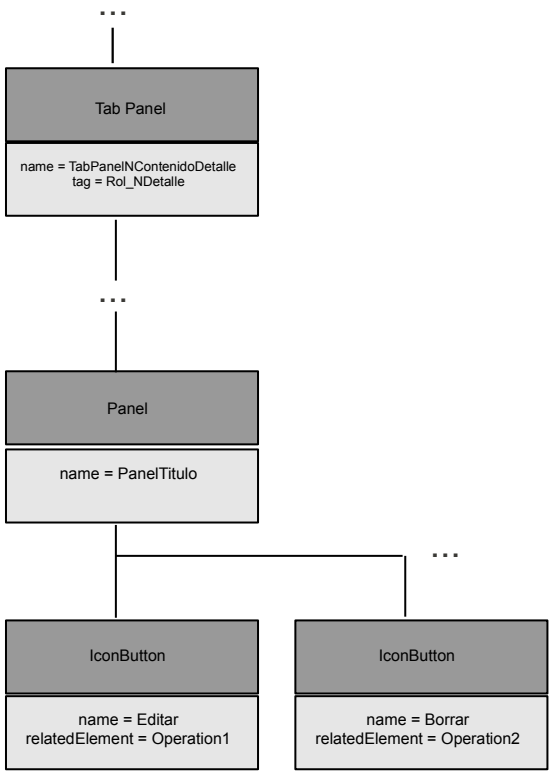
- Elemento *InformationIU* con *name* = “Ventana Detalle 2 Consulta”

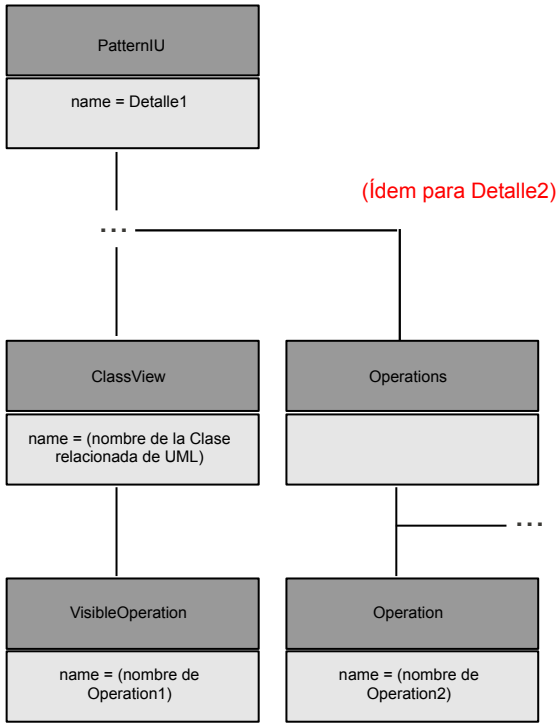
- Elemento *VisualizationSet* con *name* = “Ventana Detalle 2 Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- *Elemento ClassView con name* = “ClassView1” → *contiene 6 visibleFeature → 5 VisibleAttribute* y 1 *VisibleOperation*

El atributo *operation* de los *VisibleOperation* apunta al *Operation* relacionado

Identificador	Descripción
TFR-007-V04-002	Elementos *Operation* procedentes de un modelo UML relacionados en los elementos *IconButton* del Panel Título de los N Detalles **Maestro 1CEI (Registro) – N Detalle 1C (Tabular) 1EI (Registro)** **N = 2**
Severidad	**Tipo**
Bloqueante	Funcional
Datos de entrada	**Datos de salida esperados**





Pruebas realizadas

- Elemento *PatternIU* con *name* = “MnD”

- Elemento *Role* con *name* = “Maestro”

- Elemento *PatternIU* con *name* = “MnD”

- Elemento *EditableInformationIU* con *name* = “Ventana Maestro Consulta-Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Maestro Consulta-Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- *Elemento ClassView con name* = “ClassView1” → *contiene 5 visibleFeature → 4 VisibleAttribute* y 1 *VisibleOperation*

- Elemento *Filter con name* = “Búsqueda” → contiene 4 *filterParameter*

- Elemento *Operations* → contiene 2 *Operation*

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle 1”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle 1 Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 1 Edición-Inserción” → contiene un *visualizationPattern*

y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

- Elemento *Operations* → contiene 2 *Operation* (nombre de las *Operation* relacionadas)

- Elemento *InformationIU* con *name* = “Ventana Detalle 1 Consulta”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 1 Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- *Elemento ClassView con name* = “ClassView1” → *contiene 5 visibleFeature → 5 VisibleAttribute* y

- Elemento *ClassView* con *name* = (nombre clase UML) → contiene 1 *visibleFeature* → 1 *VisibleOperation* (nombre de la *Operation* relacionada)

- Elemento *Role* con *name* = “Detalle”

- Elemento *PatternIU* con *name* = “Detalle 2”

- Elemento *EditableInformationIU* con *name* = “Ventana Detalle 2 Edición-Inserción”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 2 Edición-Inserción” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Registro

- Elemento *ClassView* con *name* = “ClassView1” → contiene 4 *visibleFeature* → 4 *VisibleAttribute*

- Elemento *Operations* → contiene 2 *Operation* (nombre de las *Operation* relacionadas)

- Elemento *InformationIU* con *name* = “Ventana Detalle 2 Consulta”

- Elemento *VisualizationSet* con *name* = “Ventana Detalle 2 Consulta” → contiene un *visualizationPattern* y un *defaultVisualizationPattern* de tipo Tabular

- *Elemento ClassView con name* = “ClassView1” → *contiene 5 visibleFeature → 5 VisibleAttribute* y

- Elemento *ClassView* con *name* = (nombre clase UML) → contiene 1 *visibleFeature* → 1 *VisibleOperation* (nombre de la *Operation* relacionada)

El atributo *operation* de los *VisibleOperation* apunta al *Operation* relacionado

5.2. Proceso de automatización de las pruebas

Conociendo el proceso de ejecución de las pruebas automáticas, en el siguiente apartado procederemos a explicar su implementación.

Existen una serie de proyectos (*plugins*) desarrollados que entran en juego en todo el proceso: en el lanzamiento de la transformación, en la validación de los modelos y en la ejecución de los test *JUnit*.

A continuación detallaremos uno a uno todos estos proyectos, indicando sus clases más importantes y profundizando en su funcionamiento:

5.2.1. Plugin es.cv.gvcase.mdt.sketcher.sketcher2uim.test

Este proyecto está formado por 3 carpetas, un script *Ant* y una clase *JUnit*. Se trata de un proyecto de tipo *plug-in project*, ya que mediante este tipo de proyecto podremos tener cargados los modelos en memoria y ejecutar la aplicación *JUnit Plugin Test*.

La carpeta **models** contiene una serie de ficheros con extensión ".hutn" y otros ".model". Los primeros son las plantillas en lenguaje *HUTN* que especifican los modelos de entrada que se utilizarán para las pruebas. Cada uno tiene por nombre el identificador de la prueba que representa. Los segundos se generan a partir de las plantillas bien al construir el *workspace*, al salvar el contenido de los ".hutn" o al generarlos manualmente con el menú contextual. Se trata de los modelos de entrada.

La carpeta **configurations** incluye, en caso de que sea necesario, una serie de plantillas *HUTN* que definen los patrones de configuración aplicables a las transformaciones de *MOSKitt* (en este caso *Sketcher2UIM*) para algunos modelos de entrada. Se pueden considerar parte de los modelos de entrada.

En este proyecto en concreto, dicho directorio se encuentra vacío ya que actualmente la transformación *Sketcher2UIM* no dispone de configuración posible.

Por último, también está presente la carpeta **validations**, la cual incorpora todos los ".evl" que corresponden a los ficheros de validación de los modelos generados tras la transformación.

El script *Ant* corresponde con el fichero ***moveModels.xml***. Se trata de una plantilla *XML* que contiene una cabecera y tres grupos de código etiquetados como <*target*>, cada uno con su función especifica.

En la cabecera se establecen una serie de propiedades genéricas que corresponden con los directorios donde se copiarán los modelos de configuración, así como dónde se generarán los modelos resultantes de la transformación.

Las siguientes 3 secciones de código corresponden a:

1. **create-folders**: Realiza una serie de acciones *mkdir*, que no es más que la creación de los directorios especificados en la cabecera, donde se copiarán los modelos.

2. **rename-confmodels**: Aquí se renombran todos los modelos de configuración, con extensión "*.model"*, a otros con el mismo identificador pero con extensión "*.transformationconfiguration*". Posteriormente son copiados a un directorio presente en el $HOME del usuario.

3. **replace-confmodels**: Para cada fichero de configuración, ya con su extensión modificada a "*.transformationconfiguration*" reemplaza la cadena "*resource*" por "*plugin*".

Finalmente, nos encontramos con la única clase *Java* del *plugin* de test, que corresponde con una clase *JUnit* y tiene por nombre ***TransformationAndValidation.java***.

Cada método de prueba se etiqueta con @Test. También hay un par de métodos que se etiquetan con @BeforeClass y @AfterClass. El primero de ellos es un método que se ejecuta previo al lanzamiento de todos los test, mientras que el otro justo al contrario, cuando acaban todas las pruebas es cuando se lanza.

```
@BeforeClass
public static void setUpBeforeClass() throws Exception {
	nTests = nHits = nFailures = 0;
	start = System.currentTimeMillis();

	metamodelFile = "uim_1.3.0.ecore";
	metamodelUri = "http:///es.cv.gvcase.mdt.uim/1.3.0";
	dirValidationInputModels = "/outputmodels/";
	transformationInputPath =
	"platform:/plugin/es.cv.gvcase.mdt.sketcher.sketcher2uim.test/models/";
	namePlugin = "es.cv.gvcase.mdt.sketcher.sketcher2uim.test";

	String home = System.getProperty("user.home");
	String wkspace = home + "/tempWorkspace/" + namePlugin;

	transformationOutputPath = "file:/" + wkspace + "/outputmodels/";
	dirConfigurationModels = "file:/" + wkspace + "/confmodels/";
	dirValidations =
	"platform:/plugin/es.cv.gvcase.mdt.sketcher.sketcher2uim.test/validations/";
	transformationId = "es.cv.gvcase.mdt.sketcher.sketcher2uim.Sketcher2UIM";

	EPackage.Registry.INSTANCE.put(metamodelFile, UimPackage.eINSTANCE);

	tav = new TransformAndValidate(transformationId,
		dirValidationInputModels, dirValidations, metamodelFile,
		metamodelUri, transformationInputPath,
		transformationOutputPath, dirConfigurationModels, "uim", namePlugin);
}
```

Método setUpBeforeClass()

Sobre estas líneas vemos el contenido del método *setUpBeforeClass()*, que se encarga de asignar valores a ciertas propiedades así como de instanciar la clase con la que se ejecutarán las validaciones.

Las variables *dirValidations*, *dirValidationInputModels*, *transformationInputPath*, *transformationOutputPath* y *dirConfigurationModels* contienen las direcciones de los modelos. Como podemos observar, los modelos de entrada (carpeta *models*) estarán situados en el *plugin* que tendremos cargado en la plataforma (*platform:/plugin*). Sin embargo, debido a que en algunos casos los modelos de configuración son sobreescritos para modificar cierta información, no pueden ser dependientes del *plugin*, luego debemos copiarlos a un directorio distinto al del *plugin* (podemos observar que se guardan en una carpeta temporal en el $HOME).

Para los ficheros de validación *EVL* se sigue el mismo esquema que en los modelos de entrada; esto es, se encontrarán también en el propio *plugin*.

En cuanto a los modelos que serán generados como resultado de la transformación, los almacenaremos en una ruta absoluta (*file:/*), que compartirán sitio con los modelos de configuración comentados anteriormente. Más tarde, cuando los métodos de validación busquen estos modelos resultantes de la transformación, se les indicará que se encuentran también en la misma ruta que se ha definido aquí.

La instrucción

```
EPackage.Registry.INSTANCE.put(metamodelFile, UimPackage.eINSTANCE);
```

registra el metamodelo de *UIM* en el entorno, ya que es indispensable para la ejecución de las validaciones por código.

La variable **tav** es de tipo *TransformAndValidate*, que está definida en el *plugin es.cv.gvcase.linkers.test* y que más tarde analizaremos. Como parámetros del constructor se pasan el nombre del metamodelo y su URI, las rutas de los modelos de entrada y salida de la transformación, la de los modelos de entrada de la validación y la de las propias validaciones, y el identificador de la transformación, así como el nombre del propio *plugin* de test.

```
@Test
public void TFR_MAP_001_V00_001() {
	boolean error;
	error = tav.execute("TFR-MAP-001-V00-001", ol);
	if(error == false)
		nHits++;
	else
		nFailures++;
	nTests++;
	assertFalse(error);
}// end TFR_MAP_001_V00_001
```

Método de test

El fragmento de código de arriba representa un método a probar. Como nombre se elige el identificador de la prueba, sustituyendo los guiones por subguiones.

Se declara la variable booleana *error*, que contendrá el valor *true* si se han detectado fallos en el método *tav.execute("[Identificador de la prueba]")* o *false*, en caso de que

la prueba se supere con éxito. Puesto que el valor esperado para la variable *error* es *false*, para que *JUnit* la reconozca como prueba satisfactoria se ejecuta el método *assertFalse(error)*. Se declara un método de prueba por cada caso que deseamos comprobar.

El resto de código es sencillo de comprender, simplemente se actualiza el número de tests ejecutados, así como el número de aciertos (o fallos) para luego mostrar un pequeño sumario.

5.2.2. Plugin es.cv.gvcase.linkers.test

Este proyecto contiene los métodos utilizados para ejecutar la transformación y la validación por código. Sus elementos son independientes del tipo de transformación, es decir, el código puede ser reutilizado por cualquier otro proyecto de *testing* automático, ya sea para este tipo de transformación o para otro diferente.

En él distinguimos dos paquetes:

5.2.2.1. Paquete es.cv.gvcase.linkers.test.evl

Contiene tres clases que implementan los métodos necesarios para invocar la validación de modelos utilizando *EVL*. Como ya sabemos, *EVL* forma parte del *Epsilon Project* y nos proporciona cierta funcionalidad, por lo que las clases modifican, extienden y/o dependen de otras existentes en dicho proyecto.

1. **Clase *EpsilonStandaloneFile***

Es una modificación de la clase *EpsilonStandalone* de *Epsilon*, sobre la que se basan las diferentes tecnologías que componen el mismo. Los métodos modificados y que nos conciernen son *execute()*, *createEmfModel()* y *createEmfModelByURI()*.

```
public boolean execute(String modelFile, String metamodelFile, String metamodelUri,
String dirModels, String dirValidations, String validationFile, PrintWriter output)
throws Exception {


	module = createModule();
	String validationPath =
	FileLocator.toFileURL(Platform.getBundle(namePlugin).getResource("validations/"
	+ validationFile)).getPath();
	//We can pass the name of the file instead calling the method getSource()
	module.parse(new File(validationPath));
	if (module.getParseProblems().size() > 0) {
```

```
                System.err.println("Parse errors occured...");
                for (ParseProblem problem : module.getParseProblems()) {
                        System.err.println(problem.toString());
                }
                System.exit(-1);
        }
        for(IModel model:getModels(modelFile, metamodelFile, metamodelUri, dirModels)) {
                module.getContext().getModelRepository().addModel(model);
        }
        module.execute();
        //hasErrors contains if errors exist when validating
        boolean hasErrors = postProcess(validationFile, output);
        module.getContext().getModelRepository().dispose();
        //returns a value in order to have a feedback in JUnit
        return hasErrors;
}
```

Método execute()

Al método *execute()* se le pasan el modelo, el metamodelo, la *URI*, la dirección de los modelos, la dirección de los archivos de validación y el archivo de validación. Se crea un módulo y se le pasa el archivo de validación, y si no hay problemas se añade el modelo a validar. Finalmente se ejecuta el módulo, realizando la validación.

Existen dos formas de acceder al metamodelo sobre el cual están basados los modelos. Una se basa en el nombre del fichero *ecore* y la otra trabaja con su *URI*. En nuestro propósito seguiremos la segunda opción.

Los métodos ***createEmfModel()*** y ***createEmfModelByURI()*** crean un modelo *EMF*, aunque el primero accede al metamodelo por el nombre del fichero y el otro por la *URI*. En la variable *propertyModelFile* metemos el *path* del modelo original, el que se quiere comprobar, creamos un nuevo elemento *emfModel* y asignamos sus propiedades.

2. **Clase *EvlStandAloneFile***

Extiende a *EpsilonStandaloneFile*, e implementa el método *getModels()* y el *postProcess()*, que nos indica si se han superado las restricciones o no, mostrando por consola los mensajes de error y el número que no se ha completado con éxito. El método *getModels()* que implementa esta clase invoca a *createEmfModel()*.

3. **Clase *EvlStandaloneMetamodelByUriFile***

Es la tercera y última clase del paquete. Extiende a *EvlStandaloneFile* e implementa el método *getModels()*.

5.2.2.2. Paquete es.cv.gvcase.linkers.test.junit

En este paquete existen un par de clases que se encargan de lanzar la transformación para su validación y generar un informe con los resultados obtenidos.

1. **Clase *TransformAndValidate***

Realiza exactamente lo que su nombre indica, la transformación de un modelo y su validación.

En el constructor de la clase se establecen los valores de las rutas en las que se encuentran los modelos y archivos de validación, la *URI* del metamodelo, el nombre del *plugin* de test y el identificador de la transformación.

La variable *evl* es del tipo *EvlStandaloneMetamodelByUriFile*, clase que se declara en el paquete de la sección anterior.

La clase define un solo método, *execute(String [id prueba])*, que analizaremos detenidamente:

```
public boolean execute(String fileName, PrintWriter ol){
	TransformationDesc tr =
	TransformationsRegistry.getTransformation(transformationID);
	ArrayList<String> messagesList = new ArrayList<String>();
	HashMap<String, TransformedResource> inputParams = new HashMap<String,
	TransformedResource>();
	HashMap<String, TransformedResource> outputParams = new HashMap<String,
	TransformedResource>();
	inputParams.put("projectInputParam", new
	TransformedResource("projectInputParam", transformationInputPath + fileName +
	".model"));
	outputParams.put("fileOutputParam", new TransformedResource("fileOutputParam",
	transformationOutputPath + fileName + "." + extension));
	boolean errors = true;
	System.out.println("\nTransforming: " + fileName);
	boolean execOK = TransformationsRegistry.executeTransformation(tr.getId(),
	inputParams, outputParams, messagesList);
	try {
		if (execOK == true) {
			System.out.println("Validating: " + fileName);
				errors = evl.execute(fileName + "." + extension,
	metamodelFile, metamodelUri, dirModels, dirValidations, fileName + ".evl", ol);
		}
		else {
			OutputLog.addToLog("\nIn validation " + fileName, ol);
```

```
                    OutputLog.addToLog("Can't transform " + fileName + "
                    succesfully.", ol);
                    System.out.println("Can't transform " + fileName + "
                    succesfully.");
                    OutputLog.addToLog("Reasons:", ol);
                    System.out.println("Reasons:\n");
                    Iterator<String> iter = messagesList.iterator();
                    while(iter.hasNext()){
                           String message = iter.next();
                           System.out.println(message);
                           OutputLog.addToLog(message, ol);
                    }
             }
      } catch (Exception e) {
             e.printStackTrace();
             nErrors++;
             OutputLog.addToLog("\nIn validation " + fileName, ol);
             OutputLog.addToLog(e.getMessage(), ol);
      }
      return errors;
}
```

Método execute()

A la variable local *tr*, que es de tipo *TransformationDesc* (descripción de una transformación), se le asigna el identificador de la transformación a aplicar.

Se crean dos *HashMap*, uno hará referencia al modelo de entrada (*inputParams*) y otro al recurso generado (*outputParams*) y se asignan sus valores a partir de los valores de variables de la clase, que lo toman de los parámetros con que se crea dicha clase.

Posteriormente se invoca la transformación, pasando como parámetros el identificador de la misma, que indica cual se debe aplicar, el recurso a transformar y el que debe ser el transformado, y un elemento *ArrayList* vacío.

Si el paso anterior se ha completado sin errores se inicia la validación, que toma como parámetros el modelo a comprobar, el metamodelo y su URI, el *path* del modelo, el directorio donde se encuentran los archivos de validación y el nombre del que utilizamos.

El método devuelve *true* si se han detectado errores y *false* si se ha completado con éxito. Además, tanto como si han habido errores en la transformación como en alguna

de las validaciones, estos se van almacenando en un *log* que será el que se mostrará en el sumario final cuando finalicen todos los test.

2. Clase ***OutputLog***

Se encarga de instanciar a la clase *PrintWriter*, que será el *log* que contiene la lista de errores (si han habido) ocurridos durante el lanzamiento de las pruebas.

5.3. Proceso de construcción

Ya conocemos los pasos a seguir en un proceso de construcción automático, luego en este apartado procederemos a explicar el modo en el que ha sido realizado para este proyecto.

Para la construcción de cualquier módulo, componente e incluso el propio producto *MOSKitt* se ha utilizado un *job* de *Hudson*.

Nosotros partimos de la base de que la plataforma *MOSKitt* ya está construida y disponible, por lo que nos hemos centrado en la construcción de aquellos módulos que complementen a la herramienta para poder lanzar los casos de prueba definidos.

Estos módulos son aquellos que contienen las transformaciones modelo a modelo. Concretamente, en nuestro proyecto nos hemos centrado en el módulo *sketcher*, que contiene la transformación *Sketcher2UIM*.

Aun así, no sólo nos basta con construir el módulo de la transformación, ya que nos hará falta el *plugin* que contiene los modelos y la clase *JUnit* que ejecuta las pruebas. En un principio los construimos por separado por aspectos ajenos a este proyecto, pero lo lógico es que el *plugin* de test se incluya junto con el módulo de la transformación y se construya todo junto.

Así pues, el *script* empleado en la construcción de cualquiera de estos componentes sigue el siguiente esquema general:

```
cd $HOME/moskitt-galileo-dev/p2_scripts
./updatingBuilder.sh branches/moskittbase/
cd $HOME/moskitt-galileo-dev/build/es.cv.gvcase.releng.builder
rm -rf $WORKSPACE/*
./changeLines nocomment
./build.sh -Dcomponent=$MODULO \
  -DbuildDirectory=$WORKSPACE \
  -DtransformedRepoLocation=$WORKSPACE/transformedRepo \
  -DcomponentsLocation=$REPOS/components \
  -DrepoBaseLocation=$REPOS/base-context/trunk \
  -Dp2.build.repo=file:$WORKSPACE/result/site.p2 -DskipMirroring=true \
  -DtagPath=$TAGPATH \
  -DforceContextQualifier=$QUALIFIER
```

Script de construcción de un módulo en Hudson

En el código de arriba podemos diferenciar la ejecución de varios *scripts*:

updatingBuilder.sh

Este *script* realiza una actualización del contenido del directorio *es.cv.gvcase.releng.builder*, a través de la orden de comando *svn export*.

En esta carpeta existe un subdirectorio por cada componente o módulo que se quiera construir. En cada uno de estos directorios existen una serie de ficheros que configuran la construcción del componente.

changeLines.sh

Se trata de un *script* que modifica el contenido del *build.xml* utilizado para la construcción. En este fichero hay un par de líneas que deben ser sustraídas o no ser eliminadas en función del componente que se vaya a construir.

```
<property name="repoBaseLocation" value="${buildDirectory}/inputRepositories" />
<property name="transformedRepoLocation" value="${buildDirectory}/transformedRepo" />
```

Estas líneas indican dos propiedades sobre ciertos directorios a tener en cuenta en el proceso de construcción:

- ***repoBaseLocation*** indica el directorio donde se sitúa el contexto, es decir, todos aquellos componentes que son dependencia directa del componente a construir

- ***transformedRepoLocation*** es un directorio donde se genera un repositorio que contiene todos los componentes presentes en el contexto, para que se pueda compilar contra éste.

Existen ciertos componentes que tienen dependencias con otros, pero hay algunos que no dependen de nadie. Aquellos que tengan dependencias deben mantener las 2 líneas, pero los que no deben quitarlas, ya que estos 2 parámetros no se especificarían en la ejecución de su construcción.

build.sh

Se trata de un *script* que invoca a la aplicación *antRunner* de *Eclipse* para ejecutar un *script Ant* que pone en marcha la construcción de un componente.

El fichero de *Ant* que se lanza es un *build.xml* que se encarga de lanzar el *build.xml* comentado anteriormente (esqueleto principal). En él se especifican una serie de propiedades referentes a la configuración de la construcción del componente. Entre ellas, podemos destacar las que pasamos como parámetro en el *job*:

- **component:** no es más que el nombre del componente o módulo.

- **buildDirectory**: directorio donde se descargarán todos los fuentes y donde se interactuará con el componente para su construcción. Es el directorio de trabajo del *job*.

- **transformedRepoLocation:** directorio donde se generará el repositorio que contiene todos los componentes presentes en el contexto.

- **componentsLocation:** se trata de un directorio donde se hará una copia del repositorio resultante de la construcción del módulo.

- **repoBaseLocation:** directorio donde se encuentra el contexto necesario para la compilación del componente a construir.

- **p2.build.repo:** directorio donde se generará el repositorio *p2* resultante de la construcción.

- **tagPath:** localización de los fuentes en el repositorio *subversion*.

- **forceContextQualifier:** identificador de la construcción en el formato *yyyymmddhhnn* (año, mes, día, hora y minuto). Todas las *features* y *plugins* envueltos en el componente llevarán este sufijo detrás de su número de versión.

En la sección de análisis vimos el proceso que se sigue para la construcción de un componente. Ahora vamos a conocer el funcionamiento del proceso internamente.

```
<project default="main">
	...
		<echo message=""/>
		<echo message="gvCASE Build Launcher"/>
		<echo message="======================================="/>
		<echo message="Build script: ${buildScript}"/>
		<echo message="Script location: ${buildScriptLocation}"/>
		<echo message="Component: ${component}"/>
		<echo message="Build dir: ${buildDirectory}"/>
		<echo message="Eclipse Base dir: ${baseLocation}"/>
		<echo message="buildId: ${buildId}"/>
		<echo message="======================================="/>

		<!--run the build for the specified component-->
		<ant antfile="${buildScript}" dir="${buildScriptLocation}">
			<property name="builder" value="${basedir}/${component}" />
			<property name="builderBaseDir" value="${basedir}" />
		</ant>
	</target>
...
</project>
```

Contenido del build.xml inicial

Podemos observar como la función principal de este fichero es la llamada al *build.xml* genérico para la construcción. Antes de realizar esta llamada, se establecen todas las propiedades que contienen la información acerca del componente.

En el *build.xml* general se realizan todos los pasos comentados en la sección de análisis, desde la obtención de los *maps* y los fuentes hasta la compilación y ensamblaje del componente.

El resultado de la construcción es un repositorio *p2*, que podremos instalar en la plataforma a través de una referencia del mismo como *update site*.

5.4. Integración de las pruebas en la construcción

Aunque sabemos que existe un apartado referente a los *test* en el proceso de construcción, en el cual se ejecutarían los test oportunos y en caso de que exista algún tipo de error se detendría el proceso; en nuestra implementación hemos optado por separar la ejecución de los casos de prueba o *test* de la construcción del componente.

Esto es debido, en parte, a que para poder ejecutar los *test*, el propio *plugin* que los contiene debe ser instalado en *MOSKitt*, luego nos tenemos que esperar a que esté generado.

Aun así, no es problema la separación de los dos procesos, ya que se encadenan el lanzamiento de ambos; nada más acabe la construcción con resultado satisfactorio se ejecutará el lanzamiento de los test.

Como habíamos comentado en la fase de análisis, nos basamos en el lanzamiento de una versión *headless* de *Eclipse* a través de la cual ejecutamos una aplicación *JUnit Plugin Test* que nos ejecutará los diferentes test escritos mediante *JUnit*.

A continuación observamos el *script* que realiza todo el proceso a través de *Hudson*:

```
cd $HOME/moskitt-galileo-dev/p2_scripts


./downloadAndInstall.sh
-Dsource=http://www.moskitt.org/fileadmin/conselleria/documentacion/Descargas/1.3.5/mosk
itt-1.3.5.v201105190953-linux.gtk.x86.zip -Ddest=$HOME/tests/ -f downloadAndInstall.xml


cp -r $HOME/tests/junit-sketcher2uim/ $HOME/tests/moskitt/


./installRepo.sh file:$HOME/public_html/repos-1.3.x/epsilon/ $HOME/tests/moskitt/
org.eclipse.epsilon.feature,org.eclipse.epsilon.eugenia.feature,org.eclipse.epsilon.evl.
emf.validation.feature,org.eclipse.epsilon.gmf.feature,org.eclipse.epsilon.hutn.feature


./installRepo.sh file:$HOME/public_html/repos-1.3.x/linkers.test/ $HOME/tests/moskitt/
es.cv.gvcase.features.linkers.test


./installRepo.sh file:$HOME/public_html/repos-1.3.x/mdt.storage/ $HOME/tests/moskitt/
es.cv.gvcase.features.mdt.common.storage


./installRepo.sh file:$HOME/public_html/repos-1.3.x/uitemplates.cit/
$HOME/tests/moskitt/ es.cv.gvcase.module.uitemplates.cit
```

```
./installRepo.sh file:$HOME/public_html/repos-1.3.x/sketcher-1.1.0/ $HOME/tests/moskitt/
es.cv.gvcase.module.sketcher


./installRepo.sh file:$HOME/public_html/repos-1.3.x/sketcher2uim.test/
$HOME/tests/moskitt/ es.cv.gvcase.features.mdt.sketcher.sketcher2uim.test


./copyFiles.sh -f $HOME/tests/moskitt/sketcher2uim.test/moveModels.xml


./runTest.sh $HOME/tests/moskitt/ 80 es.cv.gvcase.mdt.sketcher.sketcher2uim.test
TransformationAndValidation $HOME/tests/moskitt/junit-sketcher2uim
```

Script para la ejecución de los test de sketcher2uim

En él diferenciamos una serie de llamadas a ciertos *scripts*:

downloadAndInstall.sh

Es un *script* que invoca a un fichero *Ant* llamado *downloadAndInstall.xml*. La función que desempeña es bastante trivial, el nombre lo dice todo. Se encarga de obtener la herramienta *MOSKitt* (ya sea de un directorio local o de una *url*) y de descomprimirla en el directorio especificado en *dest*.

En este caso está obteniendo la herramienta de la página oficial de *MOSKitt*, a través de la *url* de descarga, aunque si ya se tuviese descargada en local podría ser referenciada la ruta y el proceso sería más rápido.

installRepo.sh

El nombre también da una pequeña referencia a su funcionamiento. Este *script* instala el repositorio (módulo) pasado como parámetro en la herramienta *MOSKitt*.

Bajo estas líneas se encuentra el comando principal del *script* que realiza la instalación del componente elegido sobre cualquier herramienta basada en *Eclipse*. Es una aplicación conocida como *p2.director*.

Para poder ejecutar la aplicación, también utilizamos un *Eclipse* “base” que no levanta ningún tipo de interfaz gráfica.

```
...
${HOME}/moskitt-galileo-dev/build/org.eclipse.releng.basebuilder/eclipse -application
org.eclipse.equinox.p2.director \
-repository ${repo} \
-installIU ${listOfFeatures} \
```

```
-destination ${platformFolder} \
-bundlepool ${platformFolder} \
-vmargs -Dorg.eclipse.ecf.provider.filetransfer.excludeContributors=
org.eclipse.ecf.provider.filetransfer.httpclient \
-Dhttp.proxyPort=8080 \
-Dhttp.proxyHost=193.145.204.53
...
```

Script que instala un componente en *MOSKitt*

Se le tienen que pasar 3 parámetros de entrada:

1. *Url* del repositorio del componente a instalar
2. Directorio de instalación de la plataforma
3. Lista de *features* que se quieren instalar

Los argumentos adicionales especifican la utilización de un *proxy* para las conexiones que tenga que usar para actualizarse. Se trata de un requisito externo al proyecto.

copyFiles.sh

Es el *script* encargado de ejecutar el fichero *Ant moveModels.xml* que se encarga de copiar los modelos de configuración de la transformación a un directorio local.

runTest.sh

A continuación podemos observar la parte más importante del contenido de este *script*:

```
...
java -XX:MaxPermSize=256m -Xms40m -Xmx512m \
-classpath                      ${platformFolder}/plugins/org.eclipse.equinox.launcher_*
org.eclipse.equinox.launcher.Main \
-port ${port} \
-testLoaderClass                 org.eclipse.jdt.internal.junit4.runner.JUnit4TestLoader
-loaderpluginname org.eclipse.jdt.junit4.runtime \
-classNames ${namePlugin}.${testClass} \
-application org.eclipse.pde.junit.runtime.uitestapplication \
-data ${workspace} \
-testpluginname ${namePlugin} \
-consoleLog -debug
```

Script que ejecuta los test en *MOSKitt*

Realiza la función principal de todo el proyecto, ya que es el encargado de lanzar los casos de prueba diseñados a través de una aplicación de *Eclipse* en modo *headless*.

Acepta 5 parámetros de entrada:

1. Directorio de instalación de la plataforma
2. Puerto
3. Nombre del *plugin* de test
4. Nombre de la clase que contiene los test *JUnit*
5. Directorio del workspace

El **puerto** es un parámetro que estamos obligados a establecer en este tipo de aplicación, y cuyo propósito es quedarse escuchando en un puerto disponible a que los resultados de los test sean enviados, para posteriormente hacer un *report* mediante *Ant*. En nuestro caso no estamos dándole funcionalidad, simplemente indicamos un puerto conocido porque es un requisito exigido.

Mediante el **nombre del *plugin*** y de la **clase *Java*** proporcionadas, la aplicación *org.eclipse.pde.junit.runtime.uitestapplication* sabe de dónde extraer los diferentes test y así poder lanzarlos.

El **directorio referente al *workspace*** es vital por si necesitamos utilizar algún modelo auxiliar, ya que podemos crearnos una carpeta que contenga todos estos modelos y luego apuntar para que ésa sea el espacio de trabajo.

Además, en mitad del proceso podemos ver como se copia un directorio *junit-sketcher2uim* al lugar en el que se descomprime *MOSKitt*. Esta carpeta se trata de un proyecto creado con la herramienta que contiene una serie de modelos auxiliares necesarios para la definición de los modelos de entrada.

Esto es debido a que dichos modelos auxiliares no pueden ser incluidos dentro del *plugin* y son referenciados mediante *platform:/resource*, por lo que se tiene preparado un directorio que se copia al *workspace* para se tengan disponibles.

La aplicación *uitestapplication* tiene una peculiaridad, y es que realmente no deja ejecutarse si no existe en marcha una instancia de *Eclipse* levantada (con la interfaz gráfica incluida). Aunque podamos lanzar el proceso en modo *headless*, este tipo de aplicación necesitará levantar una instancia durante el periodo de ejecución de los test.

Esto no supone un problema aunque queramos automatizar el proceso desde una máquina externa, como es el caso. Existen herramientas que permiten 'virtualizar' una sesión gráfica durante la ejecución de una tarea en *Hudson*. Una de ellas es *xvnc*.

Simplemente con indicar que utilizamos *xvnc* en la ejecución de este *job*, el proceso entero levantará la plataforma en el *buffer* virtual y se podrá ejecutar sin problemas.

Llegados a este punto, hemos visto la implementación de todas las partes del proyecto, desde el diseño de los casos de prueba a ejecutar hasta el lanzamiento de los test en modo *headless*.

Así pues, tan sólo nos quedará contemplar una ejecución real para observar los resultados que se producen, y así poder confirmar el funcionamiento de toda la infraestructura montada.

En la siguiente sección expondremos un caso real de ejecución de *test* sobre la transformación *Sketcher2UIM*.

6. Resultados

En este apartado vamos a mostrar un ejemplo de ejecución del funcionamiento de nuestra aplicación, en la cual lanzaremos una serie de casos de prueba sobre la transformación *Sketcher2UIM*.

6.1. Preparación

El primero paso es más que evidente. Se diseñan los distintos casos de prueba que se pretenden ejecutar. En este ejemplo utilizaremos los que hemos visto en el apartado 5.1.

Ya tenemos pues, los diferentes modelos generados con *HUTN* que nos servirán como entrada a nuestra aplicación, así como los diferentes ficheros de validación realizados con *EVL* que contrastarán los resultados obtenidos con los esperados (indicados en los casos de prueba).

Todos estos modelos, junto con el *moveModels.xml* estarán disponibles en el *plugin es.cv.gvcase.mdt.sketcher.sketcher2uim.test*, que será el que tengamos que construir como módulo para instalarlo en *MOSKitt* y lanzar los test que contiene.

Además, partimos de que disponemos de todos los repositorios de los módulos que necesitamos para lanzar la transformación *Sketcher2UIM* (suponemos que ya están construidos). Éstos son:

1. **RCP MOSKitt** → contiene *UIM*.
2. **Sketcher** → contiene también la transformación *Sketcher2UIM*.
3. **Storage** → dependencia de s*ketcher*.
4. **UiTemplates.CIT** → dependencia de *sketcher*.
5. **Linkers.test** → dependencia del *plugin* de test.
6. **Epsilon** → dependencia de *linkers.test*.

Estos repositorios serán instalados en *MOSKitt* en el proceso de ejecución de los test, junto con el que contiene el *plugin* de test, que nosotros construiremos.

6.2. Ejecución

Podemos dividir la ejecución en 2 partes, una por cada tarea de *Hudson* que se ejecutará.

Pese a que la ejecución de las 2 tareas se realiza de forma encadenada, al elaborarlas por separado debemos dedicar un apartado a cada una de ellas.

6.2.1.1. Construcción del plugin de test

El *plugin es.cv.gvcase.mdt.sketcher.sketcher2uim.test* debe ser construido para poder ser instalado como módulo en *MOSKitt* y así poder ejecutar los test que contiene.

Para ello debemos crear una *feature* que lo contenga, ya que nuestro modo de construcción se realizar por *features*. Suponiendo que tenemos esta *feature* (*es.cv.gvcase.features.mdt.sketcher.sketcher2uim.test*) en nuestro repositorio *SVN* junto con el *plugin* de test, definimos el *job* de *Hudson* que obtendrá el repositorio del módulo de test.

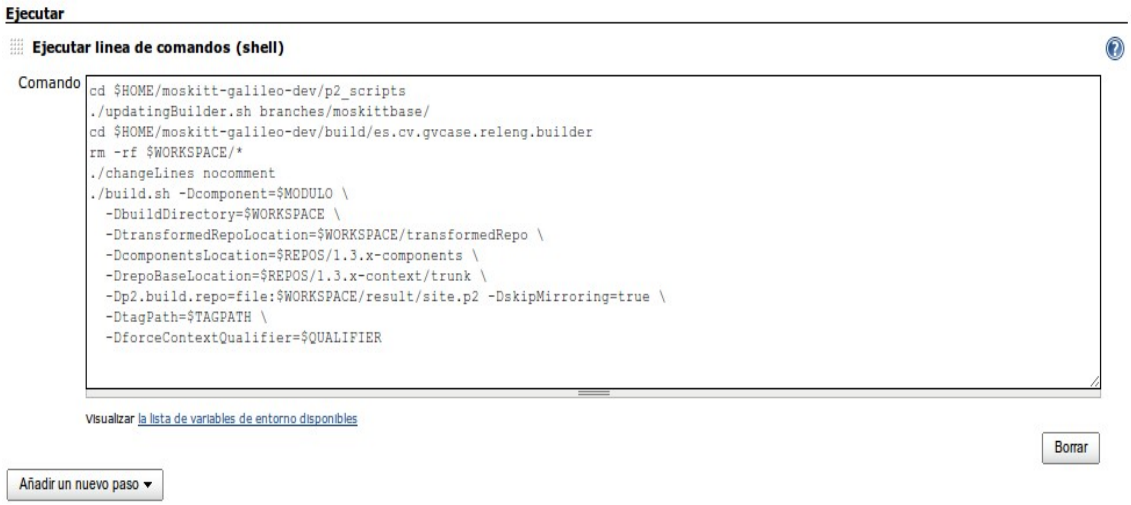


Figura 11. Definición del job de construcción del plugin de test

Ponemos en marcha la construcción y esperamos a que termine. Una vez obtenemos un resultado satisfactorio, se habrá generado un repositorio con el *plugin* de test y ya estaremos en condiciones de poder lanzar los test para probar la transformación.

Localizamos todos los repositorios necesarios en un directorio común para que el siguiente paso resulte más sencillo.

6.2.1.2. Lanzamiento de los test

Definimos una nueva tarea de *Hudson* que cogerá todos los repositorios, los instalará en *MOSKitt* y lanzará los test.

Figura 12. Definición del job de ejecución de los test

Esta tarea coge un binario de *MOSKitt* localizado en un directorio local, y va instalando los distintos repositorios (también situados en un directorio local), contando con el repositorio construido en el apartado anterior (que contendrá los test).

Una vez todo instalado, llama a la aplicación de *Eclipse* que permite ejecutar test *JUnit*.

Durante la ejecución del *job*, se abrirá una sesión *xvnc* que permitirá levantar temporalmente una instancia de *MOSKitt* para la ejecución de los test.

6.3. Informe de resultados

A continuación podemos observar la salida de consola generada por la tarea de *Hudson* ejecutada en el apartado anterior. Podemos apreciar al final del todo el pequeño sumario que recoge un resumen de todos los test ejecutados:

```
Starting xvnc
[workspace] $ vncserver :10

New 'mkserver:10 (hudson)' desktop is mkserver:10

Starting applications specified in /home/hudson/.vnc/xstartup
Log file is /home/hudson/.vnc/mkserver:10.log

[workspace] $ /bin/sh -xe /home/hudson/apache-tomcat/temp/hudson2371670403486397580.sh
+ rm -rf /home/hudson/tests/moskitt/
+ cd /home/hudson/moskitt-galileo-dev/p2_scripts
+./downloadAndInstall.sh -Dsource=file:///home/hudson/public_html/rcp/moskitt-
1.3.5.v201105190953-linux.gtk.x86.zip -Ddest=/home/hudson/tests/ -f downloadAndInstall.xml
Archivo de construcción: downloadAndInstall.xml

check:

install:

download:
    [echo] "Downloading MOSKitt from file:///home/hudson/public_html/rcp/moskitt-
1.3.5.v201105190953-linux.gtk.x86.zip..."
     [get] Getting: file:/home/hudson/public_html/rcp/moskitt-1.3.5.v201105190953-linux.gtk.x86.zip
     [get] To: /home/hudson/moskitt-galileo-dev/p2_scripts/MOSKitt.zip
     [get] ..................................................

unzip:
    [echo] "Unzipping MOSKitt in /home/hudson/tests/..."
   [unzip] Expanding: /home/hudson/moskitt-galileo-dev/p2_scripts/MOSKitt.zip into
/home/hudson/tests

clean:
  [delete] Deleting: /home/hudson/moskitt-galileo-dev/p2_scripts/MOSKitt.zip

+ chmod +x /home/hudson/tests/moskitt/MOSKitt
+ cp -r /home/hudson/tests/junit-sketcher2uim/ /home/hudson/tests/moskitt/
+ ./installRepo.sh file:/home/hudson/public_html/repos-1.3.x/epsilon/ /home/hudson/tests/moskitt/
org.eclipse.epsilon.feature,org.eclipse.epsilon.eugenia.feature,org.eclipse.epsilon.evl.emf.validation.feat
ure,org.eclipse.epsilon.gmf.feature,org.eclipse.epsilon.hutn.feature

URL Repo: file:/home/hudson/public_html/repos-1.3.x/epsilon/
Platform location: /home/hudson/tests/moskitt/
List of features to install:
org.eclipse.epsilon.feature,org.eclipse.epsilon.eugenia.feature,org.eclipse.epsilon.evl.emf.validation.feat
ure,org.eclipse.epsilon.gmf.feature,org.eclipse.epsilon.hutn.feature

Creating list of features

Installing org.eclipse.epsilon.feature.feature.group 0.8.9.201004151115.
Installing org.eclipse.epsilon.eugenia.feature.feature.group 0.8.9.201004151115.
Installing org.eclipse.epsilon.evl.emf.validation.feature.feature.group 0.8.9.201004151115.
Installing org.eclipse.epsilon.gmf.feature.feature.group 0.8.9.201004151115.
Installing org.eclipse.epsilon.hutn.feature.feature.group 0.7.9.201004151115.
Operation completed in 48878 ms.
```

Deleting temporary files and folders

+ ./installRepo.sh file:/home/hudson/public_html/repos-1.3.x/linkers.test/ /home/hudson/tests/moskitt/ es.cv.gvcase.features.linkers.test

URL Repo: file:/home/hudson/public_html/repos-1.3.x/linkers.test/
Platform location: /home/hudson/tests/moskitt/
List of features to install: es.cv.gvcase.features.linkers.test

Creating list of features

Installing es.cv.gvcase.features.linkers.test.feature.group 1.0.0.201104060947.
Operation completed in 30124 ms.

Deleting temporary files and folders

+ ./installRepo.sh file:/home/hudson/public_html/repos-1.3.x/mdt.storage/ /home/hudson/tests/moskitt/ es.cv.gvcase.features.mdt.common.storage

URL Repo: file:/home/hudson/public_html/repos-1.3.x/mdt.storage/
Platform location: /home/hudson/tests/moskitt/
List of features to install: es.cv.gvcase.features.mdt.common.storage

Creating list of features

Installing es.cv.gvcase.features.mdt.common.storage.feature.group 0.9.3.201104261240.
Operation completed in 30484 ms.

Deleting temporary files and folders

+ ./installRepo.sh file:/home/hudson/public_html/repos-1.3.x/uitemplates.cit/ /home/hudson/tests/moskitt/ es.cv.gvcase.module.uitemplates.cit

URL Repo: file:/home/hudson/public_html/repos-1.3.x/uitemplates.cit/
Platform location: /home/hudson/tests/moskitt/
List of features to install: es.cv.gvcase.module.uitemplates.cit

Creating list of features

Installing es.cv.gvcase.module.uitemplates.cit.feature.group 1.0.0.201104061030.
Operation completed in 30524 ms.

Deleting temporary files and folders

+ ./installRepo.sh file:/home/hudson/public_html/repos-1.3.x/sketcher-1.5.0/ /home/hudson/tests/moskitt/ es.cv.gvcase.module.sketcher

URL Repo: file:/home/hudson/public_html/repos-1.3.x/sketcher-1.5.0/
Platform location: /home/hudson/tests/moskitt/
List of features to install: es.cv.gvcase.module.sketcher

Creating list of features
Installing es.cv.gvcase.module.sketcher.feature.group 1.5.0.201105191148.
Operation completed in 36303 ms.

Deleting temporary files and folders

+ ./installRepo.sh file:/home/hudson/public_html/repos-1.3.x/sketcher2uim.test/
/home/hudson/tests/moskitt/ es.cv.gvcase.features.mdt.sketcher.sketcher2uim.test

URL Repo: file:/home/hudson/public_html/repos-1.3.x/sketcher2uim.test/
Platform location: /home/hudson/tests/moskitt/
List of features to install: es.cv.gvcase.features.mdt.sketcher.sketcher2uim.test

Creating list of features

Installing es.cv.gvcase.features.mdt.sketcher.sketcher2uim.test.feature.group 0.9.0.201105241447.
Operation completed in 31532 ms.

Deleting temporary files and folders

+ ./copyFiles.sh -f /home/hudson/tests/moskitt/sketcher2uim.test/moveModels.xml
Unzipping es.cv.gvcase.mdt.sketcher.sketcher2uim.test_.jar
Archive:
/home/hudson/tests/moskitt/plugins/es.cv.gvcase.mdt.sketcher.sketcher2uim.test_0.9.0.201105241447
.jar

Archivo de construcción: /home/hudson/tests/moskitt/sketcher2uim.test/moveModels.xml

create-folders:
rename-confmodels:
replace-confmodels:

+ ./runTest.sh /home/hudson/tests/moskitt/ 80 es.cv.gvcase.mdt.sketcher.sketcher2uim.test
TransformationAndValidation /home/hudson/tests/moskitt/junit-sketcher2uim

Platform location: /home/hudson/tests/moskitt/
Port: 80
Plugin name: es.cv.gvcase.mdt.sketcher.sketcher2uim.test
Test Class: TransformationAndValidation
Workspace: /home/hudson/tests/moskitt/junit-sketcher2uim
Install location:
  file:/home/hudson/tests/moskitt/
Time to load bundles: 101

!SESSION 2011-05-24 15:06:23.453 ---

eclipse.buildId=unknown
java.version=1.6.0_23
BootLoader constants: OS=linux, ARCH=x86, WS=gtk, NL=es_ES
Framework arguments: -port 80 -testLoaderClass org.eclipse.jdt.internal.junit4.runner.JUnit4TestLoader
-loaderpluginname org.eclipse.jdt.junit4.runtime -classNames
es.cv.gvcase.mdt.sketcher.sketcher2uim.test.TransformationAndValidation -application
org.eclipse.pde.junit.runtime.uitestapplication -testpluginname
es.cv.gvcase.mdt.sketcher.sketcher2uim.test
Command-line arguments: -port 80 -testLoaderClass
org.eclipse.jdt.internal.junit4.runner.JUnit4TestLoader -loaderpluginname org.eclipse.jdt.junit4.runtime
-classNames es.cv.gvcase.mdt.sketcher.sketcher2uim.test.TransformationAndValidation -application
org.eclipse.pde.junit.runtime.uitestapplication -data /home/hudson/tests/moskitt/junit-sketcher2uim
-testpluginname es.cv.gvcase.mdt.sketcher.sketcher2uim.test -consoleLog -debug

```
!ENTRY org.eclipse.equinox.registry 2 0 2011-05-24 15:06:33.691
Starting application: 12048
Application Started: 22102

Transforming: TFR-MAP-001-V00-001
Validating: TFR-MAP-001-V00-001
All constraints have been satisfied

Transforming: TFR-MAP-001-V01-001
Validating: TFR-MAP-001-V01-001
All constraints have been satisfied

Transforming: TFR-MAP-001-V02-001
Validating: TFR-MAP-001-V02-001
All constraints have been satisfied

Transforming: TFR-MAP-001-V03-001
Validating: TFR-MAP-001-V03-001
All constraints have been satisfied

Transforming: TFR-MAP-001-V04-001
Validating: TFR-MAP-001-V04-001
All constraints have been satisfied

Transforming: TFR-MAP-002-V00-001
Validating: TFR-MAP-002-V00-001
All constraints have been satisfied
Transforming: TFR-MAP-002-V01-001
Validating: TFR-MAP-002-V01-001
All constraints have been satisfied

Transforming: TFR-MAP-002-V02-001
Validating: TFR-MAP-002-V02-001
All constraints have been satisfied

Transforming: TFR-MAP-002-V04-001
Validating: TFR-MAP-002-V04-001
All constraints have been satisfied
Transforming: TFR-MAP-003-V00-001
Validating: TFR-MAP-003-V00-001
All constraints have been satisfied

Transforming: TFR-MAP-003-V01-001
Validating: TFR-MAP-003-V01-001
All constraints have been satisfied

Transforming: TFR-MAP-003-V03-001
Validating: TFR-MAP-003-V03-001
All constraints have been satisfied

Transforming: TFR-MAP-003-V04-001
Validating: TFR-MAP-003-V04-001
All constraints have been satisfied

Transforming: TFR-MAP-004-V00-001
```

Validating: TFR-MAP-004-V00-001
All constraints have been satisfied

Transforming: TFR-MAP-004-V02-001
Validating: TFR-MAP-004-V02-001
All constraints have been satisfied

Transforming: TFR-MAP-005-V00-001
Validating: TFR-MAP-005-V00-001
All constraints have been satisfied

Transforming: TFR-MAP-005-V01-001
Validating: TFR-MAP-005-V01-001
All constraints have been satisfied

Transforming: TFR-MAP-005-V02-001
Validating: TFR-MAP-005-V02-001
All constraints have been satisfied

Transforming: TFR-MAP-005-V02-002
Validating: TFR-MAP-005-V02-002
All constraints have been satisfied

Transforming: TFR-MAP-005-V03-001
Validating: TFR-MAP-005-V03-001
All constraints have been satisfied

Transforming: TFR-MAP-005-V04-001
Validating: TFR-MAP-005-V04-001
All constraints have been satisfied

Transforming: TFR-MAP-005-V04-002
Validating: TFR-MAP-005-V04-002
All constraints have been satisfied

Transforming: TFR-MAP-006-V00-001
Validating: TFR-MAP-006-V00-001
All constraints have been satisfied

Transforming: TFR-MAP-007-V00-001
Validating: TFR-MAP-007-V00-001
All constraints have been satisfied

Transforming: TFR-MAP-007-V01-001
Validating: TFR-MAP-007-V01-001
All constraints have been satisfied

Transforming: TFR-MAP-007-V02-001
Validating: TFR-MAP-007-V02-001
All constraints have been satisfied

Transforming: TFR-MAP-007-V02-002
Validating: TFR-MAP-007-V02-002
All constraints have been satisfied

Transforming: TFR-MAP-007-V03-001
Validating: TFR-MAP-007-V03-001
All constraints have been satisfied

Transforming: TFR-MAP-007-V04-001
Validating: TFR-MAP-007-V04-001
All constraints have been satisfied

Transforming: TFR-MAP-007-V04-002
Validating: TFR-MAP-007-V04-002
All constraints have been satisfied

TEST SUMMARY:

Time elapsed: 206.824 seconds
Tests executed: 30
Hits: 30
Failures: 0 which 0 are errors

+ rm -rf /home/hudson/tests/moskitt/
Terminating xvnc.
$ vncserver -kill :10
Killing Xvnc4 process ID 26799
Finished: SUCCESS

7. Usos futuros

Este proyecto está centrado en la transformación entre modelos de *Sketcher* y *UIM*, aunque también ha sido aplicado a otras transformaciones entre modelos presentes en *MOSKitt* como son *uml2db*, *bpmn2uml* y *dd2cld*.

Uno de los grandes beneficios de este montaje es que es aplicable a cualquier transformación entre modelos que se pueda crear en *MOSKitt*. Así pues, uno de los posibles usos futuros o ampliaciones del proyecto sería adaptarlo sobre el resto de transformaciones que existen a día de hoy en el proyecto *MOSKitt*.

Otra propuesta interesante de cara al futuro sería el poder realizar test sobre modelos gráficos. Comprobar que los diagramas están bien generados, aunque para ello se deberían utilizar tecnologías más complejas, pero la estructura sería la misma.

8. Conclusiones

La automatización de cualquier procedimiento siempre es un voto positivo para el avance de cualquier proyecto. En este caso, la automatización de las pruebas (aunque sólo sea de una parte, las transformaciones de modelos) favorece el rápido desarrollo de la herramienta *MOSKitt*, ya que no se desperdicia tiempo en *testear* una y otra vez.

Además, la inclusión de dichas pruebas en el entorno de construcción de *MOSKitt* aligera y hace aún más automático el proceso; ya que nos permite conocer de antemano, sin necesidad de llegar a probar nada, si el componente que ha sido ensamblado es funcionalmente correcto.

La herramienta desarrollada en el presente proyecto proporciona un entorno que permite automatizar la ejecución de las pruebas sobre transformaciones de modelos, además se han implementado los casos de prueba para la transformación *Sketcher2UIM*.

Para la especificación de los modelos se utiliza *HUTN*, un lenguaje que permite generarlos sólo con detallar las propiedades relevantes para el caso de prueba en que son utilizados. Así se mantienen lo más independientes posible a los cambios de su metamodelo.

Para la validación de los modelos resultantes se ha empleado *EVL*, una extensión de *OCL*.

Todo el proceso está integrado con *JUnit*, que permite ejecutar las pruebas apenas sin esfuerzo y mostrando de forma clara y precisa los resultados de las validaciones.

Con la utilización de *Hudson* para la construcción de componentes de *MOSKitt*, también hemos sido capaces de ejecutar una serie de *scripts* que nos permiten realizar todo este proceso de automatización de las pruebas sin tener que depender de un *Eclipse* con el entorno gráfico levantado.

9. Referencias

- Iturria Sánchez, Héctor

  Automatización de las pruebas sobre transformaciones de modelos y aplicación a la transformación UML2DB (Proyecto Final de Carrera)

- Build and Test Automation for plug-ins and features

  http://www.eclipse.org/articles/Article-PDE-Automation/automation.html

- Wikipedia

  http://es.wikipedia.org/wiki/Apache_Ant

  http://es.wikipedia.org/wiki/Desarrollo_guiado_por_pruebas

  http://es.wikipedia.org/wiki/Eclipse_(software)

  http://es.wikipedia.org/wiki/Framework_de_modelado_Eclipse

  http://es.wikipedia.org/wiki/Hudson_(software)

  http://es.wikipedia.org/wiki/Integración_continua

  http://es.wikipedia.org/wiki/Intérprete_de_comandos

  http://es.wikipedia.org/wiki/JUnit

  http://en.wikipedia.org/wiki/Metamodeling

  http://en.wikipedia.org/wiki/Model-driven_engineering

  http://en.wikipedia.org/wiki/Model_transformation

  http://es.wikipedia.org/wiki/Object_Management_Group

  http://es.wikipedia.org/wiki/Pruebas_de_software

  http://es.wikipedia.org/wiki/Script_(informática)

  http://en.wikipedia.org/wiki/Shell_script

  http://es.wikipedia.org/wiki/Subversion

  http://en.wikipedia.org/wiki/Test_automation

- MOSKitt

  http://www.moskitt.org/

- Epsilon Project

  http://www.eclipse.org/gmt/epsilon/

- Hudson
  http://wiki.hudson-ci.org/display/HUDSON/Meet+Hudson

- Installing software using the p2 director application
  http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/p2_director.html

- Equinox p2 publisher (operations with p2 repositories)
  http://wiki.eclipse.org/Equinox/p2/Publisher

- Automating Eclipse PDE Unit Tests using Ant
  http://www.eclipse.org/articles/article.php?file=Article-PDEJUnitAntAutomation/index.html

- Creating a p2 repository from features and plugins
  http://maksim.sorokin.dk/it/2010/11/26/creating-a-p2-repository-from-features-and-plugins/

- The Eclipse Test Framework
  http://dev.eclipse.org/viewcvs/viewvc.cgi/org.eclipse.test/testframework.html?view=co

- Xvnc plugin for Hudson
  http://wiki.hudson-ci.org/display/HUDSON/Xvnc+Plugin

10. Anexos

10.1. Anexo I: Instalación de Eclipse

1) Requisitos previos

Tener instalado *Eclipse Modeling Tools* 3.5.0 (Galileo)

Disponible aquí: http://www.eclipse.org/downloads/packages/eclipse-modeling-tools-includes-incubating-components/galileor

2) Instalación de plugins

Localizar el fichero *dropins.zip*, que contiene los diferentes *plugins* que extenderán a *Eclipse* (entre ellos *Epsilon*) para disponer de todas las funcionalidades deseadas. Extraer su contenido en una carpeta, la cual contendrá una serie de directorios que serán cada uno de los distintos *plugins*.

Mover todos estos ficheros extraídos a la carpeta *dropins*, dentro de la carpeta de instalación de *Eclipse*.

3) Configuración de Eclipse

Hay que modificar el compilador de *Java* para que utilice la versión 1.5. Para ello, vamos al menú *window* → *preferences*. Una vez aquí, nos situamos sobre la opción *Java* → *Compiler* y en *Compiler compliance* level elegimos 1.5.

Otro paso a realizar para que no haya ningún problema con los *plugins* de *Epsilon* es importar los mismos como proyectos al *workspace*. Para ello, nos situamos en la vista *Plug-in Development*. Localizamos los *plugins* pertenecientes al proyecto *Epsilon* (*org.eclipse.epsilon.**), los seleccionamos todos y con el botón derecho del ratón seleccionamos *import as binary project*.

La **figura 13** muestra el orden a seguir para importar estos *plugins* al *workspace*.

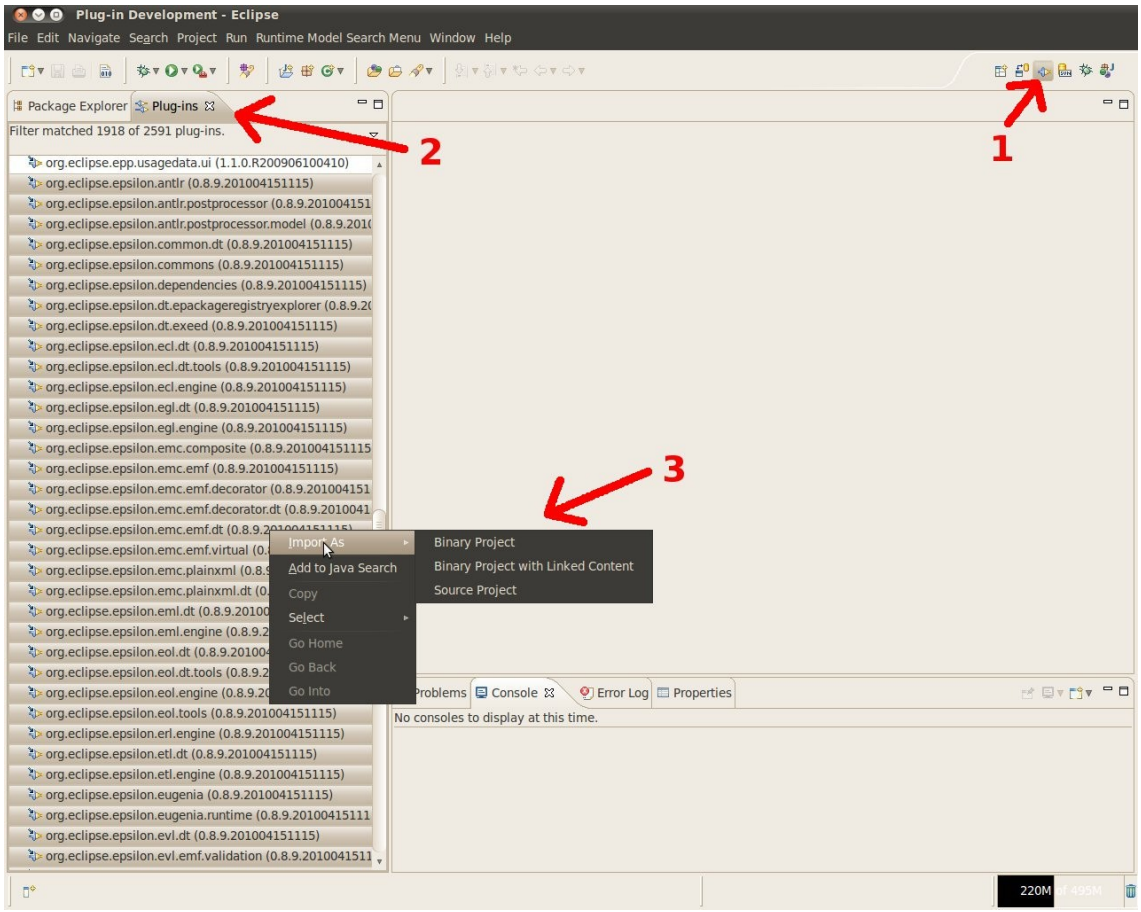


Figura 13. Importación de los plugins de Epsilon al workspace

Una vez realizados dichos pasos, ya estamos en condiciones de empezar a desarrollar casos de prueba para cualquier transformación.

10.2. Anexo II: Guía para la creación de casos de prueba para una transformación de modelos cualquiera

1) Crear proyecto .test

Para empezar, debemos crearnos el proyecto que corresponde a todo el tema de automatización de pruebas.

Se trata de un *plug-in project* que va a contener tanto las plantillas *HUTN* con sus modelos generados, así como los ficheros *EVL* de validación de los modelos resultantes de la transformación, además de las clases necesarias para realizar los test.

Así pues, en primer lugar crearemos 2 carpetas: *configurations*, que contendrá los modelos de configuración de las transformaciones (si son necesarios) y *models*, que contendrá los propios modelos generados en HUTN.

Además, dicho *plugin* contendrá una carpeta *validations*, que es la que contendrá los *EVL*.

También debemos crear una clase *JUnit*, a la que llamaremos *TransformationAndValidation.java*, la cual contendrá un método *setUpBeforeClass()* con la etiqueta *@BeforeClass*, que se ejecutará antes de empezar los test, con todas las variables y parámetros necesarios para identificar las pruebas. También crearemos todas las pruebas como métodos con la etiqueta *@Test*.

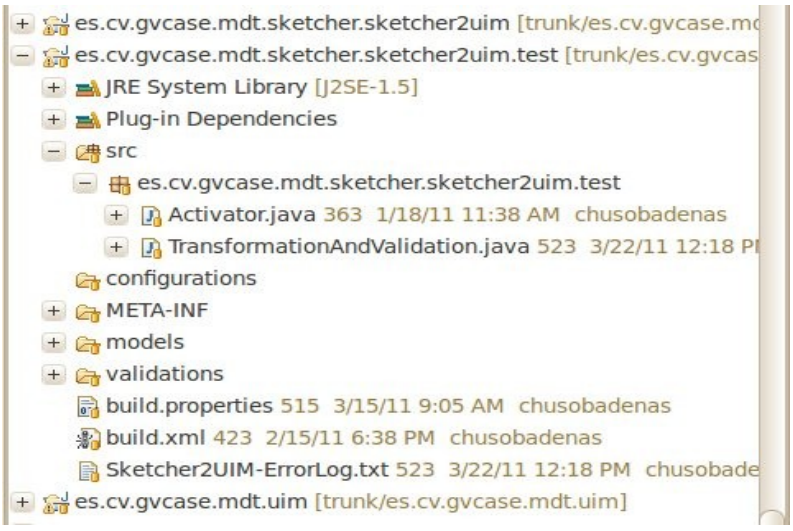


Figura 14. Estructura del plugin de test de la transformación sketcher2uim

Cada uno de los métodos correspondientes a los test, se identificarán con el nombre del caso de prueba correspondiente. Dentro de cada test se realizarán las siguientes acciones:

- Una llamada al método *execute()*, que realiza la transformación del modelo que se le pasa por parámetro y devuelve el resultado de la validación
- Un incremento del número de test realizados, así como de los aciertos o fallos de los mismos
- Una llamada al método *assertFalse()*, que nos dará error si la prueba no ha sido superada con éxito

Por último crearemos un método *tearDownAfterClass()* con la etiqueta @AfterClass que se lanzará al finalizar la ejecución de todos los casos de prueba.

En este método se muestra por pantalla el pequeño sumario o resumen del resultado de los test, es decir, número de test ejecutados, aciertos, fallos y tiempo transcurrido. A su vez, se genera un fichero de texto (*log*) con esta misma información.

2) Crear modelos HUTN

Para crear un archivo *HUTN* haremos *click* en *File → New → Other → Epsilon → HUTN File* y como nombre le asignaremos el identificador del caso de prueba.

Una vez se especifican los elementos del modelo y sus propiedades, se guarda el fichero y se creará de forma automática el modelo (si no lo hace, podemos hacerlo manualmente haciendo *click* sobre el fichero *HUTN → Generate model*).

3) Crear la segunda instancia

(Opcional, en caso de que queramos lanzar los test sobre Eclipse con interfaz gráfica)

Navegamos hasta *Run → Run Configurations...* Aquí dentro creamos una nueva instancia de *Eclipse Application*, y la nombraremos como junit-*nombretransformación*.

Una vez creada, toca configurarla. Dentro de la pestaña *Main*, debemos indicarle el *workspace* que utilizaremos. Aquí debemos crear una carpeta nueva a la que llamaremos con el mismo nombre que le hemos puesto a la instancia.

En la pestaña *plugins* debemos seleccionar la opción *Launch with → plug-ins selected below only,* y aquí seleccionar todos aquellos *plugins* necesarios (dependencias) para la transformación que queramos probar.

4) Ejecutar script ANT

El siguiente paso consiste en copiar los modelos de configuración (si los hubiese) generados en *HUTN* a un directorio excluido del *plugin*. Para ello no hay más que situarse encima del fichero *build.xml* y con el botón derecho del ratón *Run As → Ant Build*.

¡OJO! Es muy importante la configuración del script. Se debe actualizar el nombre de los directorios donde se copiarán los modelos y donde se generarán los modelos resultantes de la transformación. El cambio consiste en poner el nombre del plugin a la carpeta contenida en ${user.home}/tempWorkspace.

5) Crear ficheros de validación EVL

Para crear un nuevo fichero *EVL*, haremos *click* en *File → New → Other → Epsilon → EVL Validation.*

El nombre del fichero corresponderá al identificador de la prueba, y si dicha prueba contiene un patrón de configuración asociado, se le añadirá al final del nombre el número del patrón (el nombre del fichero coincidirá con el nombre asociado al modelo *HUTN* de configuración si tiene; si no, al nombre del modelo de entrada).

6) Crear JUnit Plug-in Test

(Opcional, en caso de que queramos lanzar los test sobre Eclipse con interfaz gráfica)

Volvemos a *Run Configurations...* y ahora debemos crear un *Junit Plug-in Test*, cuyo nombre será junit-*nombretransformacion*_test.

En la pestaña *Test*, debemos seleccionar como *Project* nuestro plugin de test creado. Como *Test Class* elegiremos la clase *TransformationAndValidation* (en *Test Runner* debemos poner la versión *Junit 4*).

En la pestaña *Main*, debemos seleccionar como *workspace* el mismo que utilizamos para crear la segunda instancia. Además, en la sección programa a ejecutar, selecionaremos “Run an application: *[No application] Headless mode*”. Esto es debido a que, pese a tener que lanzar una segunda instancia, no es necesario que se levante la interfaz.

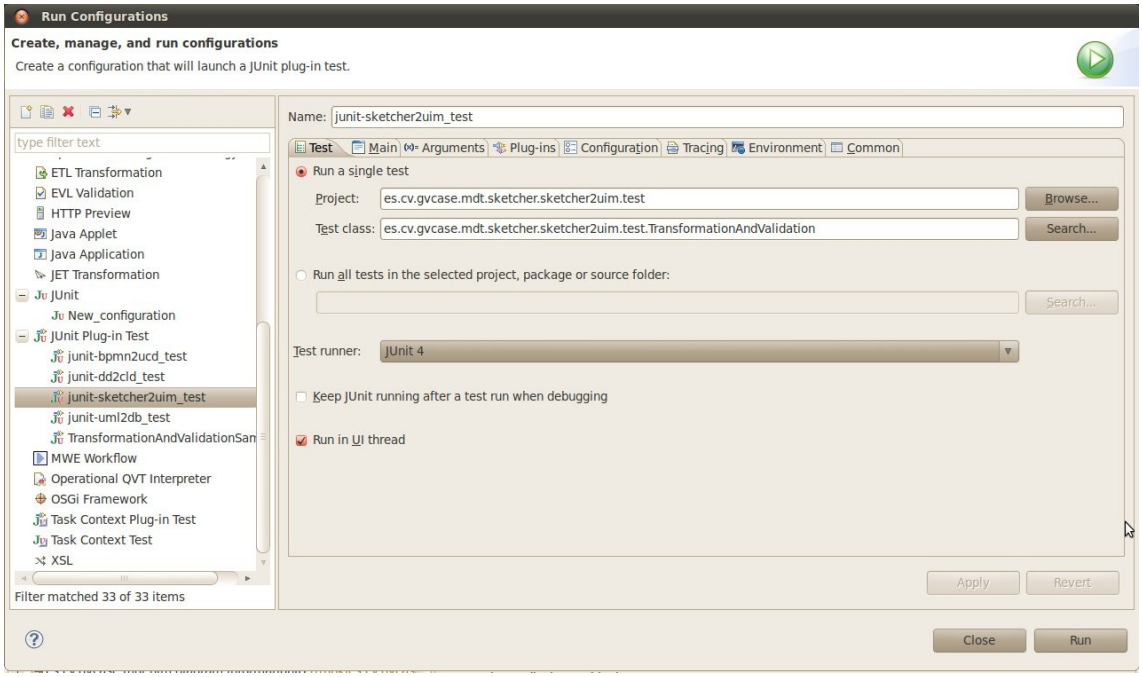


Figura 15. Pestaña Test (Run Configurations)

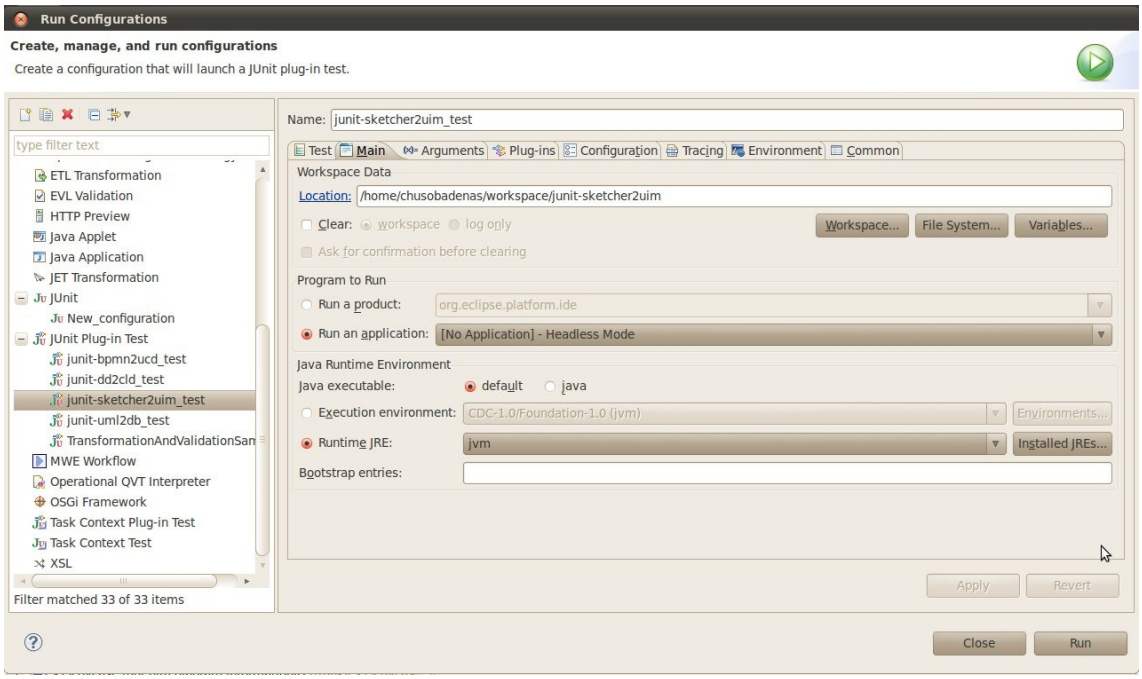


Figura 16. Pestaña Main (Run Configurations)

Por último en la pestaña *Plug-ins*, seguiremos el mismo procedimiento que hicimos con la segunda instancia, es decir, seleccionaremos sólo aquellos *plugins* que sean necesarios para hacer posible la transformación y que además coinciden con los asignados a la segunda instancia.

Al realizar todos estos pasos, ya estaremos preparados para lanzar las pruebas. Simplemente con ejecutar el *Junit Plug-in Test* creado, veremos como en la vista *JUnit* de *Eclipse* aparecerán los distintos test creados y por la consola obtendremos el resultado de las validaciones.

10.3. Anexo III: Preparación del entorno de construcción

Descripción

El entorno de construcción de los módulos funcionales de *MOSKitt* está formado por 2 partes:

- *Basebuilder*, o *Eclipse headless*: no es más que el *Eclipse* sin el entorno gráfico (*workbench*).
- *Builder:* una serie de *plugins* que definen unas tareas con *Ant* que realizan la compilación y la construcción de los módulos.

Requisitos previos

A continuación se listan la serie de características que deben estar instaladas en la máquina donde se vaya a realizar la construcción:

1. Cliente CVS
    - Para instalarlo, podemos ejecutar en un terminal *sudo apt-get install cvs*.

2. Cliente SVN (*Subversion*)
    - Para instalarlo, podemos ejecutar en un terminal *sudo apt-get install subversion*.

Es necesaria la versión 1.6 o superior. Para conocer la versión podemos ejecutar en la terminal *svn --version*.

3. Máquina virtual Java
    - Guía de instalación aquí.

Montaje del entorno

Seguiremos una serie de pasos para tener montado el entorno y poder construir:

- Instalar la plataforma *Eclipse* y los requisitos correspondientes a la versión a construir (ver *Requisitos previos*).
- Crear el directorio de construcción. Un buen sitio para ubicarlo es en *$HOME/moskitt-galileo-dev/build*
- Descargar el framework *basebuilder* correspondiente a la plataforma.

Situados en el directorio de construcción, ejecutamos (suponemos versión 3.5.0):

```
cvs -d :pserver:anonymous@dev.eclipse.org:/cvsroot/eclipse export -r R3_5 org.eclipse.releng.basebuilder
```

NOTA: Otro modo de realizar este paso es a través de *Eclipse*, mediante su *plugin CVS*. Para ello, nos creamos un *workspace* vacío y luego añadimos un nuevo repositorio con los siguientes parámetros:

- *Host*: dev.eclipse.org
- *Repository path*: /cvsroot/eclipse
- *User*: anonymous

Una vez conectados al repositorio navegamos hasta *Versions* → *org.eclipse.releng.basebuilder* → *org.eclipse.releng.basebuilder R3_5*

Una vez aquí, haremos un *checkout*. Cuando estén todos los fuentes descargados en el *workspace*, debemos copiar ese contenido al directorio de construcción. En primer lugar desconectamos del servidor mediante T*eam* → *Disconnect* → *Also delete the SVN meta information from the file system*. Ahora ya podemos copiar dicho contenido al directorio de construcción.

- El proyecto *org.eclipse.releng.basebuilder* no tiene acceso a *subversion*. Para solventarlo añadiremos esa funcionalidad descargando y descomprimiendo en el directorio de construcción el *plugin* disponible aquí.

- Descargar el proyecto *releng.builder* de *gvcase* ejecutando (en el directorio de construcción):

```
svn export https://svn.gvcase.org/svn/gvcase-releng/trunk/es.cv.gvcase.releng.builder
```

- Sólo nos queda indicar dónde se sitúa la plataforma. Dentro del directorio *$HOME/moskitt-galileo-dev* creamos un enlace al directorio de instalación de eclipse mediante la orden *ln -s*.

Llegados a este punto, ya tenemos todo configurado para empezar a construir.

Iniciando la automatización de la construcción

El método de realizar una construcción es a través de la ejecución del fichero *build.sh* contenido en *$HOME/moskitt-galileo-dev/build*. En la llamada, se pueden incluir varios parámetros que caracterizarán la construcción.

El modo de realizar una construcción con todos los parámetros por defecto es mediante:

```
./build.sh -Dcomponent=nombrecomponente
```

donde *nombrecomponente* es el módulo que queramos construir.

Como podemos observar, dentro del *builder* encontramos una carpeta para cada componente o módulo de *MOSKitt*. Más adelante se explicará cómo añadir componentes nuevos.

El resultado de la construcción con los parámetros por defecto es el siguiente:

- *tempbuild*: Directorio de trabajo para la construcción. Dentro de él podremos observar un directorio etiquetado con la versión del *build* correspondiente (*Ixxxxxxxx*) que contendrá el repositorio *p2* en formato *.zip*.

- *buildRepo*: El repositorio *p2* del módulo construido. Es un directorio que se encuentra dentro de *tempbuild*. Mediante un parámetro de entrada es posible una diferente ubicación.

- *maps*: Aquí se encuentran los *maps* de cada componente. No es más que un modo de especificar dónde están situados los fuentes de cada componente y todas sus dependencias. Situado también dentro de *tempbuild*.

- *directory.txt*: Incluye el contenido de todos los *maps* presentes en la carpeta maps.

Parámetros de entrada

Como ya se ha comentado, a la ejecución se le pueden añadir una serie de parámetros por consola, que determinarán el resultado final.

Una cosa a tener en cuenta es que para escribirlos deben empezar todos por *-D* seguido del nombre del parámetro (sin espacios, ejemplo: *-DrepoBaseLocation*).

Los más importantes se detallan a continuación:

- *buildDirectory*: define el directorio temporal donde se construye. Por defecto es *$HOME/moskitt-galileo-dev/build/es.cv.gvcase.releng.builder/tempBuild*.

La ruta debe ser absoluta. Lo podemos encontrar en el fichero *build.xml*.

- *baseLocation*: define el directorio donde se encuentra la plataforma. Por defecto es *$HOME/moskitt-galileo-dev/eclipse*.

Lo podemos encontrar en el fichero *build.xml*.

- *tagPath*: define de dónde se sacan los fuentes del módulo. Por defecto es */trunk*.

Lo podemos encontrar en el fichero *customTargets.xml*.

- *svnurl*: define el directorio de donde se sacan los *maps*.

Lo podemos encontrar en el fichero *customTargets.xml*.

- *p2.build.repo*: define el directorio donde se genera el repositorio p2. Por defecto es *$HOME/moskitt-galileo-dev/build/es.cv.gvcase.releng.builder/tempBuild/buildRepo*.

Hay que añadirle delante de la ruta la cadena *file:* y además la ruta debe ser absoluta. Lo podemos encontrar en el fichero *build.xml*.

- *repoBaseLocation*: define el directorio donde se encuentran el resto de repositorios *p2* (dependencias) que sirven para poder generar el nuevo componente. Por defecto es *$HOME/moskitt-galileo-dev/build/es.cv.gvcase.releng.builder/buildRepo*.

Lo podemos encontrar en el fichero *build.xml*.

- *componentsLocation*: define el directorio donde se hará una copia del repositorio generado.

- *forceContextQualifier*: se trata del número de *build* que se quiera dar al componente. El formato es *yyyymmddhhnn* (año, mes, día, hora y minuto).

Componente con/sin dependencias

Si lo que queremos es construir un módulo o componente que no dependa de ningún otro, lanzaremos el script *changeLines* con el parámetro de entrada *comment*.

En caso de tener un componente que dependa de algún otro, el parámetro de entrada será *nocomment*.

Creando un nuevo componente

Si lo que queremos es crear un nuevo módulo del cual tengamos todas sus *features* y *plugins* y queramos construirlo, debemos crear una nueva carpeta con el nombre del módulo dentro de *$HOME/moskitt-galileo-dev/build/es.cv.gvcase.releng.builder*.

Dentro deberán existir 3 ficheros:

- allElements.xml
- build.properties
- customTargets.xml

Lo más rápido es copiarnos los ficheros de otro componente y editarlos a partir de ahí. Simplemente debemos entrar en cada fichero y substituir el nombre del componente por el que estamos utilizando.

Además, debemos crearnos un nuevo *.map* en el que indiquemos los *features* y *plugins* que formen parte del módulo y su localización en el repositorio *subversion*.