Proyecto Final de Carrera

Diseño y Desarrollo de una Herramienta CASE para la Generación Automática de Código para Sistemas Pervasivos

Julio del 2006, Valencia

Carlos Cetina Englada
ccetina@dsic.upv.es

Dirigido por: *Dr. Vicente Pelechado Ferragud*
pele@dsic.upv.es

Indice

1 Introducción 4

2 Contexto 7
2.1 Método de producción de Sistemas Pervasivos 7
2.2 Tecnología utilizada 8

3 Descripción del lenguaje PervML 12
3.1 Elementos Comunes 12
3.2 Modelo de Servicios 13
3.3 Relaciones entre Servicios 13
3.4 Diagrama de Transición de Estados 16
3.5 Modelo Estructural 16
3.6 Modelo de Interacción 17
3.7 Modelo de Proveedores de Enlace 18
3.8 Especificación Estructural de Componentes 19
3.9 Especificación Funcional de Componentes 19

4 Descripción del framework 20
4.1 Estructura del Framework 20
4.2 Estrategia Global de Ejecución 24
4.3 Instanciación del Framework 29

5 Gestión de modelos 33
5.1 Tareas necesarias para la gestión de modelos 33
5.2 Tecnología utilizada 33
5.3 Creación de un modelo ecore 35
5.4 Clases Java representativas de elementos del metamodelo 39
5.5 Clases adaptadoras 41
5.6 Editor simple de modelos PervML 43
5.7 Persistencia de modelos 44
5.8 Clases test 45

6 Edición Gráfica de modelos 46
6.1 Tecnología utilizada 47
6.2 Recursos graficos 56
6.3 Vista Modelo servicios 59
6.4 Vista Modelo diagrama transición de estados 60
6.5 Vista Modelo componentes 61
6.6 Vista Modelo Binding providers 62
6.7 Vista Modelo estructura componentes 63
6.8 Restricciones 64
6.9 Unión de las vistas 65

Indice

7 Generación 69
7.1 Tecnología utilizada 69
7.2 Ficheros derivados 71
7.3 Punto de entrada a la ejecución 72
7.4 Parte común (operaciones) 77
7.5 Modelo servicios 79
7.6 Modelo componentes 85
7.7 Modelo Binding providers 89
7.8 Modelo Interacción 93

8 Caso de estudio 98
8.1 Requisitos del Sistema 98
8.2 Identificación de los Servicios del Sistema 98
8.3 El Modelo de Servicios (Descripción de los Tipos de Servicios) 99
8.4 El Modelo Estructural (Los Servicios del Sistema) 104
8.5 El Modelo de Interacción 106
8.6 El Modelo de Proveedores de Enlace 106
8.7 La Especificación Estructural de Componentes 108
8.8 La Especificación Funcional de Componentes 111
8.9 Despliegue 115
8.10 Sistema generado en funcionamiento 117

9 Conclusiones 119
9.1 Resumen del trabajo realizado 119
9.2 Calidad 119
9.3 Esfuerzo 120
9.4 Conclusiones para futuros proyectos 120
9.5 Publicaciones 120

1 Introducción

En este proyecto fin de carrera se ha desarrollado una herramienta de generación automática de código a partir de modelos para sistemas pervasivos. La herramienta sigue el enfoque de desarrollo de software dirigido por modelos, según cual el modelo de un sistema no es únicamente utilizado como documentación, sino que es trasformado en la implementación del sistema. La herramienta genera código para un tipo de sistema software específico: los sistemas pervasivos. En estos sistemas los elementos de computación desaparecen del entorno pero su funcionalidad continúa disponible para los usuarios. La esencia de esta visión fue descrita inicialmente por Weiser [1] a principio de la década de los 90.

La herramienta desarrollada en este proyecto final de carrera da soporte a la especificación gráfica del sistema pervasivo mediante modelos y la generación automática de código a partir de los modelos siguiendo el método de producción de sistemas pervasivos propuesto en [4]. En ese artículo se propone el lenguaje de modelado al que da soporte la herramienta para especificar estos sistemas (PervML) y un framework de implementación que permite la ejecución del código generado.

Figura 1: Herramienta desarrollada y sistema generado.

En la parte superior de la figura 1 se muestra parte del modelado de un sistema utilizando la herramienta y en la parte inferior se muestra el sistema en ejecución, que ha sido generado automáticamente.

Este proyecto se ha desarrollado en el contexto del grupo de investigación OO-METHOD[2], dentro de la línea de investigación Software Engineering and Pervasive Seaps, dirigida por Dr. Vicente Pelechado Ferragud, mediante una beca de colaboración con grupos de investigación financiada por el Ministerio de Educación, Cultura y Deporte.

Una versión industrial de esta herramienta académica proporcionaría al desarrollo de sistemas pervasivos los siguientes beneficios:

- La adaptación de sistemas pervasivos a nuevas tecnologías de implementación, es mas fácil utilizando modelos que trabajando con código de bajo nivel de abstracción. Esta característica es especialmente importante en sistemas pervasivos, donde continuamente están apareciendo nuevas tecnologías.
- La productividad en el desarrollo se incrementa al trabajar los desarrolladores con conceptos del dominio evitando detalles técnicos.
- El código generado es de gran calidad, al embeberse las buenas prácticas de programación en las reglas de transformación de modelos PervML a código. Actualmente los desarrolladores deben preocuparse de aplicar estas buenas practicasen el desarrollo de cada proyecto.

Para probar la validez de la herramienta se ha utilizado en el desarrollo satisfactorio de un caso de estudio completo como se describirá en un capitulo de la documentación.

Con el objeto de mostrar todo lo desarrollado en este proyecto final de carrera, la presente documentación se estructura en los siguientes capítulos:

- Contexto. En este capitulo es descrito brevemente el trabajo de investigación dentro del cual se enmarca este proyecto final de carrera y se comentan las tecnologías utilizadas.

- Descripción del lenguaje PervML. En este capitulo se da una descripción detallada del lenguaje PervML. Presentado su metamodelo acompañado de una descripción textual.

- Descripción del framework. En este capitulo se describe el framework que da soporte al lenguaje PervML. Introduciendo su estructura, contando con detalle la estrategia de ejecución y proporcionado ejemplos para su uso. La implementación de este framework se ha desarrollado de forma cooperativa con el proyecto fin de carrera “Desarrollo de un Sistema de Gestión del Hogar Digital aplicando un enfoque Dirigido por Modelos” con código II-B-DSIC-110/05.

- Arquitectura de la herramienta. En este capitulo se muestran los módulos en los que se divide la herramienta y se proporciona una vista detallada de la arquitectura de cada uno de ellos.

- Gestión de modelos. En este capítulo se identifican las tareas necesarias en la gestión de modelos y se describe desde un punto de vista técnico la solución utilizada.

- Edición Gráfica de modelos. En este capitulo se describen técnicamente las tecnologías utilizadas para el desarrollo del editor grafico de modelos, se justifica su utilización y se describe la relaciones entre los elementos de los modelos y sus representaciones graficas.

---

[2] http://oomethod.dsic.upv.es

- Generación. En este capitulo se introduce el lenguaje de transformación de modelos a archivos de texto, empleado para realizar la generación de código, se indica a partir de que partes del lenguaje se generaran los diferentes ficheros de código y finalmente se muestran las reglas utilizadas para la transformación.

- Caso de estudio. En este capitulo se describe un caso de estudio utilizado para validar el funcionamiento de la herramienta de generación de código. Mostrando el modelado del sistema, acompañado de pequeñas descripciones. Se indica los pasos necesarios para la puesta en marcha del sistema y se muestran imágenes del sistema en funcionamiento.

- Conclusiones. En este capitulo se hace un pequeño resumen del proyecto, se enumeran las conclusiones para futuros proyectos y se relata la experiencia personal.

2 Contexto

2.1 Método de producción de Sistemas Pervasivos

Actualmente muchos de los sistemas pervasivos son desarrollados utilizando técnicas de bajo nivel de abstracción. Es posible aplicar técnicas de ingeniería software en el área de sistemas pervasivos, con el objetivo de desarrollar sistemas de gran calidad de una manera eficiente.

Aproximación utilizada

En [4], se describe un enfoque dirigido por modelos para el desarrollo de los sistemas pervasivos, utilizando las últimas tendencias en ingeniería del software. (MDA y Factorías de software). Las factorías de software promueven la reutilización de arquitecturas, componentes y saber hacer, para el desarrollo de sistemas con características similares. Por otro lado MDA propone la especificación de modelos, siguiendo estándares de la OMG, y su transformación en otros modelos o sistemas completos.

La propuesta para el desarrollo de sistemas pervasivos de [4] se apoya en las fortalezas de las dos aproximaciones anteriores:

- De MDA se centra en los modelos de alto nivel de abstracción, proponiendo el lenguaje de modelado **PervML**, y en la generación de código.
- De las factorías de software utilizan el concepto de reutilización en dominios específicos, plasmándolo en un **framework de implementación**.

Especificar sistemas pervasivos en modelos de altos nivel de abstracción y generar el código de implementación a partir de ellos, proporciona varios beneficios:

- Facilita la adaptación del sistema pervasivo a nuevas tecnologías de implementación. Esta característica es realmente importante en el desarrollo de sistemas pervasivos, donde las tecnologías de implementación están continuamente evolucionando.

- El desarrollo de los sistemas pervasivos es más intuitivo, al trabajarse con conceptos cercanos al dominio evitando detalles de implementación.

- La aplicación de buenas prácticas de implementación se embebe en el framework de implementación y en las trasformaciones de modelo a código.

La herramienta desarrollada en este proyecto final de carrera proporciona un editor grafico para la creación y manipulación de modelos PervML y es capaz de compilarlos a código java. Este código generado es una instanciación del framework de implementación.

Método

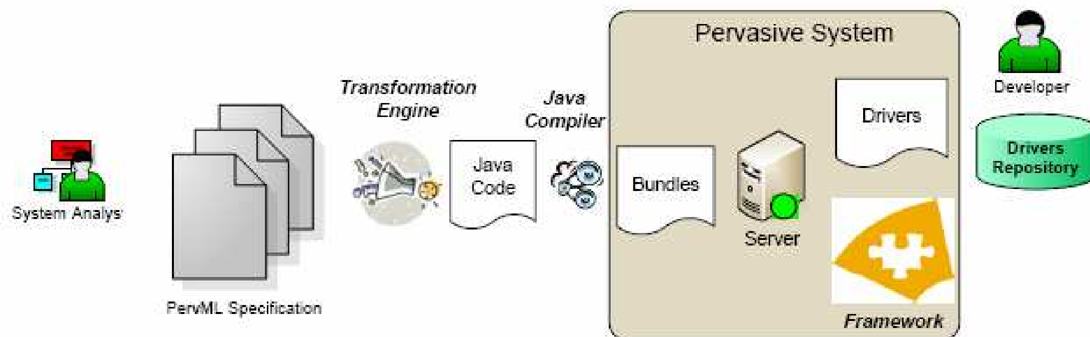


Figura 2: Pasos para producir un sistema pervasivo.

Siguiendo este enfoque dirigido por modelos, los pasos que debería dar un equipo de desarrollo para producir el software de un sistema pervasivo son:

1) El analista capturaría los requisitos del sistema y construiría los modelos gráficos que describen el sistema pervasivo.
2) Se aplicaría el motor de trasformación a los modelos del sistema, generando código java y otros recursos como resultado.
3) Se desarrollarían los drivers que proporcionan acceso a los dispositivos o al software externo al sistema.
4) Se deben ajustar los archivos java para poder utilizar los drivers deseados. El ajuste únicamente implicaría indicar el ID del driver a utilizar.
5) Finalmente el código java se compila y empaqueta en bundles (archivos jar).Para el funcionamiento del sistema, en el servidor debe haber una implementación del framework y los drivers necesarios.

Tanto la fase 1) de construcción de los modelos, como la fase 2) de generación del código java, serian realizadas utilizando la herramienta CASE para la generación de código desarrollada en este proyecto final de carrera.
Nótese que el objetivo de la herramienta y el método es generar la parte software del sistema pervasivo. No la instalación física de los dispositivos, redes… etc.

2.2 Tecnología utilizada

Para el desarrollo de la herramienta CASE para la generación automática de código en entornos de sistemas pervasivos, se han utilizado las siguientes tecnologías:

Plataforma Eclipse

Eclipse inicialmente fue el IDE de IBM para desarrollar aplicaciones utilizando el lenguaje de programación Java, el cual fue liberado como software libre. Actualmente es la plataforma base para muchas otras tecnologías y proyectos debido a su potente estructura modular. Para extender la funcionalidad de la plataforma se utilizan mecanismos de carga de plug-ins en el entorno.

La Figura 3 muestra cómo es el interfaz gráfico y los componentes genéricos que tenemos a nuestra disposición.

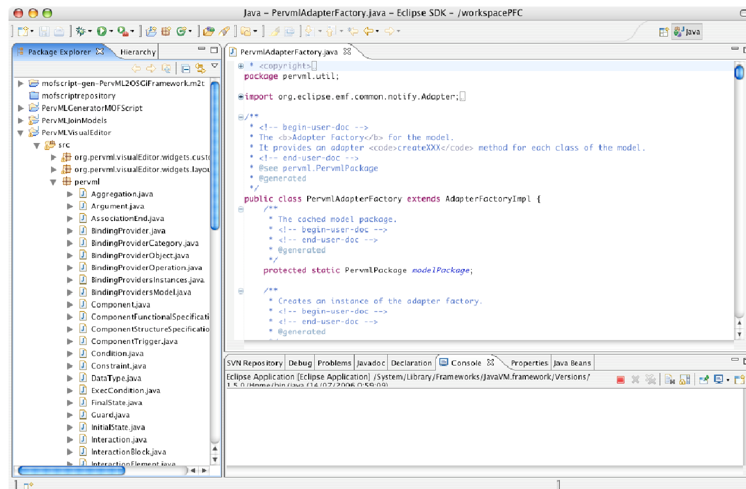


Figura 3: Interfaz grafica de eclipse.

Arquitectura de plug-ins

Un plug-in es la unidad mínima de funcionalidad de Eclipse que puede ser distribuida de manera separada. Herramientas pequeñas se escriben como un único plug-in, mientras que en las complejas la funcionalidad está en varios plug-ins. Excepto un pequeño núcleo de la plataforma Eclipse, el resto de la funcionalidad de la plataforma está implementada como plug-ins. Los plug-ins están escritos en Java. Un plug-in está formado por un JAR de código Java, ficheros de lectura y otros recursos como imágenes, catálogos de mensajes, librerías de código nativo, etc.

Cuando la Plataforma se inicializa, se le muestra al usuario además del entorno de Eclipse todos los plug-ins que tenga instalados. La experiencia del usuario depende de cómo se integren los diferentes plug-ins con la Plataforma y cómo estos puedan comunicarse entre sí.

Página web: http://www.eclipse.org/

EMF

El Eclipse Modeling Framework (EMF) es plug-in de Eclipse que facilita la creación de herramientas y otras aplicaciones basadas en un modelo de datos estructurados. EMF se inició como una implementación del estándar Meta Object Facility (MOF) 2.0 del Object Management Group (OMG), a partir del cual ha evolucionado. Actualmente EMF puede considerarse como una versión de MOF 2.0. La versión estable de EMF es la 2.0.
IBM es el principal desarrollador de EMF. Este framework se inició como un desarrollo interno de la compañía para la gestión de modelos en la serie de productos Websphere Studio. IBM liberó EMF como código abierto en otoño de 2002 y desde entonces forma parte del proyecto Eclipse.org

Para el desarrollo del proyecto se han aprovechado las facilidades que proporciona para la generación de clases java que representen modelos estructuras y su capacidad para almacenar los modelos de forma nativa en XMI 2.0.

Actualmente varios proyectos/productos utilizan EMF:

- Websphere Studio de IBM: IDEs para el desarrollo de todo tipo de productos software. (http://www-306.ibm.com/software/info/websphere/r/studio/)
- openArchitectureWare: paquete de herramientas libres para el desarrollo dirigido por modelos. (http://architecturware.sourceforge.net/)
- Omondo EclipseUML: Herramienta de modelado UML (http://www.omondo.com/)
- Lomboz: Extensión de eclipse con asistentes para el desarrollo de aplicaciones J2EE. (http://www.objectlearn.com/)
- UML 2.0 metamodel implementation: Metamodelo completo de UML 2.0 implementado con EMF. (http://www.uml2.org/)

Página web: http://www.eclipse.org/emf/

GEF

El Graphical Editing Framework (GEF) es un plug-in de Eclipse para el desarrollo de editores gráficos de modelos. Básicamente, proporciona una infraestructura para desarrollar la componente Controller de aplicaciones que sigan el patrón arquitectónico Model-View-Controller [6]. Aunque GEF no está ligado a ninguna librería, la manera más natural es trabajar con Draw2D en la parte "View" y EMF en la parte "Model".
Podría considerarse que el objetivo de GEF es facilitar la tarea de aplicaciones basadas en la estructuración que se muestra en la siguiente figura:

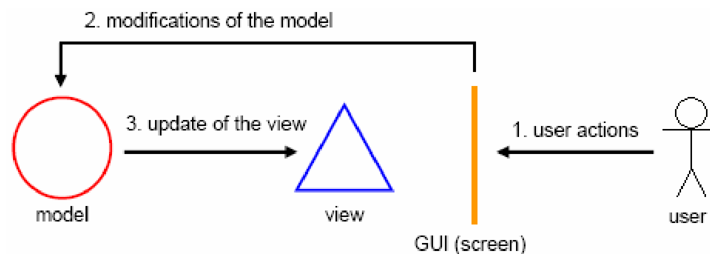


Figura 4: Interacción del usuario en GEF.

El objetivo de utilizar este plug-in en la herramienta es proporcionar a la herramienta desarrollada en este proyecto final de carrera una buena arquitectura de código, en la estén embebidas buenas prácticas de diseño.

Página web: http://www.eclipse.org/gef/

GMF

El Graphical Editing Framework (GMF) es un nuevo proyecto de Eclipse que pretende proporcionar mecanismos para facilitar la creación de editores gráficos a partir de EMF y GEF. La filosofía general es partir de modelos del dominio y definiciones de diagramas para generar un editor visual. Este proyecto está siendo liderado por Borland y cuenta con el interés de IBM Rational, openArchitectureWare, la Technical University of Berlin o la Vanderbilt University (desarrolladores de GME) entre otros.

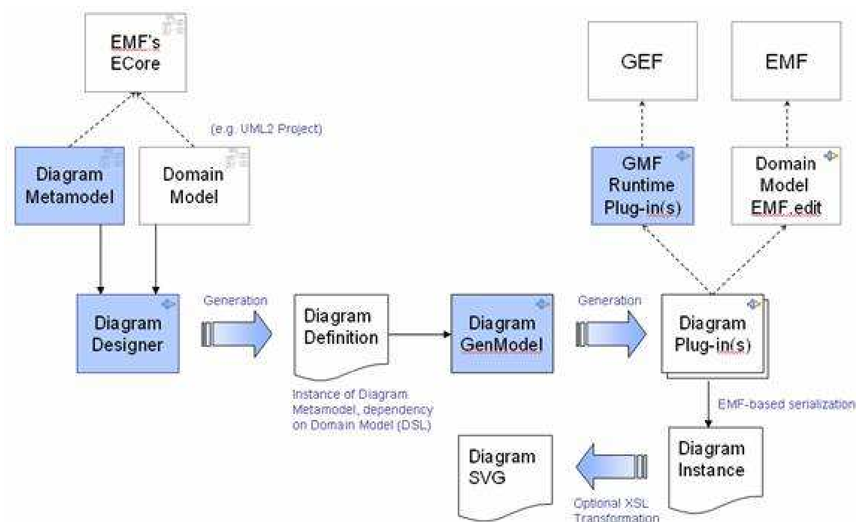


Figura 5: Proceso de generación en GMF.

GMF permite describir de una forma declarativa, mediante el uso de modelos, las asociaciones entre elementos de un modelo y representaciones visuales en un diagrama. Generando a partir de los modelos que contiene las definiciones de las asociaciones un editor grafico GEF, en el que para cada representación visual se generan comportamientos por defecto de creación, modificación y eliminación. Esta aproximación nos abstrae de los detalles tecnológicos de GEF permitiendo centrarse en la definición de las asociaciones.

La utilización de este plug-in en el proyecto viene justificada por el gran incremento que se obtiene en la productividad y por la garantía de calidad en el código generado para el editor gráfico.

Página web: http://www.eclipse.org/gmf/

MOFScript

La herramienta MOFScript es un plug-in de eclipse que implemeta el lenguaje de transformación de modelo a texto MOFScript. Actualmete el lenguaje MOFScript es un candidsto en el proceso RFP de la OMG para determinar el lenguaje de transformación de modelo a texto para Meta Object Facility.

MOFScript permite definir conjuntos de reglas en las que se indican como realizar transformaciones desde un modelo estructurado a texto. Como modelo de entrada es posible utilizar modelos soportados por el plug-in EMF y no existe ninguna restricción en el texto generado.

En el contexto del proyecto, este plug-in se utilizara para definir las transformaciones que generaran el código java a partir de los modelos de PervML.

Página web: http://www.eclipse.org/gmt/mofscript/

3 Descripción del lenguaje PervML

En este capitulo se describe PervML, para ello se presenta su metamodelo acompañado de una descripción textual. El metamodelo de PervML se estructura en diez paquetes, cada uno de los cuales define una vista del lenguaje. El paquete CommonElements contiene elementos que son utilizados en varios paquetes y que se han agrupado para evitar su definición replicada en varias partes del metamodelo.

Para describir un sistema pervasivo con PervML a alto nivel se utilizarían los paquetes ServicesModel, ServicesRelationships, StructuralModel, STD e InteractionModel, mientras que para un sistema concreto se deberían utilizar además los elementos de los paquetes BindingProvidersModel, ComponentStructuralSpecification y ComponentFunctionalSpecification. El paquete Main contiene la raíz del metamodelo; es decir, el elemento que define qué es una descripción de un sistema pervasivo.

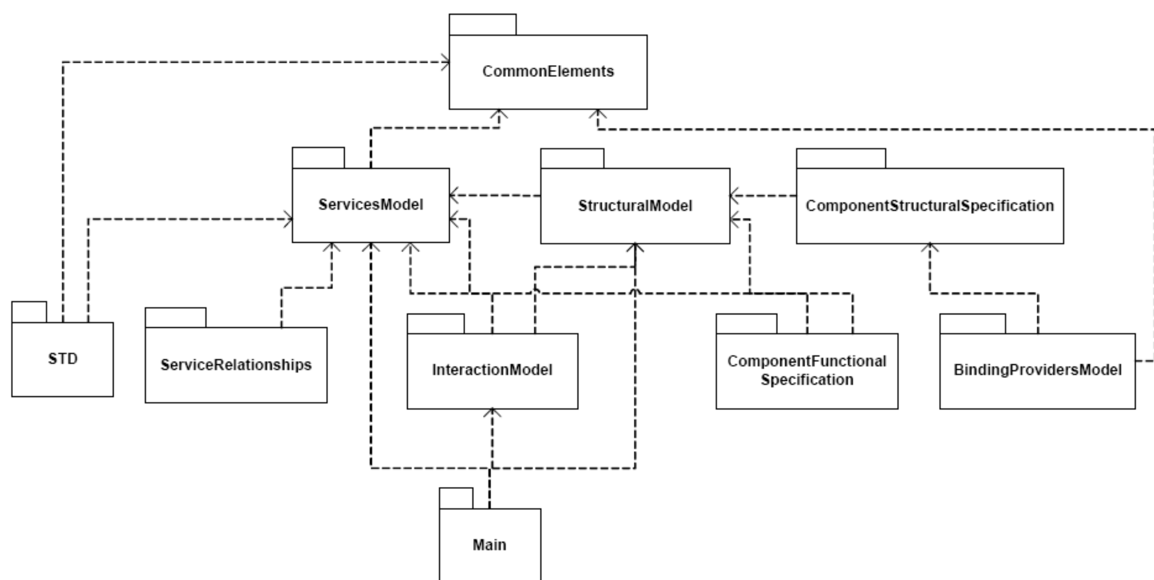


Figura 6: Organización de los paquetes que componen el metamodelo de PervML.

3.1 Elementos Comunes

El paquete CommonElements contiene elementos para la descripción de operaciones, restricciones y disparadores que serán utilizados por otros paquetes. En el paquete se especifica que toda operación (Operation) podría tener varios argumentos (OperationDatum) y un valor de retorno definido por su tipo (DataType). Las operaciones también podrían estar restringidas por precondiciones y poscondiciones. Una precondición es una restricción (Constraint) que debe satisfacerse (ser cierta) antes de que se realiza la operación.

Una poscondición es una condición que debe ser cierta tras realizar la operaciones. Por otra parte se encuentran los disparadores (Triggers), que representan condiciones que cuando son satisfechas activan alguna funcionalidad del sistema. Tanto las restricciones como los disparadores son expresiones definidas en OCL.

3.2 Modelo de Servicios

El paquete ServicesModel contiene los elementos para la descripción de los servicios del sistema. Como puede observarse en la figura 7, se utilizan varios de los elementos definidos en el paquete CommonElements mediante el mecanismo de especialización. En definitiva, se describe que un servicio (Service) está formado por operaciones (ServiceOperation) y propiedades (ServiceProperty).

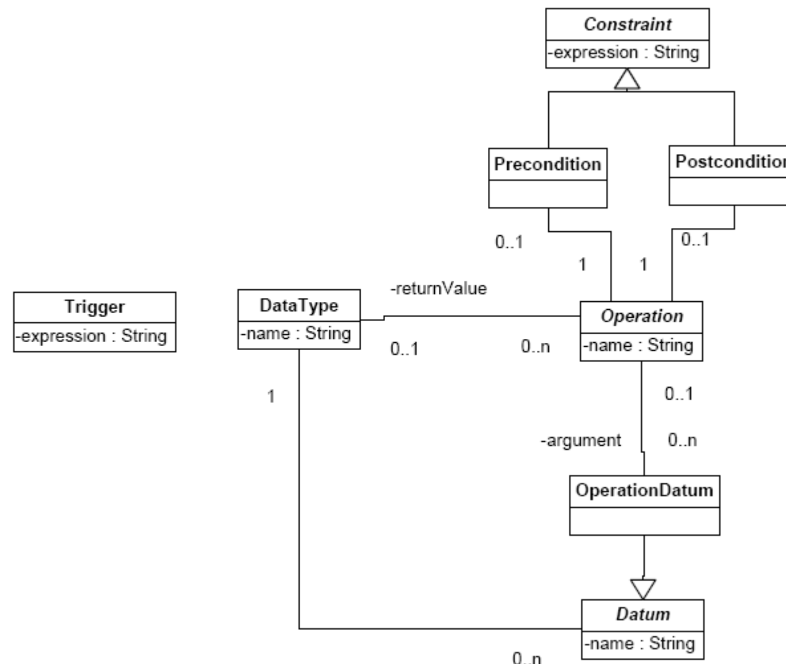


Figura 7: Contenido del paquete 'CommonElements'

También es posible definir disparadores (ServiceTrigger) y restricciones de integridad IntegrityConstraint) sobre el servicio. Además, todos los servicios pertenecerán al menos a una categoría (ServiceCategory), las cuales se encuentran formando una estructura jerárquica. Finalmente, el servicio tiene asociado un diagrama de transición de estados, que se encuentra definido en el paquete STD.

3.3 Relaciones entre Servicios

El paquete ServicesRelationships contiene los elementos para la descripción de las relaciones entre los servicios del sistema. Se pueden definir dos tipos de relaciones: de agregación o de especialización.

Caracterización de la Agregación entre Servicios

La relación de agregación es una relación binaria utilizada para describir que el funcionamiento de un tipo de servicio implica por su propia naturaleza la utilización de servicios de otro tipo determinado. Para dotar a esta relación de una semántica precisa, a continuación se describen varias características a partir de la caracterización de relaciones de asociación propuesta en [2].

Figura 8: Contenido del paquete 'servicesModel '.

Figura 9: Contenido del paquete 'ServicesRelationships'

Visibilidad: La existencia de una agregación entre servicios supone que desde el servicio agregador puede accederse a las operaciones y propiedades del servicio agregado. De manera inversa esto no es cierto; es decir, el servicio agregado no tendría conocimiento de su agregador.

Multiplicidad: Un servicio puede agregar a varios servicios. Recíprocamente un servicio podrá ser agregado por varios servicios. Por tanto, en el modelo de servicios se deberá establecer cuales son las multiplicidades mínimas y máximas de cada extremo de la agregación.

Reflexividad: Podrán establecerse relaciones de agregación reflexivas, en las que un servicio se pueda agregar de otros servicios del mismo tipo. En este caso, la multiplicidad mínima de uno de los dos extremos deberá ser cero.

Características temporales: Pueden modificarse durante el funcionamiento del sistema los enlaces entre componentes que ofrezcan servicios agregados para incluir nuevos componentes o eliminarlos de la agregación.

Propagación de borrado y Proyección de identificación: Todos los servicios tienen entidad propia dentro del sistema pervasivo, por lo que no dependen del servicio agregado para existir. Esto hace que cuando se elimine un servicio agregador, el que agrupa otros servicios, únicamente se deban eliminar los enlaces con sus servicios agregados, permaneciendo los

servicios en el sistema. Por la misma rascón, no existirá proyección de identificación; es decir, los servicios agregados no se identificarían también por su servicio agregador.

Antisimetría: Esta propiedad dependerá de cada agregación, por lo que deberá especificarse en todas las agregaciones si es antisimétrica o no.

Caraterización de la especialización entre Servicios

Una relación de especialización es una relación taxonómica entre un elemento mas general y uno mas especifico. En PervML se utilizan relaciones de especialización para describir servicios que son una extensión de otros servicios más genéricos. En [3] se caracterizan las relaciones de especialización mediante una serie de características. A continuación se utilizan ese marco conceptual para definir el significado de una especialización en PervML.

Interpretación semántica: En nuestro caso la semántica de la relación taxonómica es de especialización según [3], ya que se entiende como la construcción de nuevos servicios añadiendo propiedades y operaciones.

Relación entre propiedades/operaciones: Esta propiedad define como se heredan los elementos y cual es la visibilidad entre ellos. En PervML se pueden dar dos tipos de relaciones entre elementos (propiedades y operaciones) de un servicio especificó y del servicio general: elemento heredado o elemento nuevo. No se permitirá la modificación o cancelación de elementos.

Restricciones estáticas: Define las restricciones entre la población de las subclases y la de la superclase que participan en la relación. Todas las relaciones de especialización en PervML son no completas y disjuntas. Es decir, por una parte pueden existir componentes que proporcionen la clase mas general y, por otra parte, un componente no podrá proporcionar dos servicios especializados.

Restricciones dinámicas: Especifica si los individuos de las poblaciones pueden cambiar de tipo durante el tiempo de vida del sistema. Todas las relaciones de especialización en PervML son estáticas; es decir, los componentes siempre realizaran el mismo servicio durante el funcionamiento del sistema.

Dependencia Existencial: Determina si los participantes de la relación dependen uno de la existencia del otro y viceversa. En PervML sólo hay dependencia existencial del servicio mas especializado respecto al servicio mas general, ya que se trata de especializaciones (que no tiene dependencia de la superclase a la subclase).

Identidad: Esta propiedad estudia las relaciones existentes entre la identidad de los objetos de la superclase y la de los objetos de la subclase. Al tratarse de especializaciones, esta propiedad no tiene sentido ya que cada tipo de servicio tendrá un mecanismo de identificación propio.

Criterios de especialización: Estudia algunos principios básicos que guíen el proceso de clasificación y el modelado de relaciones de especialización, determinando las propiedades y los tipos de criterios de especialización en los que se basa el particionamiento de una clase en un conjunto de subclases. El criterio de selección de un servicio especializado son los requisitos funcionales del sistema a desarrollar. Se elegirá crear un componente que realice un servicio u otro dependiendo de las necesidades del sistema. Por lo tanto, no se utilizaran criterios estáticos ni dinámicos.

3.4 Diagrama de Transición de Estados

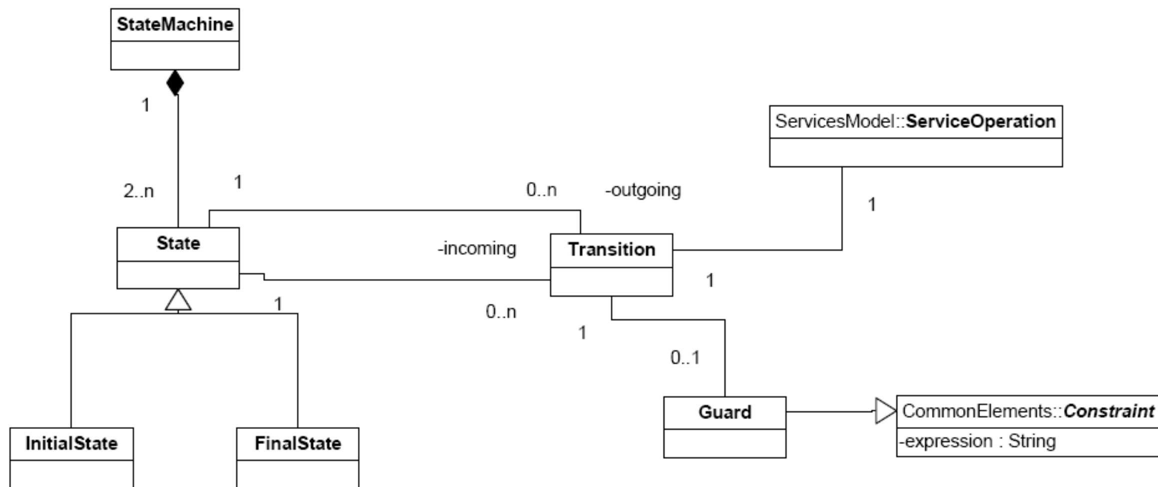


Figura 10: Contenido del paquete 'STD'

El paquete StateTransitionDiagram, mostrado en la figura 10, contiene los elementos para la descripción de Diagramas de Transición de Estados que caracterizan el comportamiento de los servicios. Un diagrama de transición de estados (StateMachine) se compone de un conjunto de estados(State), uno de los cuales debe ser inicial (InitialState) y al menos otro debe ser final (FinalState). Existen transiciones (Transition) entre estados, por lo que todo estado se relacionara con un conjunto de transiciones salientes y transiciones entrantes. En el caso del estado inicial el conjunto de transiciones entrantes será vacío, mientras que en los estados finales el conjunto de las transiciones salientes será vacío. Las transiciones están asociadas con una operación del servicio con el que están relacionados. La invocación de esta operación será la realización de la transición y, por tanto, el cambio de estado. Finalmente, existen guardas (Guard) que deben satisfacerse antes de producirse la transición. Si la condición de guarda no se cumple ni la transición ni la operación invocada se realizan.

3.5 Modelo Estructural

El paquete StructuralModel, mostrado en la figura 11, contiene los elementos para la descripción del Modelo Estructural de PervML. La estructura del sistema esta compuesta por un conjunto de componentes (Component), cada uno de los cuales proporciona uno de los tipos de servicios definidos en el Modelo de Servicios. Se debe indicar la ubicación física (Location) de cada uno de los componentes del sistema. También se contempla
la posibilidad de definir disparadores a nivel de componente (Component-Trigger), que son añadidos a aquellos definidos a nivel de servicio.

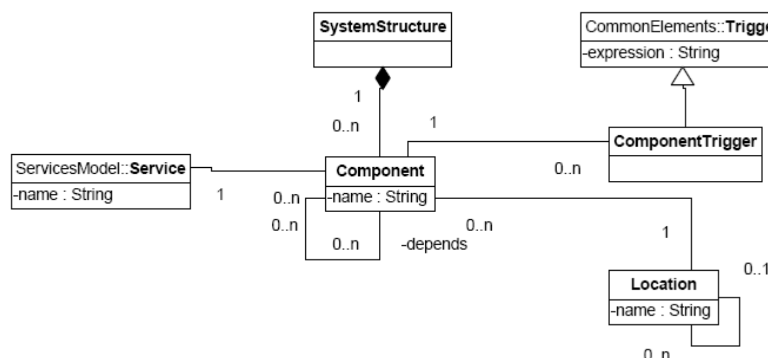


Figura 11: Contenido del paquete 'StructuralModel'

3.6 Modelo de Interacción

El paquete InteractionModel, mostrado en la figura 12, contiene los elementos para la descripción del Modelo de Interacción de PervML. En general, se puede decir que este paquete modela una versión de un diagrama de secuencia, donde los objetos que participan en la interacción son componentes del Modelo Estructural de PervML. Concretamente, una interacción (Interaction) está compuesta de una conjunto de elementos de interacción (InteractionElement) numerados en orden mediante su atributo
order. Los elementos de la interacción pueden tener restricciones para su realización (Condition) o condiciones que mientras se evalúan positivamente implican la repetición del elemento (RepeatCondition). Los elementos de la interacción pueden ser mensajes atómicos entre componentes (Message) o agrupaciones de otros elementos (InteractionBlock). Los mensajes tienen un componente origen y un componente destino, y debe indicarse cual es la operación que se invoca en el componente destino. Además, se deberán proporcionar argumentos (Parameter) para los parámetros de dicha operación, cuyo contenido est´a determinado por expresiones que hacen uso de operacio-nes de los componentes u operadores. Finalmente, el inicio de la interacción está determinado por la invocación en alguna operación mediante su nombre o por la satisfacción de un disparador (InteractionTrigeer).

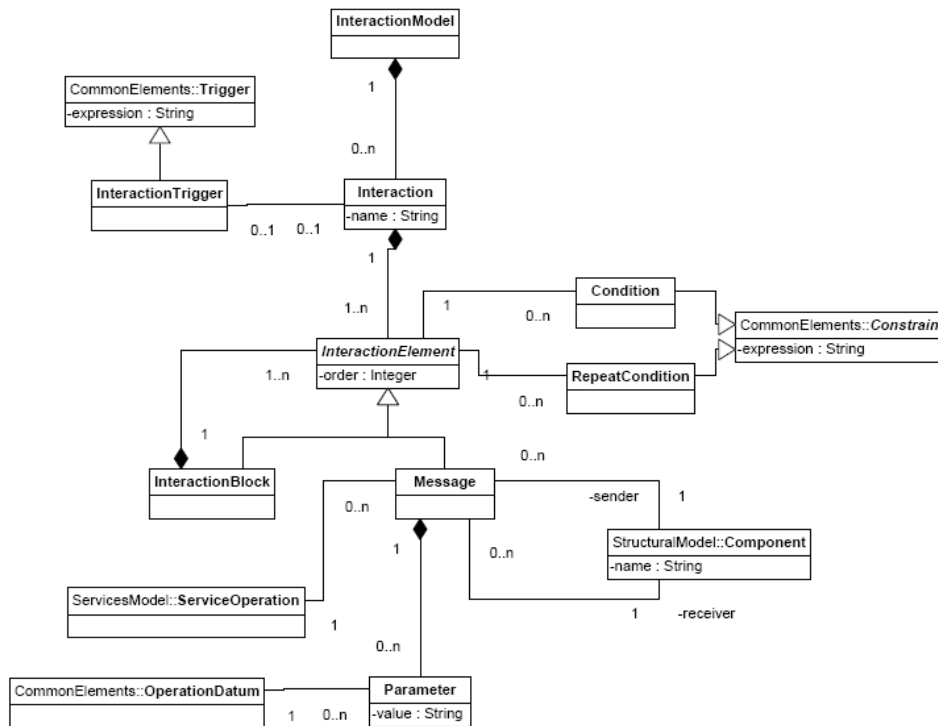


Figura 12: Contenido del paquete 'InteractionModel'

3.7 Modelo de Proveedores de Enlace

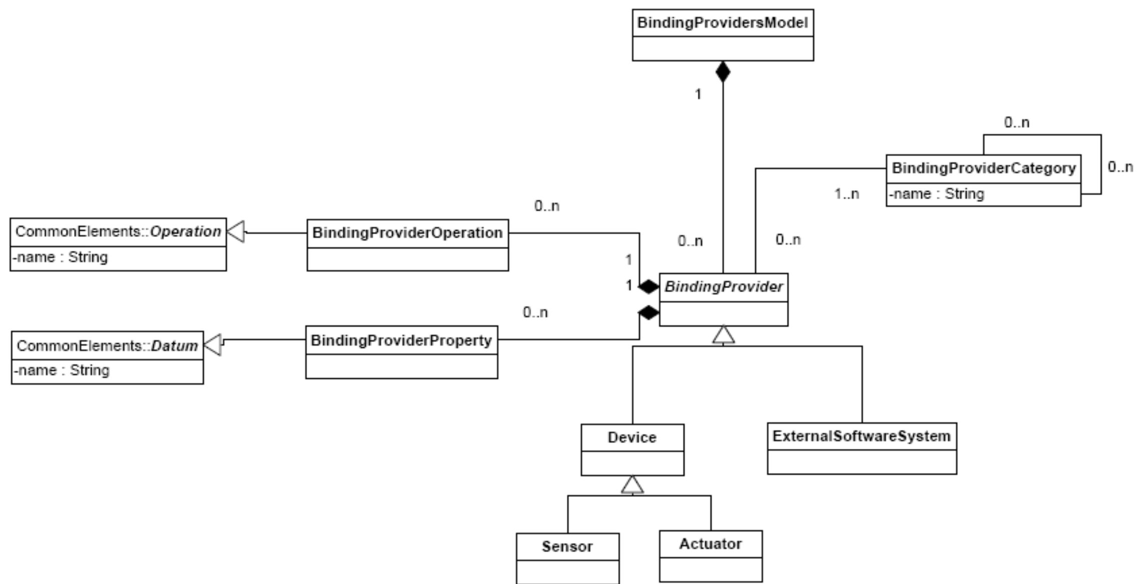


Figura 13: Contenido del paquete 'BindingProvidersModel'

El paquete BindingProvidersModel, mostrado en la figura 13, contiene los elementos para la descripción del Modelo de Proveedores de Enlace de PervML. Este paquete presenta bastantes similitudes con el paquete ServicesModel. El Modelo de Proveedores de Enlace se compone de proveedores de enlace (BindingProvider), los cuales se especializan en dispositivos (Device), sensores (Sensor), actuadores (Actuator) o sistemas software externos (ExternalSoftwareSystem). Estos elementos se componen de operaciones (BindingProvideroOperation) y propiedades (BindingProviderProperty),que reutilizan las descripciones realizadas en el paquete CommonElements.

La principal diferencia entre los distintos tipos de proveedores de enlace es que los sensores sólo tienen propiedades, que almacenan valores de las magnitudes físicas que perciben, y los actuadores sólo tienen operaciones, que realizan acciones sobre el entorno físico del sistema. Por otra parte, todos los tipos de proveedores de enlace deben encuadrarse en algunas de las categorías (BindingProviderCategory) que sean definidas.

A diferencia de lo que ocurre en el modelo de servicios, no se especifican ni diagramas de transición de estados, ni disparadores, ni restricciones de integridad en los proveedores de enlace. Únicamente se especifica su interfaz, ya que esta es habitualmente la única información que proporcionan los fabricantes de dispositivos para su manipulación.

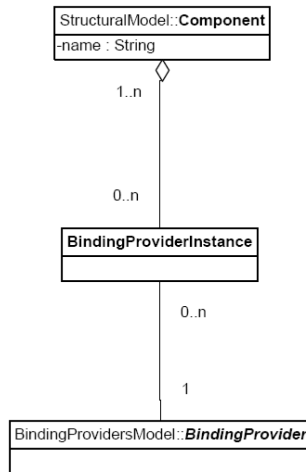


Figura 14: Contenido del paquete 'ComponentStructureSpecification'

3.8 Especificación Estructural de Componentes

Como puede observarse en la figura 14, la especificación de la estructura de un componente consiste en la identificación de los proveedores de enlace (BindingProviderInstance) que lo componen. Los proveedores de enlace han de ser de uno de los tipos definidos en el Modelo de Proveedores de Enlace. Es importante hacer notar que un mismo proveedor de enlace puede ser utilizado por varios componentes y, por lo tanto, puede colaborar para proporcionar varios servicios.

3.9 Especificación Funcional de Componentes

El paquete ComponentFunctionalSpecification, mostrado en la figura 15, define un elemento nuevo que contiene el cuerpo del método en el que se especifican las acciones que realizara un componente cuando es invocada una de las operaciones del servicio que proporciona. Las acciones se especificaran utilizan el Action Semantics Language (ASL) de UML.

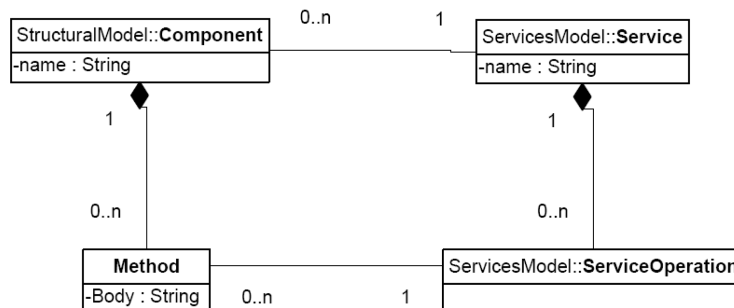


Figura 15: Contenido del paquete 'ComponentFunctionalSpecification'

4 Descripción del framework

El desarrollo de un sistema pervasivo supone el uso de diversas tecnologías para satisfacer los requisitos de los usuarios. Habitualmente estas tecnologías proporcionan a los desarrolladores constructores de bajo nivel de abstracción. MDA y las Factorías de Software [5] proporcionan estrategias para aumentar el nivel de abstracción y la productividad en el desarrollo de sistemas complejos.

Para salvar el salto de abstracción entre los lenguajes de modelado y las tecnologías de implementación, el enfoque de las Factorías de Software propone la construcción de frameworks, tal y como se muestra en la Figura 7 (A). Aplicando principios de la ingeniería de dominios, se desarrolla un framework que aumenta el nivel de abstracción de la tecnología. De esta manera se reduce la cantidad de código que debe ser generada a partir de los modelos de alto nivel de abstracción.

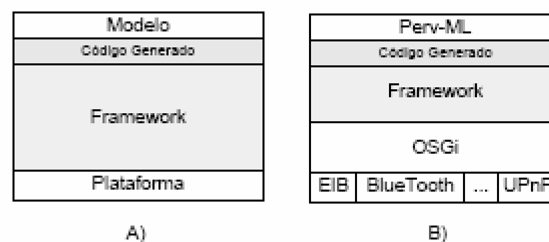


Figura 16: Estrategia propuesta por las Factorías Software y nuestra aplicación.

Se describe a continuación el framework que da soporte al lenguaje de modelado PervML. La implementación de este framework se ha desarrollado de forma cooperativa con el proyecto fin de carrera "Desarrollo de un Sistema de Gestión del Hogar Digital aplicando un enfoque Dirigido por Modelos" con código II-B-DSIC-110/05, ya que su Proyecto Final de Carrera también se ha realizado en un contexto similar. El framework, implementado utilizando el middleware OSGi, proporciona una serie de clases abstractas que deberán ser convenientemente extendidas para obtener una aplicación funcionalmente operativa. Las primitivas que proporciona el framework son similares a las que utiliza en el lenguaje de modelado.

El framework para sistemas pervasivos ha sido diseñado con la intención de servir de soporte a los sistemas especificados utilizando el lenguaje de modelado PervML. Por lo tanto, un objetivo crítico ha sido proporcionar primitivas similares a aquellas utilizadas por el lenguaje. Este requisito ha hecho que la arquitectura de los sistemas generados por el framework siga una estructura y estrategia de ejecución análoga a la propuesta por el lenguaje de modelado para especificar los sistemas. A continuación se describe tanto la estructura como la estrategia global de ejecución, y por último el uso del framework.

4.1 Estructura del Framework

La estructura de los sistemas desarrollados con el framework para sistemas pervasivos sigue los patrones arquitectónicos Layers y Model-View-Controller. Mediante el patrón Layers, organizamos los elementos del sistema en capas con unas responsabilidades bien definidas. Por otra parte, mediante el patrón Model-View-Controller proporcionamos soporte a la existencia de varias interfaces de usuario para interactuar con el sistema.

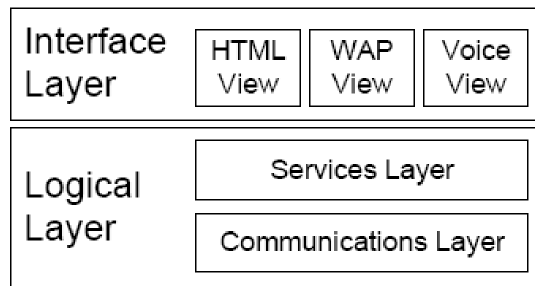


Figura 17: Estructura del framework

El framework se puede estructurar en dos partes bien diferenciadas como se muestra en la figura 8: el soporte a los elementos de la Capa Lógica (proveedores de enlace, componente e interacciones) y el soporte a las Interfaces de Usuario que se encuentra en la Capa de Interfaz.

Capa Lógica

La capa lógica se puede dividir en dos capas: la capa de Comunicaciones y la capa de Servicios.

La Capa de Comunicaciones proporciona una representación de los proveedores de enlace (binding providers) que son usados en capas superiores. Esta capa contiene los aspectos de la comunicación con los dispositivos y sistemas software que son independientes de tecnología y fabricante. Por ejemplo, se encarga de registrar los accesos al proveedor de enlace, de notificar los cambios en el driver a todos los elementos de capas superiores que requieran esa información, etc. Por todo ello, existe una relación uno a uno entre los drivers y los elementos de esta capa. Por ejemplo, si el sistema contiene un sensor de temperatura de la tecnología LonWorks, existirá un driver para tratar con las cuestiones específicas de esa tecnología y un proveedor de enlace con funcionalidad común a todos los sensores de temperatura.

La Capa de Servicios se encarga de abstraer la funcionalidad que implementan las capas inferiores para ofrecerla tal y como la esperan los usuarios del sistema. Cada uno de los elementos de esta capa, a los que llamamos componentes, proporciona un tipo de servicio. Para hacer uso de la funcionalidad proporcionada por un tipo de servicio se establece un contrato (definido mediante pre y post condiciones de las operaciones) y un protocolo, que establece las operaciones que pueden invocarse en un momento dado. Para implementar su funcionalidad, un componente puede hacer uso de proveedores de enlace (dispositivos y sistemas software externos) o de otros componentes.

Finalmente, en esta capa también se pueden implementar interacciones entre componentes. En este ámbito entendemos una interacción como una secuencia de comunicaciones entre componentes, conceptualmente no relacionados a priori, para proporcionar una funcionalidad requerida por el usuario. Por ejemplo, el usuario puede requerir que se atenúe la iluminación de una sala al iniciarse la reproducción de un objeto multimedia. Conceptualmente, los servicios de iluminación y de reproducción de objetos multimedia no están relacionados entre sí pero, en un sistema en concreto, el usuario puede desear establecer una interacción entre ellos.

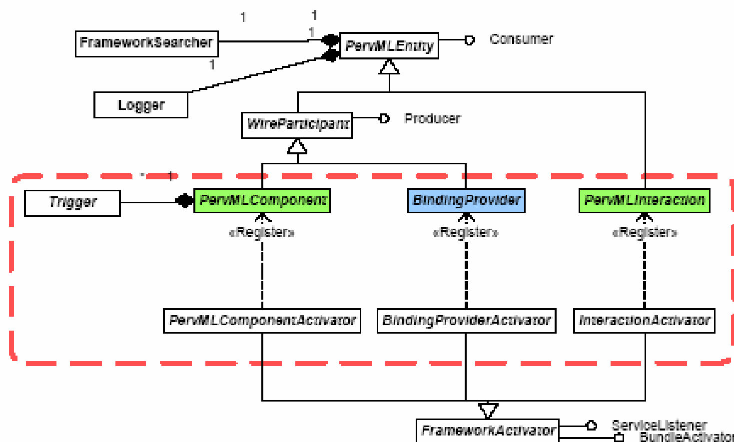


Figura 18: Estructura de clases para soportar las capas de Comunicaciones y Servicios

La Figura 9 muestra la estructura de clases que proporcionan soporte a la Capa Lógica. Los elementos dentro del recuadro punteado serán los utilizados finalmente para extender del framework y generar el sistema final.

Las clases PervMLEntity y WireParticipant se encargan de implementar las interfaces que permiten utilizar el mecanismos de Wire de OSGi, así como de facilitar acceso a objetos de las clases FrameworkSearcher y Logger. El resto de clases de esa jerarquía (en el recuadro) contienen los puntos de extensión del framework. En ellas se definen los atributos que deben inicializarse al instanciar el framework, y se implementan, utilizando el patrón de diseño Template Method, las estrategias de ejecución de cada uno de los elementos.

La otra jerarquía de clases mostrada en la figura contiene los elementos de soporte a la creación de los activadores. FrameworkActivator es la clase más abstracta y se encarga de definir la estrategia general de los activadores y los puntos de extensión. También implementa algunas operaciones que se encargan de realizar funcionalidad atómica, como registrar en OSGi un objeto bajo una interfaz o crear un Wire entre dos servicios. El resto de activadores más específicos implementan algunos de los puntos de extensión, dejando otros abstractos para que sean escritos al instanciar el framework.

Capa de interfaz

La Capa de Interfaz se encarga de proporcionar el acceso al sistema por parte de cualquier tipo de usuario (tanto personas como otros sistemas software). En un contexto pervasivo es habitual proporcionar distintos tipos de interfaces para acceder al mismo sistema. Por ejemplo, una interfaz web para acceder remotamente, una PDA o un teléfono móvil para acceder con movilidad, o una interfaz a través de TV para gestionar de manera centralizada un hogar. Por ello, se ha considerado adecuado aplicar el patrón Model-View-Controller en el framework. Siguiendo este patrón, sobre la capa de servicios se sitúa un elemento controlador. Las distintas interfaces (vistas) del sistema se comunican con este controlador para interactuar con el sistema.

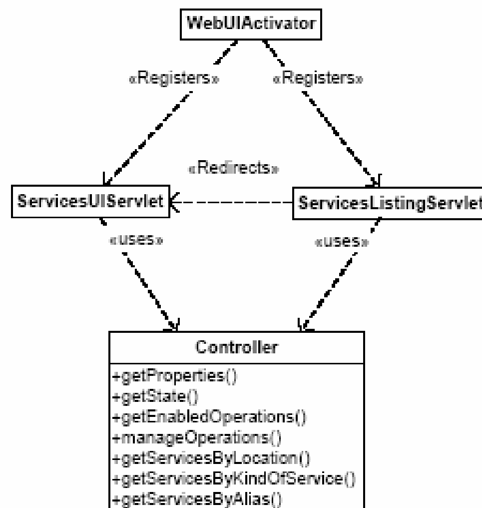


Figura 19: Estructura de clases para soportar la capa de Interfaz

La Figura 19 muestra el diagrama de clases que proporcionan soporte a la Capa de la Interfaz. Como ya se ha comentado al describir la arquitectura, se ha aplicado el patrón MVC [6]. En nuestros sistemas, los elementos que forman parte del Modelo son los Component; mientras que la capa de interfaz contiene el Controlador y las distintas Vistas.

Al realizar el diseño del Controlador se ha considerado que el usuario realizará dos tareas a la hora de interactuar con el sistema:

1. Seleccionar el servicio con el que desea interactuar de entre todos los disponibles en el sistema. Para ello será necesario proporcionar mecanismos de estructuración, acceso y búsqueda de los componentes.

2. Interactuar con un servicio en concreto. En esta interacción el usuario recibirá información del servicio y podrá solicitar la ejecución de alguna funcionalidad que este ofrece.

Siguiendo esta filosofía, el controlador tiene dos grupos de métodos para soportar estas tareas:

- El primero de los grupos está formado por los métodos getServicesByLocation, getServicesByKindOfService y getServicesByAlias. Estos métodos devuelven una lista con información sobre los servicios del sistema, filtrándolos por distintos criterios.

- El segundo de los grupos está formado por los métodos getProperties, getState, getEnabledOperations y manageOperation. Las tres primeras operaciones son invocadas a la hora de crear la interfaz de usuario, ya que devuelven información de un componente específico. En concreto, la primera de ellas devuelve las propiedades invariables de un servicio (el tipo de servicio, su ubicación, etc.), la segunda devuelve el estado del servicio (por ejemplo, si se encuentra encendido o apagado, o si está detectando presencia) y, finalmente, la tercera devuelve las operaciones que el componente puede ejecutar en ese instante. Por último, manageOperation se encarga de invocar una operación en un Component utilizando las capacidades de reflexión de Java.

En la Figura 10 también se muestran las clases que proporcionan la vista a través de páginas web. Se trata de dos servlets de Java, cada uno de los cuales se utiliza para soportar una de las dos tareas descritas anteriormente. Los servlets realizan invocaciones a las operaciones del controlador para generar las páginas web que serán mostradas a los usuarios.

Es importante hacer notar que todas las clases de esta capa son concretas; es decir, no necesitan extenderse ya que son válidas para todos los sistemas. Esto se ha conseguido gracias a que (1) todos los componentes implementan una interfaz que puede ser utilizada por el controlador y (2) se han utilizado las capacidades de reflexión de Java a la hora de ejecutar las operaciones específicas de los componentes.

4.2 Estrategia Global de Ejecución

Una vez presentada la estructura de los sistemas desarrollados utilizando el framework, es necesario describir su estrategia de ejecución; es decir, las reglas que definen las secuencias de acciones que suceden en el sistema cuando este se encuentra en funcionamiento.

Secuencia de acciones

Existen dos posibles modos de iniciar una secuencia de acciones:

1. Un *Driver* notifica a un *BindingProvider* de un cambio en el entorno.

2. El usuario solicita, utilizando una UI, la ejecución de una funcionalidad de un servicio.

A continuación se describen los pasos que se llevan a cabo al realizar estas dos secuencias de acciones; así como la estrategia de ejecución de las operaciones de los servicios y de las interacciones. Finalmente, se describe brevemente el funcionamiento de la interfaz web que se ha desarrollado.

Cuando un *Driver* notifica a un *BindingProvider* que ha habido un cambio en el entorno, el *BindingProvider* notificará este hecho a los *Component* que hacen uso de él. Estos, a su vez, evaluarán sus disparadores (*Triggers*) y notificarán a aquellos otros *Component* o *Interaction* en cuyas condiciones de disparo participa.

Detalladamente, y como indica gráficamente la figura 11, se realizan los siguientes pasos:

1. El *Driver* notifica al *BindingProvider* que ha sucedido un cambio en el entorno. No indica de qué cambio se trata.

2. El *BindingProvider* notifica a todos los elementos *Component* que lo utilizan de un cambio en el entorno. No indica de qué cambio se trata.

3. El *Component* evalúa las condiciones de sus disparadores. Para ello podrá realizar invocaciones sobre las operaciones de los *BindingProviders* que utiliza (entre ellos quizá el que inició las acciones) o de otros *Component* con los que se relaciona.

4. El Component consulta los resultados de aquellas de sus operaciones que devuelven algún valor (aquellas operaciones como “getIntensity(): int” o “isLighting(): boolean”).

    a. Si ninguno de los valores devueltos ha cambiado desde la última vez que los consultamos:

        i. Finaliza la secuencia de acciones

b. Si algún valor ha cambiado:

i. Almacenamos los nuevos valores.

ii. Continuamos con los siguientes pasos

5. El Component notifica de que ha habido un cambio en el resultado de alguna de sus operaciones a aquellos Component e Interaction que le escuchan.

6. Los elementos notificados (*Component* o *Interaction*) evalúan las condiciones de sus disparadores (el *Interaction* sólo tiene una condición), para lo cual invocarán alguna de las operaciones del *Component* que realizó la notificación.

7. Ocasionalmente, como resultado de los pasos 3 o 6, se inicia la ejecución de alguna operación de un *Component* o las acciones de una *Interaction*.

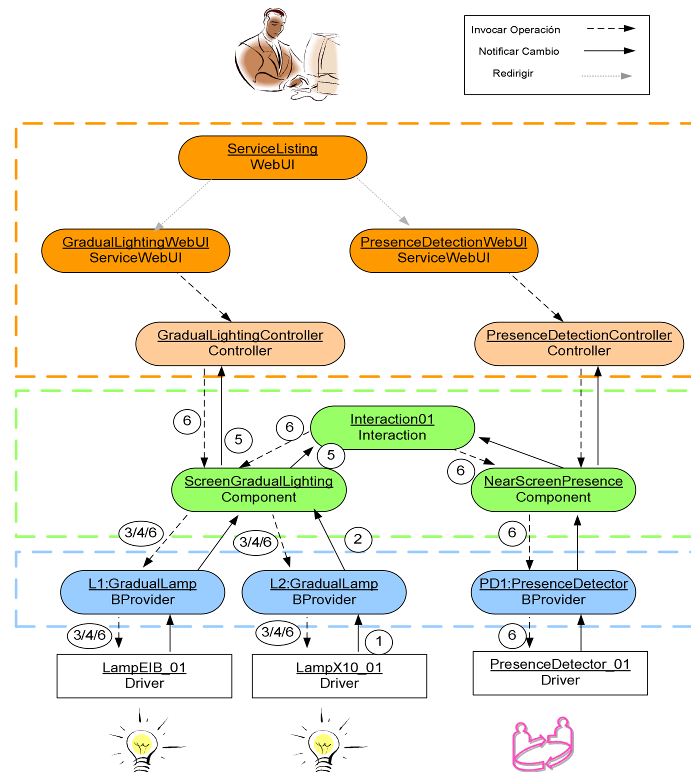


Figura 20: Pasos que se realizan ante un cambio en el entorno

Cuando un usuario desea que se lleve a cabo la funcionalidad proporcionada por un servicio, en primer lugar debe buscar en la UI la instancia del servicio (*Component*) con la que quiere interactuar. Entonces invoca la operación (pulsa un botón, pasa el puntero del ratón por una región, dice un comando, etc.) y la UI solicita a un *Controller* que dicha operación se lleve a cabo. El *Controller*, si es posible, invoca la operación sobre el *Component*.

Detalladamente, y como indica gráficamente la figura 12, se realizan los siguientes pasos:

1. El *Usuario* selecciona en la UI la instancia del servicio que quiere manejar. En el sistema web, esto se realiza mediante varias páginas que permiten ver los servicios ordenados por tipo o por ubicación; o mediante búsquedas por nombre.

2. La *UI* transmite al usuario información sobre el componente seleccionado, así como aquellas operaciones que puede ejecutar. Para ello se realizan los siguientes pasos:

a. La *UI* solicita la información al *Controller* del tipo de servicio con el que desea interactuar. Esta información está estructurada en tres paquetes:

i. Información general del *Component*: Actualmente se muestra el tipo de servicio de que se trata y su ubicación.

ii. Información sobre la situación actual del *Component*: Que viene determinada por los resultados de aquellas de sus operaciones que devuelven un valor y no reciben parámetros.

iii. Operaciones que puede invocar el usuario en ese instante.

b. El *Controller* interactúa con el *Component* para obtener la información necesaria, para lo cual:

i. Obtiene la información estática relativa al componente. Esta información consiste en localización, tipo y nombre.

ii. Invoca aquellas operaciones del *Component* que devuelven un valor y no reciben parámetros. Los resultados los devuelve empaquetados en un diccionario.

iii. Consulta al Component las operaciones que puede ejecutar en ese instante, y para cada una de las operaciones el nombre y el tipo de sus parámetros. El resultado lo devuelve en un diccionario en el que el campo clave es el nombre de la operación y el campo valor es otro diccionario, en el que la clave es el nombre del parámetro y el valor es el tipo del mismo.

c. La *UI* transmite al *Usuario* la información devuelta por el Controller. El modo en que se transmite esta información es dependiente del tipo de UI (página Web, PDA, secuencias de voz, imagen del entorno, etc).

3. El *Usuario* ordena la ejecución de una operación (pulsa un botón, pasa el puntero del ratón por una región, dice un comando, etc.). Si es necesario, la *UI* se encarga de recoger los parámetros para iniciar la petición de ejecución.

4. La *UI* transmite la petición de ejecución al *Controller*, indicándole el *Component* sobre el que quiere invocar la operación, el nombre de la operación, y los parámetros de esta, si los necesita.

5. El Controller invoca la operación en el Component.

6. El Controller vuelve a interactuar con el componente para obtener la información descrita en el punto 2 b

a. Si la operación se ha ejecutado de forma correcta se devuelve la información que refleja el estado actual a la UI.

b. Si ha habido algún problema durante la ejecución de la operación, junto a la información que refleja el estado actual del componente, el controller devuelve a la UI un diccionario que contiene los errores sucedidos y el instante en el que se dieron.

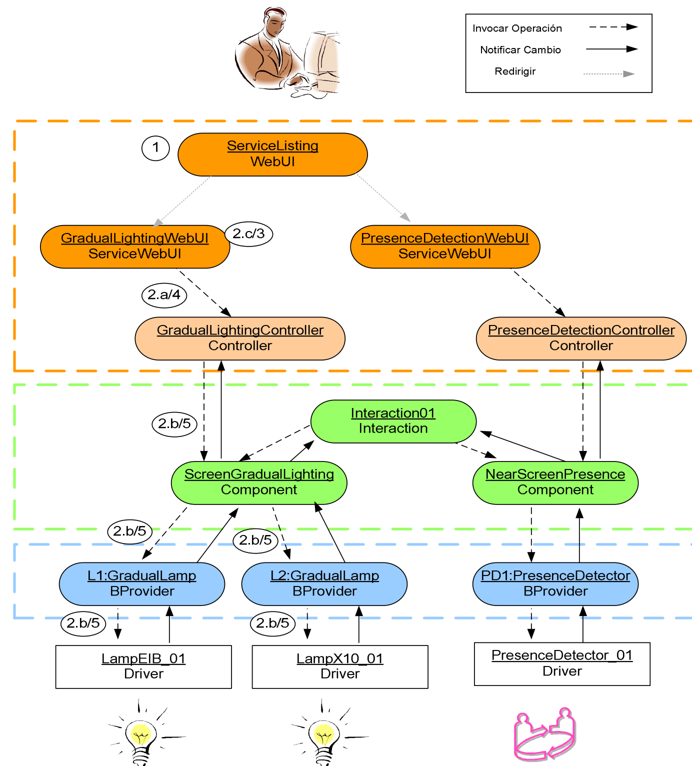


Figura 21: Pasos ante la solicitud de un servicio por parte del usuario

Ejecución de operaciones

La estrategia de ejecución de operaciones es una de las partes más importante de la estrategia de ejecución general. Para ejecutar una operación de un *Component* se realizan los siguientes pasos:

1. Comprobar que es posible ejecutar la operación. Para ello será necesario:

   a. Comprobar que, desde el estado actual del DTE, está permitido llevar a cabo la operación. A su vez, este paso se puede descomponer en:

      i. Comprobar que realmente el *Component* se encuentra en el estado que está almacenado. Para ello se evaluará la fórmula asociada a ese estado.

         1. Si no se cumple la fórmula:

            a. Evaluar las fórmulas de todos los estados del DTE

               i. Si sólo se cumple una: se actualiza el estado actual

               ii. Si se cumplen varias: se elige un estado al azar

               iii. Si no se cumple ninguna: Registrar el error y se detiene la ejecución

      ii. Comprobar que la operación se encuentra en la lista de operaciones válidas en el estado actual.

1. Si no es encuentra en la lista: se registra el error y se detiene la ejecución.

b. Comprobar la precondición. Para ello se evaluará la fórmula asociada.

i. Si la fórmula *no* se satisface: se registra el error y se detiene la ejecución.

2. Deshabilitar la recepción de notificaciones. De este modo la ejecución de las operaciones tiene un carácter atómico. Esto se realiza para evitar comportamiento no deseado y posibles ciclos como consecuencia de la invocación de operaciones en los *BindingProviders*, los cuales a su vez podrían enviar notificaciones al propio *Component*. (Por ejemplo, un servicio de Iluminación que debe activar varias bombillas. Al activar una bombilla, esta notificará al *Component* de un cambio en su estado; lo cual desencadenará una serie de acciones en él (comprobación de disparadores, notificación del *Component* a otros elementos, etc.) antes de que se hayan activado el resto de bombillas).

3. Realizar las acciones de la operación. Para ello se sigue la siguiente estrategia:

a. Conseguir las referencias de los *Component* o *BindingProviders* que se utilicen en los siguientes pasos.

i. Si no se consiguen todas las referencias: se registra el error y se detiene la ejecución

b. Ejecutar las acciones que implementan la operación.

i. Si hay alguna excepción: se registra el error y se detiene la ejecución

ii. Si hay que devolver un valor, se almacena en la variable *returnValue*.

c. Si hay que devolver un valor: hacer *return* de la variable *returnValue*.

4. Activar la recepción de notificaciones.

5. Comprobar que si la operación ha tenido el efecto esperado. Para ello se evaluará la fórmula de la postcondición.

a. Si la fórmula se satisface:

i. Transitar en el DTE

b. Si la fórmula *no* se satisface:

i. Registrar el error.

ii. Averiguar el estado del DTE en el que se encuentra el *Component*.

1. Evaluar las fórmulas de todos los estados del DTE

a. Si sólo se cumple una: se actualiza el estado actual

b. Si se cumplen varias: se elige un estado al azar

c. Si no se cumple ninguna: Registrar el error y se detiene la ejecución

6. Evaluar los disparadores del *Component*, por si se ha perdido alguna notificación durante el tiempo que se deshabilitaron las notificaciones.

7. Para todas las operaciones que no definen el estado, el Component consulta los resultados de aquellas de sus operaciones que devuelven algún valor (aquellas operaciones como “*getIntensity(): int*” o “*isLighting(): bolean*”).

a. Si algún valor ha cambiado:

i. Almacenamos los nuevos valores.

ii. Notificamos del cambio a los *Component* o *Interaction* suscritos.

4.3 Instanciación del Framework

En esta sección se describe cómo instanciar el framework para implementar un sistema persvasivo. En cada subsección se proporciona una breve visión para cada una de los elementos del sistema.

Creación de los Proveedores de Enlace

Como puede observar en la Figura 13, tanto los drivers, como los proveedores de enlace con los que se relacionan, implementan una misma interfaz. Por ejemplo, los dos drivers para acceder a lámparas que proporcionan iluminación gradual y su proveedor de enlace implementan la interfaz GradualLamp.

Sólo existe un proveedor de enlace por cada tipo. Será en el activador donde se creen dos instancias, las cuales recibirán como parámetros de su constructor su propio identificador y el identificador del driver con el que están emparejados. A continuación se muestra el código contenido en el punto de extensión del activador (método specificStart), donde los identificadores de los drivers son LampEIB_01 y LampX10_01.

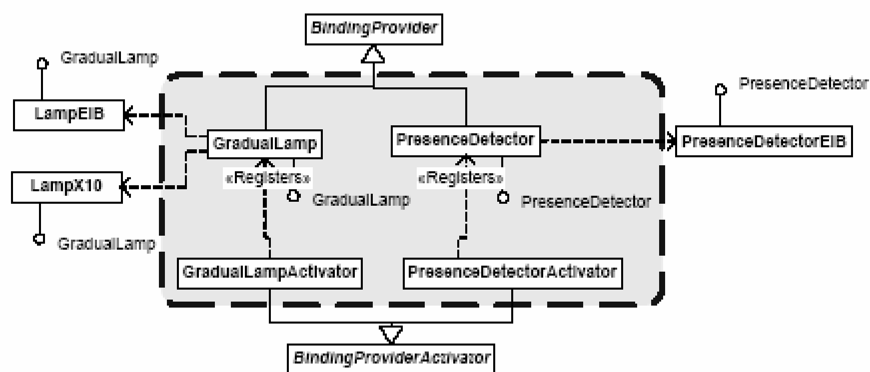


Figura 22: Estructura de clases para implementar los proveedores de enlace

```
public void specificStart(){
registerBProviders("LampEIB_01","L1",new BProvider("L1",this.context));
registerBProviders("LampX10_01","L2",new BProvider("L2",this.context));
}
```

La implementación de las operaciones del proveedor de enlace se limita a realizar la invocación en el driver. Para ello, se hace uso de un método llamado runMethod que realiza la invocación utilizando las capacidades de reflexión de Java. A continuación se muestra el código de la operación On() del proveedor de enlace que representa una lámpara gradual.

```
public void On(){
runMethod("On",new Object[0]);
}
```

Creación de los Componentes e Interacciones

La Figura 14 muestra la estructura de clases para implementar los componentes del sistema. Como puede observarse cada componente está implementado en dos clases (por ejemplo, GradualLighting y ScreenGradualLighting), atendiendo al siguiente criterio:

- La clase más abstracta (GradualLighting en el ejemplo) implementa aquella funcionalidad que es común para todos los componentes que proporcionan un mismo servicio. Esto incluye métodos para calcular las operaciones que pueden ser invocadas en un momento dado, las pre y post condiciones de las operaciones y la estrategia general de ejecución de las operaciones del servicio.

- La clase más específica (ScreenGradualLighting en el ejemplo) implementa las acciones de las operaciones. Hay que tener en cuenta que en el sistema podría existir otro componente que proporcionase el servicio de iluminación gradual, el cual utilizase otro tipo de dispositivos (por ejemplo, lámparas que permiten indicar un porcentaje de luminosidad o lámparas que sólo permiten 4 posiciones de intensidad) y, en ese caso, las acciones a llevar cabo cuando se ejecutasen las operaciones serían necesariamente distintas aunque el servicio proporcionado fuese el mismo.

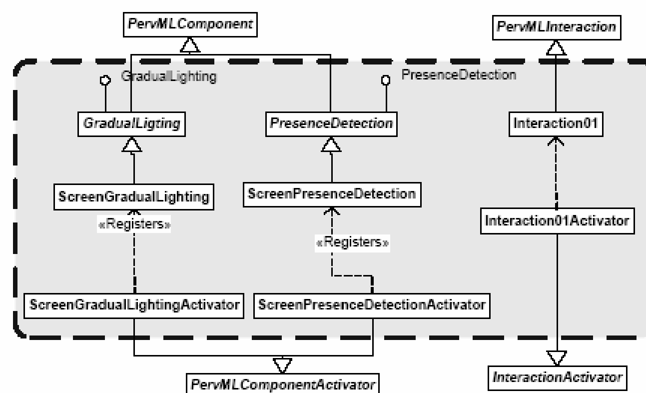


Figura 23: Estructura de clases para implementar los componentes y las interacciones

A continuación se muestra la implementación del método para fijar la intensidad del componente que proporciona el servicio GradualLighting. La estrategia que se sigue consiste en dividir la intensidad entre las dos lámparas, de manera que el primer 50% lo proporciona la lámpara L1 y el segundo 50% L2.

```
public void setIntensityImplementation(int intensity){
    org.oomethod.interfacesBProviders.gradualLamp.Interface L1, L2;
    L1 =(org.oomethod.interfacesBProviders.gradualLamp.Interface)
            frameworkSearcher.getBProvider("L1");
    L2 =(org.oomethod.interfacesBProviders.gradualLamp.Interface)
            frameworkSearcher.getBProvider("L2");
    if ( intensity >= 50) {
            L1.on();
            L2.set( (intensity - 50)*2 );
    } else {
            L1.set( intensity *2 );
            L2.off();
            }
    }
```

Por otra parte, el activador del componente es el encargado de inicializar alguna variables específicas y de crear los canales de comunicación (Wires) con los proveedores de enlace. Para ello implementa las operaciones que forman los puntos de extensión proporcionados por el framework.

```
package org.oomethod.components.screenGradualLighting;
public class Activator extends ComponentActivator {
    protected void initialize(){
            componentPID="ScreenGradualLighting";
            implLocation="/Dept/Floor1/Lab103";
            componentInstance=new Component(this.context,componentPID);
    }
    protected void createAllWires(WireAdmin wa){
            this.createWire(wa,"L1",componentPID);
            this.createWire(wa,"L2",componentPID);
    }
 }
```

En la Figura 23 también aparecen las clases para implementar la Interacción del ejemplo. Una interacción deberá implementar los puntos de extensión definidos por el framework:

- el método checkCondition evalúa la condición de inicio de la interacción.
- el método doAction realiza las invocaciones sobre los distintos componentes que participan en la interacción.

A modo de ejemplo, a continuación se muestra el código del método que evalua la condición de la interacción.

```
public boolean checkCondition(){
    ScreenPresenceDetection = frameworkSearcher.getComponent("NearScreenPresenceDetector");
        ScreenGradualLighting = frameworkSearcher.getComponent("ScreenGradualLighting");
    boolean returnValue;
    returnValue = ScreenPresenceDetection.presenceDetected() &&
        (! ScreenGradualLighting.isLighitng() );
        return returnValue;
    }
```

5 Gestión de modelos

5.1 Tareas necesarias para la gestión de modelos

Parte de la funcionalidad esencial de una herramienta CASE reside en la gestión de los modelos a los que da soporte. Intuitivamente se pueden detectar dos grandes tareas relacionadas con la gestión de modelos:

Creación, edición y eliminación de modelos.
Materialización/dematerialización de la representación de los modelos a los ficheros en los que se almacenan.

Es necesario remarcar que en todo momento los modelos deben de ser conformes al metamodelo del lenguaje PervML, para ello en todos los aspectos listados se deben proporcionar mecanismos que lo garanticen.

5.2 Tecnología utilizada

La plataforma Eclipse proporciona funcionalidad para realizar las tareas de gestión de modelos. EMF, Eclipse Modeling Framework, es un plugin para el entorno integrado de desarrollo Eclipse que extiende su funcionalidad, aportando un framework de modelado para la construcción de herramientas y otras aplicaciones basadas en modelos estructurados.
La elección de EMF como pieza fundamental para la gestión de modelos viene motivada por las siguientes características que presenta:

Trasforma modelos en código Java.
Proporciona una infraestructura para utilizar modelos de forma eficiente en una aplicación.
Es gratuito y open source.
Es una herramienta madura en uso desde el año 2002.

En EMF, el lenguaje para describir los metamodelos de los lenguajes de modelado se llama Ecore. Ecore puede verse como una implementación con ligeras diferencias de Essential MOF (EMOF), que es un subconjunto del lenguaje estándar de meta-modelado MOF 2.0. OMG en su notación de modelado MOF (Meta-Object Facility), ha definido un lenguaje para describir los elementos que constituyen los lenguajes de modelado. MOF puede considerarse como un lenguaje para describir lenguajes de modelado, como pueden ser UML, CWM (Common Warehouse Metamodel), o incluso el propio MOF.

Existen pequeñas diferencias entre EMOF y Ecore, principalmente relativas a diferencias de nombrado, pese a las cuales EMF soporta lecturas y escrituras transparente de serializaciones de modelos EMOF y la conversión de modelos EMOF a Ecore, permitiendo un intercambio de datos estándar entre herramientas.

En la figura 24 se muestra la jerarquía de clases completa del meta-metadomelo Ecore. Se puede observar que la clase EPackage contiene información sobre las clases de modelo (EClass) y los tipos de datos (EDataType). EClass representa una clase de modelado , y espcifica los atributos y referencias representantes de los datos de las instancias. EAttribute representa datos simples, especificados por un EDataType y EReference representa una asociación entre clases; su tipo es una EClass. EFactory contiene métodos para crear elementos del modelo.

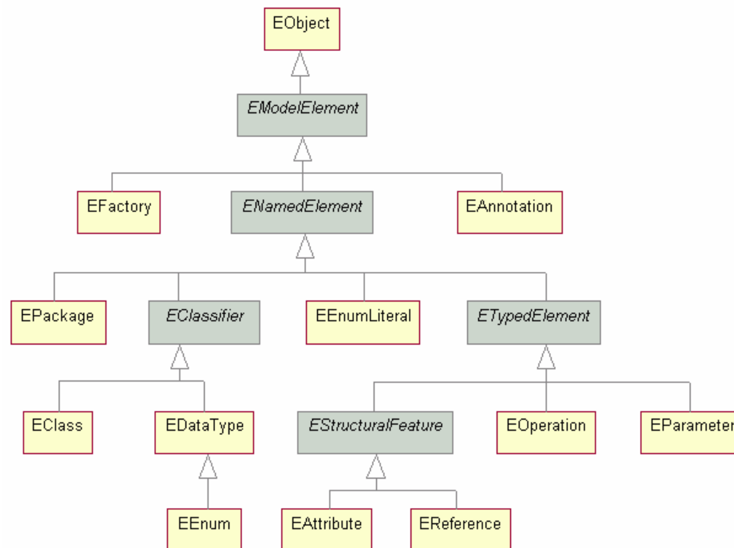


Figura 24. Jerarquia de clases de ecore.

Una de las interfaces claves en Ecore es EObject, la cual es conceptualmente equivalente a java.lang.Object. Todos los objetos modelados implementan esta interfaz para proporcionar varias características importantes:

De forma similar a Object.getClass(), utilizando el metodo eClass() se puede obtener los metadatos de la instancia.
En un objeto modelado ecore se pueden utilizar los métodos reflexivos del API (eGet(), eSet()) para acceder a sus datos. Esto es conceptualmente equivalente al método de java java.lang.reflect.Method.invoke().
De cualquier instancia de objeto se puede obtener su contenedor (padre) utilizando el metodoeContainer().
EObject extiende Notifier, lo cual permite monitorizar todos los cambios en los datos de los objetos.

Ademas existe otro punto importante a resaltar; la interfaz EObject extiende la interfaz Notifier.

La interfaz Notifier introduce una característica muy importante a cada elemento del modelo; notificaciones de cambios en el modelo como en el patron de diseño Observador [7].

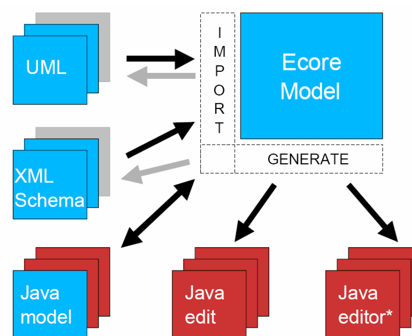


Figura 25: EMF: importación de modelos y generación.

Un modelo Ecore puede ser creado siguiendo dos aproximaciones:

1. Importándolo de un modelo en uno de los siguientes formatos:
    a. Diagrama de Rational case.
    b. Java anotado.
    c. XML.

2. Utilizando el editor grafico de modelos Ecore proporcionado por GMF.

A partir de un modelo Ecore, EMF es capaz de generar de una forma simple y eficiente:

- Clases Java que implementan el metamodelo.
- Clases para la edición independientes de la interfaz de usuario.
- Una vista de edición de modelos integradas en el IDE eclipse.
- Esqueletos para las clases de test de JUnit.

Además de lo anterior también se generan los elementos necesarios para encapsular cada uno de los puntos anteriores en plug-ins de eclipse:

- Manifiestos.
- Archivos properties.
- Iconos.
- Clases relacionadas con los plug-in.

5.3 Creación de un modelo ecore

Como se ha descrito en el punto anterior, a partir de un modelo Ecore es posible generar gran cantidad de código java que cubre algunos de los aspectos necesarios para la gestión de modelos.
En este punto se pretende describir el proceso de creación de un modelo ecore, utilizando como fuente un archivo .mdl de Rational Rose, que contiene el metamodelo del lenguaje PervML. Junto al modelo Ecore se creara un archivo genModel, que EMF utiliza almacenar información que no es propia de los modelos, pero que se utilizara para la generación del código java, como puede ser: nombres de los proyectos que se crearan, nombres de los paquetes java o rutas de los proyectos en el sistema de archivos.

I. Crear un nuevo proyecto EMF en el workspace. Para ello se debe mostrar el dialogo “File/New/Project”.

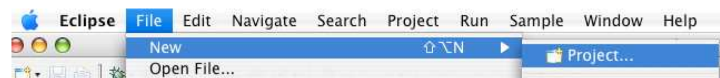


Figura 26: EMF: Creación de modelo Ecore: Nuevo proyecto.

II. Expandir la opción “Eclipse Modeling Framework” y seleccionar “EMF project”.

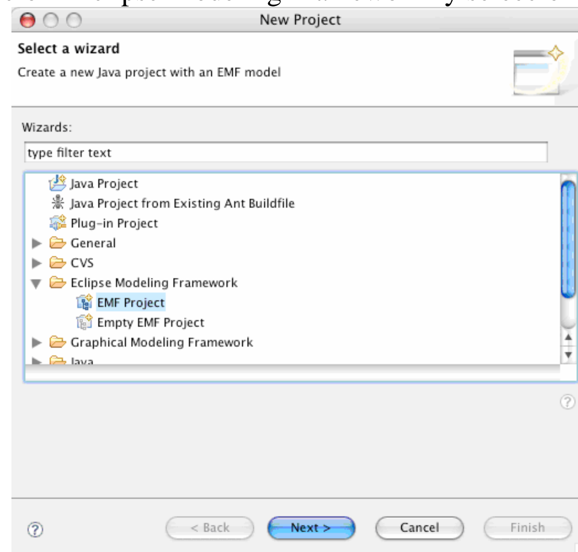


Figura 27: EMF: Creación de modelo Ecore: Proyecto EMF.

III. Dar al proyecto el nombre de "PervMLVisualEditor".

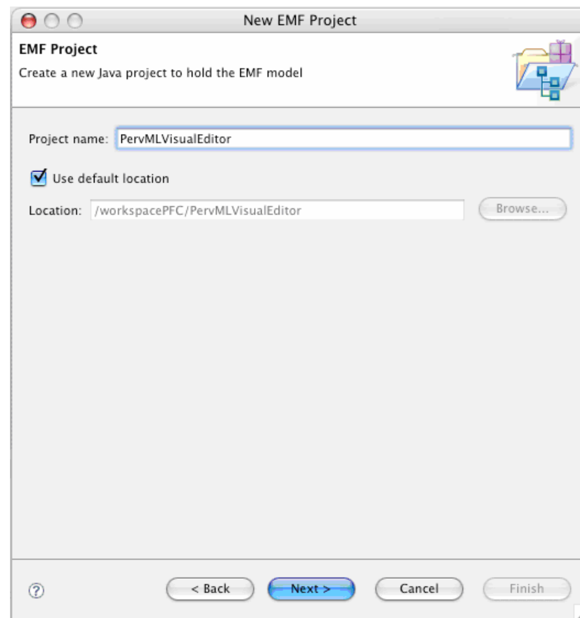


Figura 28: EMF: Creación de modelo Ecore: Nombre de proyecto.

IV. En "Model importes" seleccionar "Rose class model".

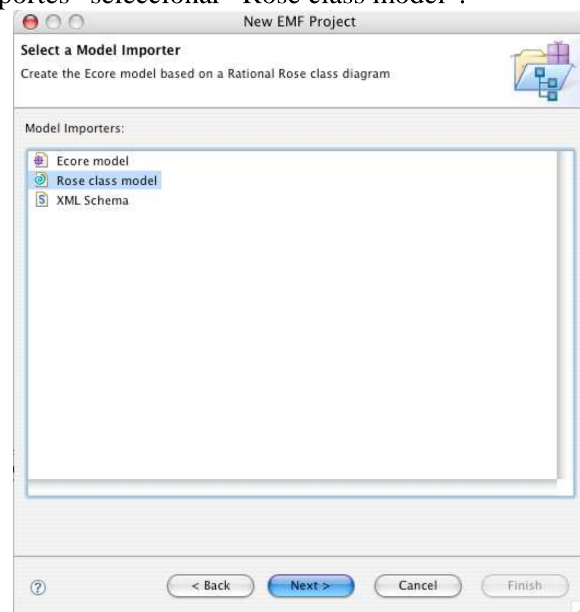


Figura 29: EMF: Creación de modelo Ecore: Origen Rational Rose.

V. Hacer clic en "Browse file system" y localizar el archivo PervMLMetamodel.mdl.

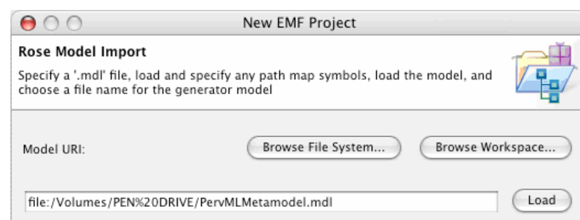


Figura 30: EMF: Creación de modelo Ecore: URI del modelo origen.

VI. El archivo será examinado y se nos sugerirá un nombre para el archivo generador, utilizar el nombre por defecto.

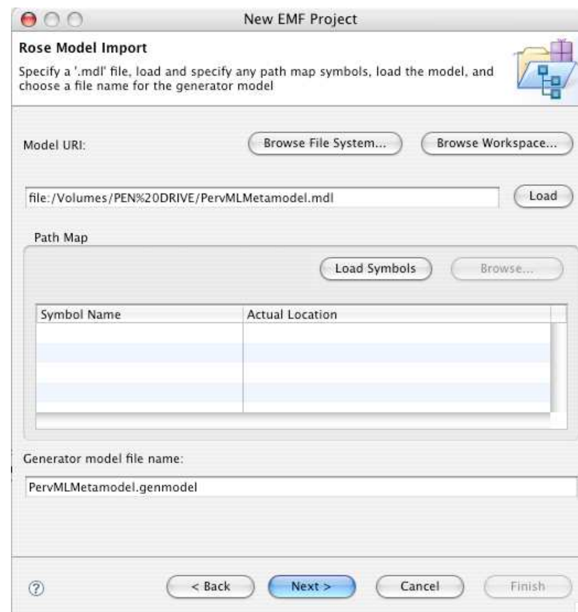


Figura 31: EMF: Creación de modelo Ecore: Archivo generador.

VII. Se mostraran todos los paquetes contenido en el archivo mdl de Ratinonal Rose. Como se persigue crear un editor para todos los modelos les lenguaje se seleccionaran todos los paquetes.

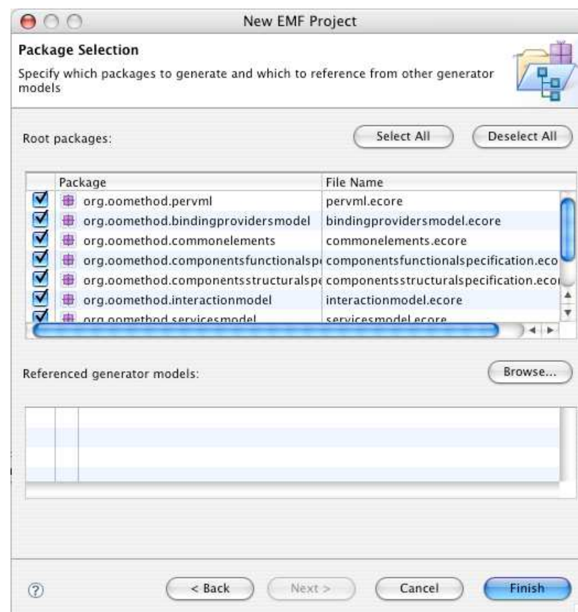


Figura 32: EMF: Creación de modelo Ecore: Selección de paquetes.

VIII. El modelo ecore (PervMLMetamodel.ecore) y el archivo generador (PervMLMetamodel.genmodel) han sido creados.

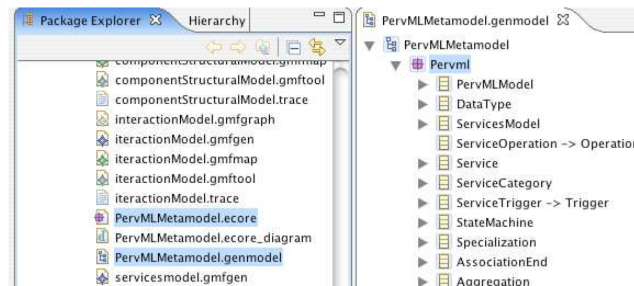


Figura 33: EMF: Creación de modelo Ecore: Resultado.

El modelo Ecore creado contiene en esencia la misma información que contenía el archivo .mdl, pero utilizando un formato con el que el plug-in EMF puede trabajar.
Es posible generar una representación grafica del modelo Ecore, para ello sólo se debe hacer clic con el botón derecho en el modelo Ecore y seleccionar “Initialize ecore_diagram diagram file”.

Tras abrir el archivo generado, “PervMLMetamodel.ecore_diagram”, con el editor asociado por defecto, se dispondrá de una representación visual del metamodelo del lenguaje PervML, similar a la disponible en la herramienta de Rational Rose .

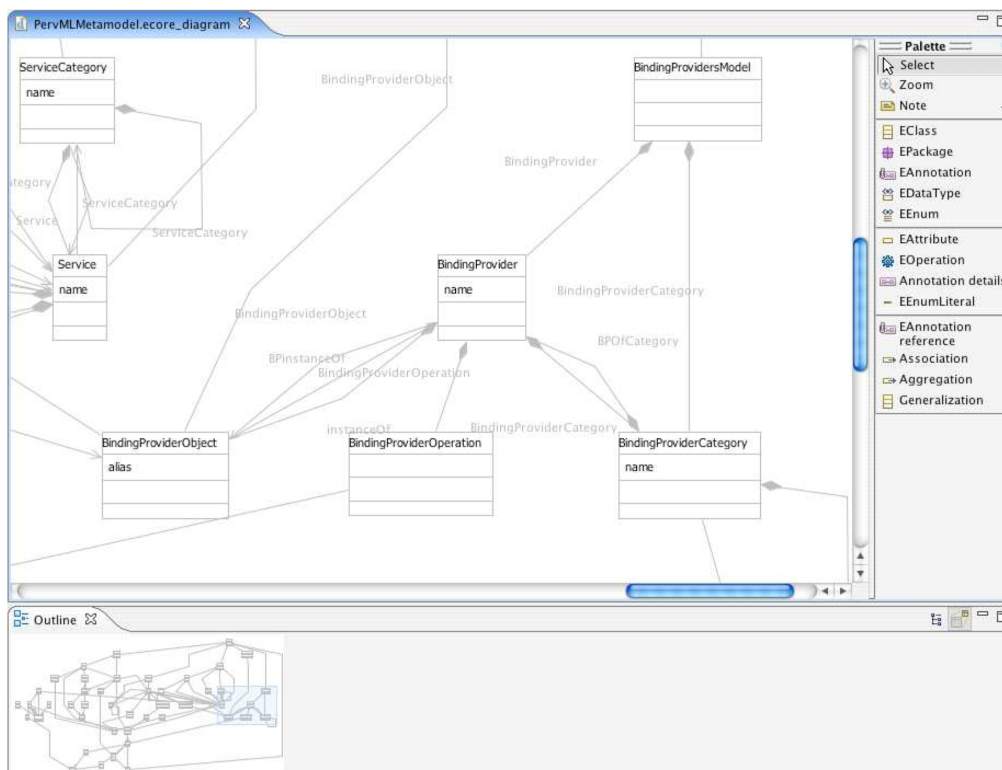


Figura 33: EMF: Representación visual del modelo ecore.

En algunos casos, tras la creación del archivo ecore_diagram puede ser necesario una redistribución de los elementos en el modelo, debido a una disposición inicial no óptima. Para ello se utiliza la función del menú contextual “Arrange all”.

5.4 Clases Java representativas de elementos del metamodelo

Tras abrir el archivo “PervMLMetamodel.genmodel” con el editor asociado por defecto, se mostrara, siguiendo una distribución en forma de árbol, un nodo raíz representando el modelo Ecore. Este nodo raíz tiene como hijos a los diferentes modelos del lenguaje PervML.
Desde el archivo genmodel se iniciara la generación de código java. Para ello se invocara la función del menú contextual “Generate Model Code” sobre el nodo raíz.

Los diferentes artefactos generados son:

- Interfaces y clases.
- Tipos enumerados.
- Clase Package (Metadatos).
- Factoría.
- Clase switch.
- Clase base para la factoría de adaptadores.
- Validadores.
- Recursos personalizados.
- Procesadores de XML.

Interfaces y clases

Cada clase del modelo ecore (EClass) se corresponde a dos elementos en java: una interfaz y su correspondiente clase de implementación. Por ejemplo la EClass para Service, se traduce en la interfaz:

```
public interface Service
```

y su correspondiente clase de implementación:

```
public class ServiceImpl extends EObjectImpl implements Service
```

Puede verse como la clase de implementación extiende de la implementación de EObject. En el punto de *tecnología utilizada* se describieron los métodos de la clase EObject, estos métodos serán parte integral de API de persistencia.

Los métodos de las clases de implementación estarán formados por métodos get y set para cada uno de los atributos de la EClass. Por ejemplo, el método get para el atributo name simplemente devuelve el valor del atributo name de este modo:

```
public String getName() {
            return name;
      }
```

El correspondiente método set actualiza el valor de la variable pero además envía una notificación a cualquier observador que pueda estar interesado en el cambio de estado:

```
public void setName(String newName) {
            String oldName = name;
            name = newName;
            if (eNotificationRequired())
                  eNotify(new ENotificationImpl(this, Notification.SET,
PervmlPackage.SERVICE__NAME, oldName, name));
      }
```

Tipos enumerados

Aquellos tipos de datos enumerados definidos en el modelo Ecore, serán traducidos a tipos de datos enumerados en java.

Factoría

Otro de los artefactos generados es la factoría. La factoria incluye métodos create por cada EClass existente en el modelo Ecore. Estos métodos create permite la instanciación de los elementos del modelo.

```
Service service=PervmlFactoryImpl.eINSTANCE.createService();
```

Clase Package (Metadatos)

La clase package generada proporciona representantes de todos los elementos en el modelo ecore. La función de estos representantes es proporcionar identificadores para manipular instancias del modelo de forma reflexiva.

Clase base para la factoría de adaptadores

También se genera un esqueleto de la clase factoría de adaptadores para el modelo. Esta base clase puede ser utilizada para implementar factorías de adaptadores que necesiten crear tipos específicos de adaptadores.

Validadores

Para garantizar que se satisfacen las restricciones definidas en el modelo Ecore se generan los siguientes validadores:

- Invariantes, definidas directamente en las clases con el estereotipo <inv>.
- Constraints.
- Multiplicidades.
- Todos los objetos referenciados están contenidos en un recurso.
- Valores validos para los tipos de datos definidos.

Clase switch

Una clase switch implementa "sentencias switch" como mecanismos de callback, para retornar instancias dependiendo del tipo de EClass. Para distinguir entre los distintos tipos de EClass posibles, se sirve de los metadatos de la clase Package. Las clases factoría se sirven de las clases switch para su implementación.

```
protected Object doSwitch(int classifierID, EObject theEObject) {
case PervmlPackage.SERVICE_OPERATION: {
      ServiceOperation serviceOperation =(ServiceOperation)theEObject;
      Object result = caseServiceOperation(serviceOperation);
      if (result == null) result = caseOperation(serviceOperation);
      if (result == null) result = defaultCase(theEObject);
      return result;
```

```
			}
case PervmlPackage.SERVICE: {
	Service service = (Service)theEObject;
	Object result = caseService(service);
	if (result == null) result = defaultCase(theEObject);
	return result;
			}
```

Recursos personalizados

Junto a las clases java se crea un manifiesto y un archivo *properties*, para que el proyectos generado pueda ser utilizado como un plug-in de eclipse.

Procesadores de XML

Finalmente también se genera un esquema XML de acuerdo al modelo Ecore.

A partir del archivo "PervMLMetamodel.genmodel" el código Java puede ser regenerado, con el objetivo de permitir modificaciones manuales en las clases java.
EMF etiquetara todos los métodos generados con:

```
//@generated
```

Cualquier método introducido a mano que no disponga de esta etiqueta será preservado frente a futuras regeneraciones del código. En casos de conflicto de nombres entre el código generado y el introducido a mano, siempre se preservara el código introducido a mano.

5.5 Clases adaptadoras

Desde el archivo .genmodel se iniciara la generación de código Java para las clases adaptadoras Para ello se invocara la función del menú contextual "Generate Edit Code" sobre el nodo raíz.

Los diferentes artefactos generados son:

- Clases adaptadoras.
- Factoría de clases adaptadoras.

En el punto de *Tecnología utilizada* se describió la jerarquía de clases utilizada en Ecore, en la que se vio que cada EClass es también un Notifier. Esto significa que frente a cambios en sus atributos o relaciones, un EClass puede enviar notificaciones. Esto es una propiedad muy importante, permite a los objetos EMF ser observados, para por ejemplo, actualizar vistas u otros objetos dependientes.

Los observadores de estas notificaciones en EMF son llamados adaptadores porque, además de sus estatus de observadores, normalmente son utilizados para extender el comportamiento del objeto al que están enlazados.

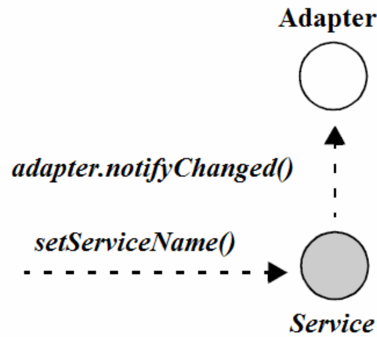


Figura 34: Invocación del método setServiceName().

Los adaptadores son la base para las interfaces de usuarios que visualizan los modelos y para la implementación de operaciones sobre elementos que tiene implicaciones en otros elementos. Para llevar a acabo estas funciones los adaptadores deben representar 4 roles:

1. *Implementar funciones para proveer etiquetas y contenidos*. Con el objetivo de independizar posibles interfaces de usuario de los modelos, los adaptadores proporcionan de forma homogénea acceso a los nombres de los objetos EMF y listados con los objetos agregados dentro de un objeto EMF.

2. *Proporcionar descriptores de propiedades para los objetos EMF*. De nuevo con el objetivo de independizar posibles interfaces de usuario de los modelos, los adaptares proporcionan las propiedades de los elementos del modelos dentro de clases envoltorio que abstraen de la heterogeneidad de tipos en el modelo y permiten manipular las propiedades de una manera homogénea.

3. *Actuar como una factoría de comandos para sus elementos asociados del modelo*. Los adaptadores proporcionan todos los mecanismos para modificar los objetos EMF, incluyendo un conjunto completo de comandos genéricos.

4. *Propagar notificaciones de los elementos del modelo*. Los adaptadores serán notificados frente a cambios de estado en los objetos EMF a los que están en lazados. La responsabilidad de los adaptadores es filtrar todos los eventos no interesantes y propagar el resto a cualquier objeto subscrito.

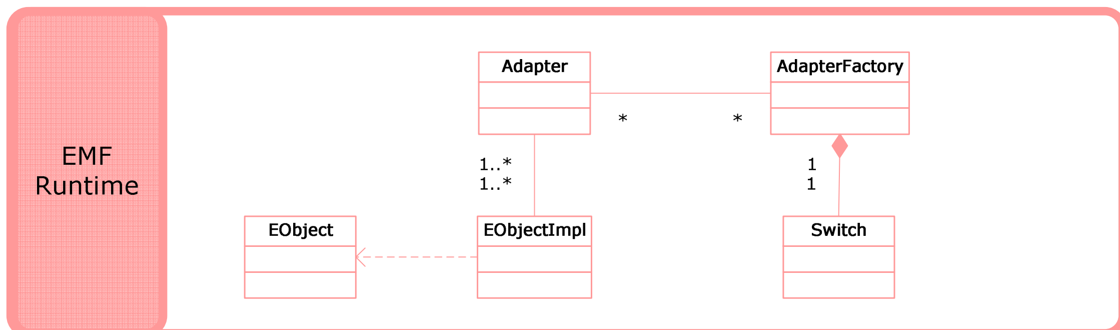


Figura 35: Diagrama de clases de EMF.

Como resuemen de Eclipse Modeling Framework en la figura 35 se representán las distintas clases EMF que han sido descritas a los largo de los anteriores puntos y sus relaciones entre ellas.

5.6 Editor simple de modelos PervML

Desde el archivo .genmodel se iniciará la generación de código Java para un editor simple de model en forma de árbol. Para ello se invocara la función del menú contextual "Generate Editor Code" sobre el nodo raíz.

Los diferentes artefactos generados son:

- Asistente para la creación de modelos.
- Editor.
- Action bar contributors.

En los puntos anteriores se ha visto como a partir de un metamodelo se ha generado una implementación con clases Java. Además con EMF también se ha generado un editor que permite visualizar y editar (copiar, pegar, seleccionar y arrastrar…) instancias de modelo, utilizando visores estándar JFace y una vista de propiedades.

Es posible visualizar la estructura del modelo siguiendo una disposición de árbol y editar las propiedades de los elementos utilizando la vista de propiedades. Todo esto integrado como un plug-in de eclipse.

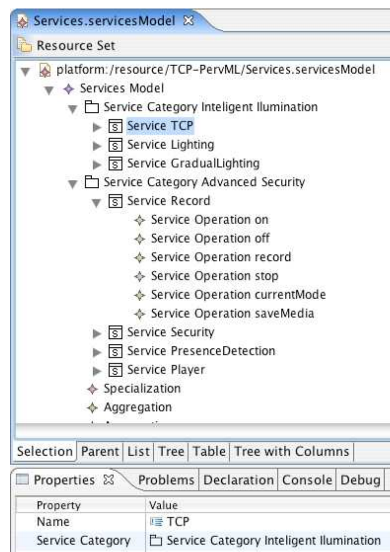


Figura 36: Editor de modelos EMF.

Conceptos básicos del editor

- *Población de vistas a partir de un modelo*: En lugar de consultar directamente a las objetos del modelo, el editor accede a los adaptares para navegar por los elementos del modelo y obtener: nombres, iconos y elementos contenidos.

- *Edición de propiedades*: De manera análoga a la población de vistas, para la edición de propiedades no se accede directamente a los elementos del modelo, de nuevo se consulta las propiedades de los elementos a través de sus adaptadores.

- *Mecanismo de acciones*: La última parte importante del editor son los mecanismos de acciones. Las acciones representas los comandos que pueden ejecutarse desde opciones de menú o desde la barra de herramientas.

Para crear y manejar acciones con origen en el editor se utilizan especializaciones de la clase EditorActionBarContributor. Junto con la generación del código del editor se crean especializaciones de EditorActionBarContributor para las acciones más comunes que se realizaran en el editor, como: cambiar el elemento seleccionado, crear elementos al mismo nivel o crear subelementos dentro de un elemento.

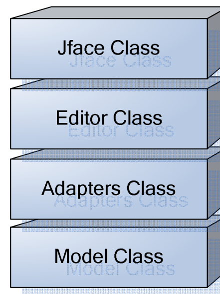


Figura 37: Capas del editor EMF.

Finalmente para la creación de un modelo nuevo se genera un asistente que puede ser invocado desde "File/new/other/Example EMF model creation wizards". Tras indicar un nombre para el archivo que contendrá el modelo y un elemento del lenguaje PervML que se considerara raíz, se abrirá una nueva instancia del editor generado.

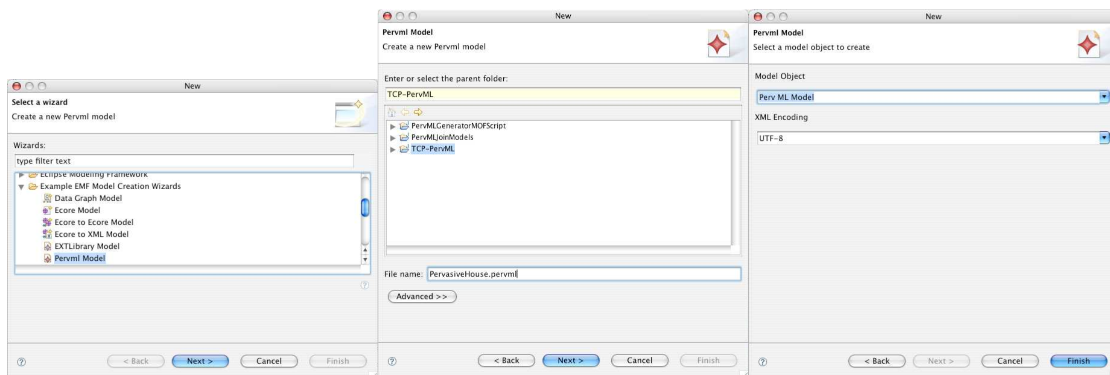


Figura 38: Asistente para la creación de modelos EMF.

5.7 Persistencia de modelos

La habilidad de almacenar y referenciar otros modelos almacenados es uno de los más importantes beneficios de utilizar modelado con EMF. El framework de EMF provee simples pero potentes mecanismos para manejar la persistencia de objetos.

XMI es por defecto el formato de serialización utilizado para los modelos en EMF. XML metadata interchage es el estándar que conecta modelado con XML, definiendo un camino simple para especificar objetos del modelo en documentos XML. La estructura de un documento XMI es muy cercana a la de su modelo correspondiente, con los mismos nombres y jerarquía de elementos que en el modelo. Como resultado la relación entre un modelo y su serialización es fácil de comprender.

Ejemplo de la relación entre un modelo y su representación en formato XMI:

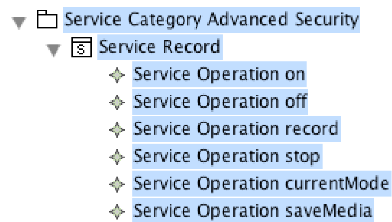


Figura 39: sección de un modelo con el editor de árbol.

```
<ServiceCategory name="Advanced Security"
Service="//@ServiceCategory.1/@ServiceOfCategory.0
//@ServiceCategory.1/@ServiceOfCategory.1
//@ServiceCategory.1/@ServiceOfCategory.2
//@ServiceCategory.1/@ServiceOfCategory.3">
<ServiceOfCategory name="Record"
ServiceCategory="//@ServiceCategory.1">
<ServiceOperation name="on"/>
<ServiceOperation name="off"/>
<ServiceOperation name="record"/>
<ServiceOperation name="stop"/>
<ServiceOperation name="currentMode"/>
<ServiceOperation name="saveMedia"/>
</ServiceOfCategory>
```

El anterior fragmento de código representa la parte de la serialización correspondiente a los electos de la figura 39.

5.8 Clases test

Desde el archivo .genmodel se iniciara la generación de los esqueletos para las clases de test de JUnit Para ello se invocara la función del menú contextual "Generate Test Code" sobre el nodo raíz.

Los diferentes artefactos generados son:

- Casos de prueba.
- Suites de casos de prueba.
- Ejemplos independientes.

JUnit es un conjunto de clases que permite realizar 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 se 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 que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente.

Para la ejecución de un caso de prueba es necesario es necesario que las variables se encuentren inicializadas. En nomenclatura JUnit el conjunto de variables inicializadas necesario para ejecutar un caso de prueba es conocido como *fixture*.

Junto a los esqueletos para los casos de prueba EMF genera *fixtures* por defecto para todos los elementos del modelo. Delegando en el programador la especificación de la pruebas a ejecutar.

6 Edición Gráfica de modelos

En el capitulo *Gestión de modelos* mediante el plug-in EMF, se genero un editor de modelos completamente funcional. En ese capitulo se pretende desarrollar un editor grafico de modelos mas rico. Este editor se servirá de metáforas graficas para representar los elementos del modelo, permitiendo manipular los modelos de una forma más cómoda para el usuario.

La figura 40 presenta una captura de pantalla del editor generado en el capitulo anterior, en ella se esta visualizando un modelos de servicios del lenguaje PervML. La figura 41 presenta el mismo modelo se servicios de la figura 40, pero mostrado con el editor grafico que se describirá en este capitulo.

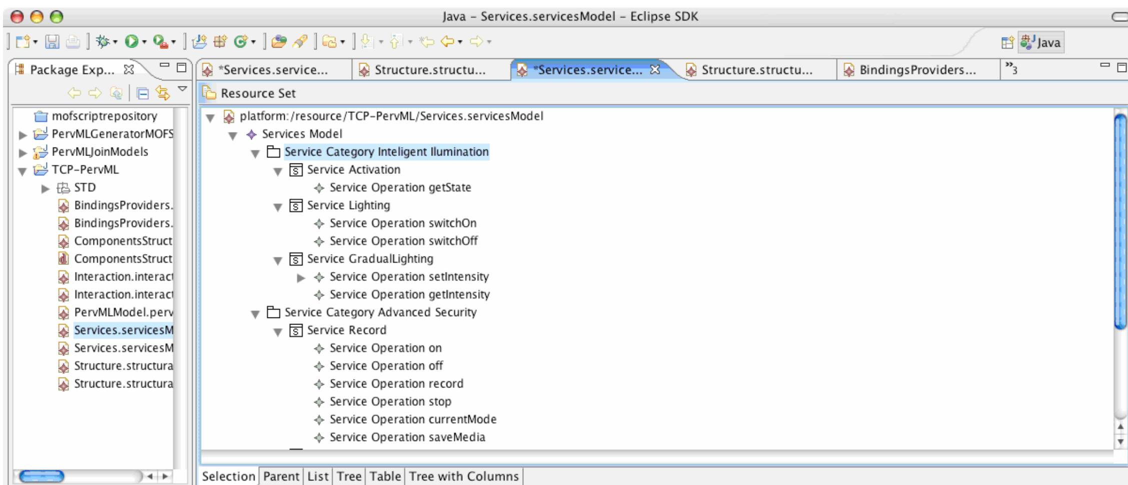


Figura 40: Editor de modelos EMF.

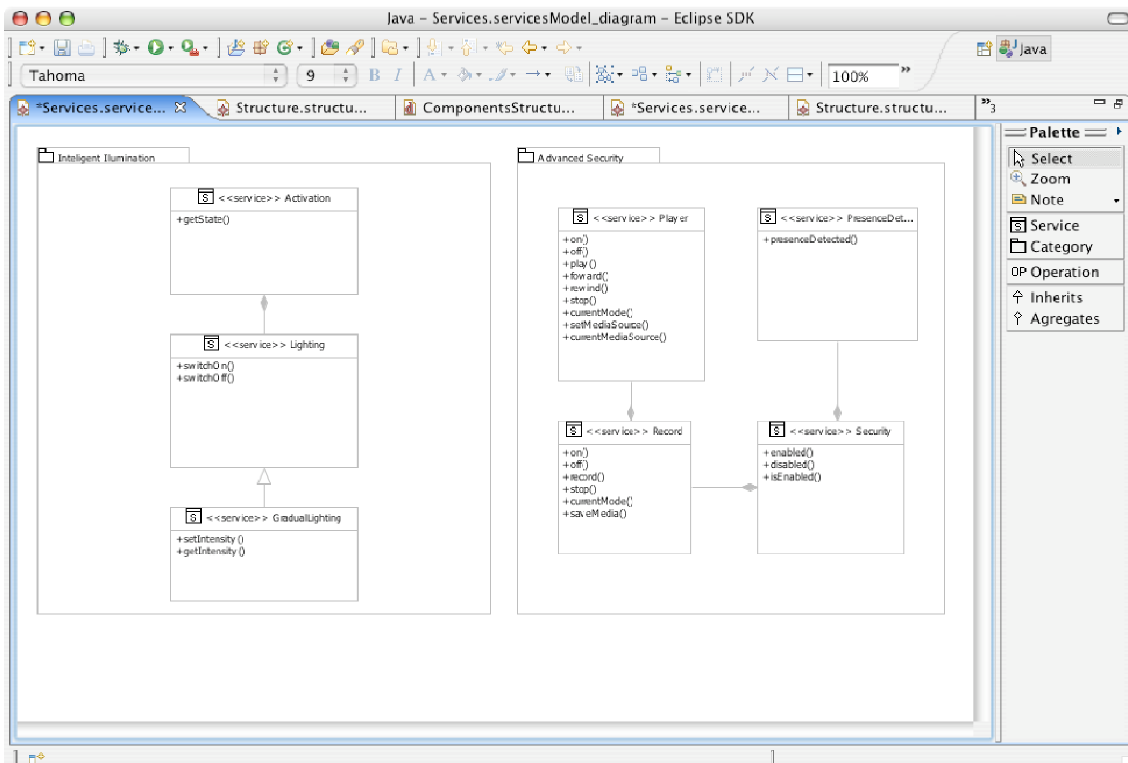


Figura 41: Editor de modelos GMF.

6.1 Tecnología utilizada

El desarrollo de un editor gráfico no es una tarea sencilla, ocupa temas tan heterogéneos como la representación gráfica de elementos, interrelaciones con modelos o cálculos de intersecciones de rectas, entre otros muchos. Si se observan distintos editores gráficos es posible factorizar elementos comunes a todos ellos, tanto estructurales: paleta de herramientas, lienzo para dibujar el modelo… como funcionales: opciones de edición (copiar, cortar, pegar), deshacer…

Con la motivación de:

- No reimplementar factores comunes a los editores.
- Heredar una buena arquitectura de código.
- Embeber buenas prácticas.
- Incrementar la productividad.

se desarrollará el editor grafico utilizando el plug-in para el entorno de desarrollo Eclipse GEF, Graphical Editing Framework. Para facilitar la tarea, se ha hecho uso de un nuevo plug-in (Graphical Modeling Framework, GMF) que facilita enormemente el desarrollo de editores gráficos con esta tecnología. Pese a esto, todavía es necesario conocer las interioridades de GEF para comprender el funcionamiento del editor.

Graphical Editing Framework

GEF parte de la premisa de que existe un modelo que se desea visualizar y editar gráficamente, a través de una representación grafica. Si algún elemento del modelo cambia de alguna manera, se debe asegurar que su representación grafica se actualiza consecuentemente. Esto requiere una comunicación entre el modelo y su representación grafica. No es una practica aceptable que la representación grafica interactué directamente con los elementos del modelo, ya que esto incurriría en un alto nivel de acoplamiento.

Para resolver este problema GEF recurre al patrón de diseño Model-View-Controller [6]

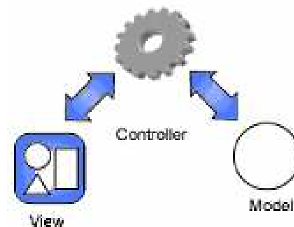


Figura 42: Model-View-Controller.

Los controladores son el puente entre la vista y el modelo, cada controlador, o EditPart utilizando la nomenclatura de GEF, es responsable tanto de asociar el modelo a la vista, como de realizar los cambios en el modelo. Los EditParts también observan el modelo y actualizan la vista para reflejar los cambios en el estado del modelo.

EDitParts

EditParts son los elementos centrales en las aplicaciones GEF. Ellos son los controladores que especifican como los elementos de los modelos se asocian a figuras visuales y como estas figuras se comportan en diferentes situaciones.

Existen 3 tipos diferentes de EditParts:

- *GraphicalEditParts*: proveen una representación grafica para el modelo.
- *ConnectionEditParts*: representan conexiones entre GraphicalEditParts.
- *TreeEditParts*: representan modelos con una disposición de arbol.

Gestión de los EditParts

Cada vista está configurada con una factoría para crear EditParts, cuando se establece el contenedor de la vista, se suele hacer proporcionando el objeto del modelo que representa el punto de entrada para la vista. Este punto de entrada es típicamente un objeto raíz del modelo. La vista utiliza la factoría para construir el EditPart del objeto punto de entrada, a partir de ahí, el EditPart solicitará a la factoría nuevos EditParts por cada uno de los elementos del modelo que contiene su elemento del modelo asociado. Este proceso se repetirá iterativamente hasta que todos los elementos del modelo dispongan de su EditPart.

En el momento que un elemento del modelo es eliminado, su EditPart asociado desaparece. Si por una acción de deshacer fuera recreado el elemento del modelo, sería un nuevo EditPart el que se le asociaría y no el que existía antes.

View

Java Draw 2D proporciona un sistema grafico ligero del que GEF depende para realizar la visualización. GEF hace uso principalmente de dos elementos de java 2D: el lienzo y las figuras. El lienzo es el área central de la edición de modelos, donde se representan las metáforas visuales de los elementos del modelo que son las figuras. Las figuras pueden tener formas arbitrarias no rectangulares y pueden jerarquizarse para componer figuras complejas. Java. 2D es una biblioteca independiente de gráficos muy amplia y cubrirla en profundidad se escapa del objetivo de esta sección.

Modelo y diagrama

Toda la información está contenida en el modelo. El modelo es lo único que persiste y es restaurado. Durante el curso de las ediciones, deshacers, y rehacers el modelo es lo único que permanece. Figuras y EditParts, como objetos en memoria que son, serán desreferenciados, eliminados por el recolector de basura y vueltos a recrear cuando sean necesarios.

Cuando el usuario interactúa con los EditParts, el modelo no es manipulado directamente por los EdiParts. En su lugar, un objeto Command que encapsula el cambio es creado. Los objetos Command pueden ser utilizados para validar la interacción del usuario y proveer soporte para hacer y deshacer.

Los objetos Command son el medio por el que el modelo es editado. Son utilizados para realizar todos los cambios invocados por el usuario. Idealmente los objetos Command solo deberían interactuar con el modelo, no deberían referenciar EditPart ni figuras, ni siquiera para invocar diálogos de consulta al usuario.

Un editor grafico de modelos básicamente se compone de un diagrama, en el que se dibujan una figuras, y un modelo al cual representan las figuras. Esta representación visual de los elementos del modelo en el diagrama, las figuras, hace emerger nuevos atributos que deben ser almacenados: posición (x, y), tamaño, color,etc.

Una alternativa para almacenar estos atributos es añadir la información emergente en los elementos del modelo. Esta opción platea dos problemas; el modelo es contaminado con atributos que no le pertenecen y un mismo elemento del modelo puede ser representado varias figuras, con lo que se debería desambiguar a qué figura pertenece cada atributo.

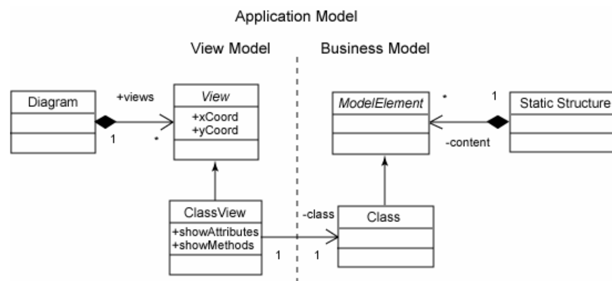


Figura 43: División entre diagrama y modelo.

En el editor grafico que se describirá en este capitulo, se utilizara la palabra modelo manteniendo el significado que se le daba en el capitulo *Gestión de modelos* y se empleara diagrama, al referirse a las representaciones visuales de los elementos del modelo. Para almacenar la información emergente de la representaciones visuales, no se utilizará el modelo, sino que se mantendrá un archivo a parte.

Secuencia de mensajes

En el punto anterior se ha visto que ante peticiones por parte del usuario no son los EditParts quienes interactúan directamente con los elementos del modelo, sino que se crean objetos Command que encapsulan el cambio.

En este punto se describe con más nivel de detalle los elementos que intervienen y la secuencia de mensajes entre estos, desde que el usuario realiza una acción en el editor grafico, esta realiza cambios en el modelo y finalmente el diagrama se actualiza para reflejar el nuevo estado del modelo.

Todas las acciones que el usuario realiza sobre la interfaz grafica, crear un elemento en el diagrama, arrastrar y soltar elementos, eliminar un elemento del diagrama, se traducen en la creación de un objeto Request. Estos objetos Request son enviados al EditPart asociado al elemento que se manipula. El EditPart no se encarga de procesar el objeto Request, sino que lo delega al objeto EditPolicie oportuno. Es el objeto EditPolicie quien crea el objeto Command, que finalmente ejecutara la acción invocada por el usuario.

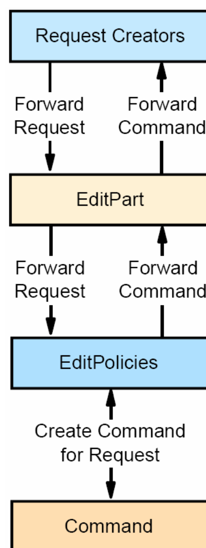


Figura 44: Cadena de comunicación.

Se introducen los objetos EditPolicies dentro de la cadena de comunicación de mensajes de GEF porque respestan la filosofía de orientanción a objetos. Las EditPolicies definen que acciones pueden hacerse contra un objeto EditPart. Se produce una delegración de responsabilidad de los EditPart en los EditPolicies.

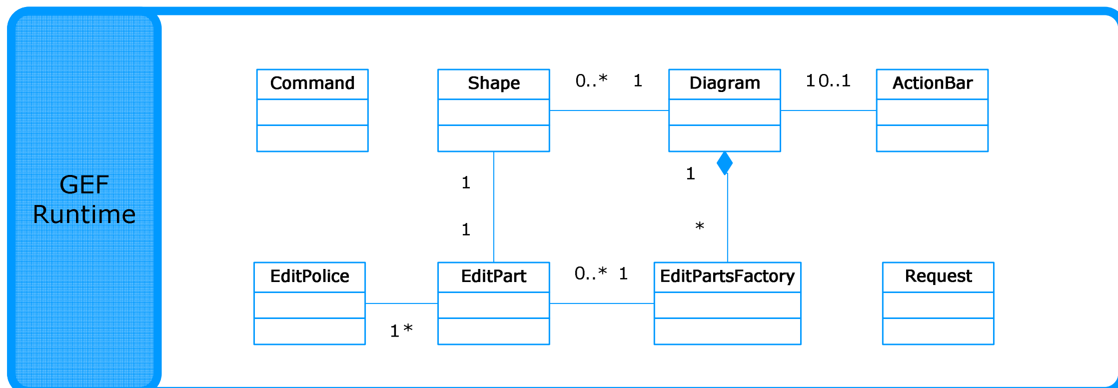


Figura 45: Diagrama de clases de GEF.

Como resumen de Graphical Editing Framework en la figura 45 se representan las distintas clases GEF que han sido descritas a los largo de los anteriores puntos y sus relaciones entre ellas.

Graphical Modeling Framework

Para la creación de editor grafico de modelos, utilizando el framework que proporciona GEF, deberemos implementar:

- EditParts:
  - Conexión con elementos del modelo.
  - Conexión con representación visual.
  - Acciones ante cambios del modelo.
  - Acciones para edición del modelo.
- Factorías de EditParts.
- Políticas de edición.
- Recursos gráficos.

El plug-in de eclipse GMF, Graphical Modeling Framework, nos abstrae de los detalles de implementación de GEF, permitiendo definir el editor grafico a un alto nivel de abstracción. Mediante modelos declararemos los elementos que aparecerán en el diagrama, pudiendo elegir entre un catalogo ya existente de figuras. Tambien estableceremos en los modelos las relaciones entre elementos del modelo y elementos del diagrama, lo que desencadenará en la creación de forma implícita de EditParts con sus conexiones a los elementos. Se asociará al diagrama del editor gráfico una factoría de EditParts, que contemplará la creación de todas las EditParts generadas. Finalmente se proporcionará para cada EditPart las políticas de edición más comunes;

- Creación.
- Edición.
- Reposicionamiento.
- Eliminación.

GMF no solo facilita la definición de un editor gráfico con GEF como base, además proporciona una serie de características ya implementadas:

- Compartimentos expandibles y comprimibles:
  Las figuras compuestas, también llamadas figuras compartimiento pueden comprimir y expandir sus secciones.

- Edición directa:
  El texto dentro de las etiquetas puede ser directamente editado e interpretado.

Figura 46: GMF: Edición directa.

- Barras pop-up:
  Proporcionan accesos directos para la creación de elementos en el contexto del elemento que se encuentra debajo de la posición del ratón.

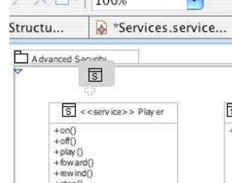


Figura 47: GMF: pop-up para crear servicios.

- Encargados de conexión:
  En los elementos del diagrama, que pueden relacionarse con otros elementos, aparecen unas flechas flotantes que permiten crear directamente el elemento con el que se relacionan y la relación.

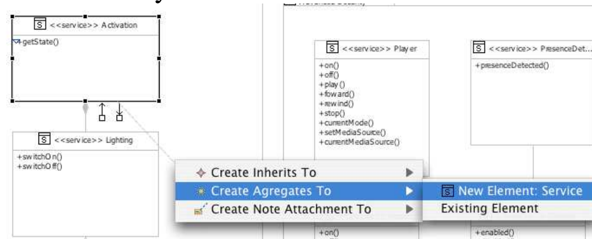


Figura 48: GMF: Encargados de conexión de servicios.

- Herramientas comunes:
  Por defecto se proporciona soporte a la creación de comentarios y el enlace de estos con cualquier elemento del diagrama.

- Comandos de menú comunes:
  GMF hace disponible un número de comandos de menú estándares para el diagrama:
    - Selección de fuente.
    - Color de relleno.
    - Estilo de línea.
    - Selección de elementos por filtros.
    - Alineación.
    - Ocultación de elementos.
    - Redimensionamiento automático.
    - Copia de formato.

- Barras de herramientas comunes:
  Los controles avanzados están disponibles desde la barra de menú.

Figura 49: GMF: Barra de herramientas.

- Zoom.
- Algoritmo de layout.
- Propiedades comunes:
  GMF también proporciona por defecto una implemetación para la edición de los atributos relacionados con la apariencia, de los elementos del diagrama.
- Impresión y visualización preliminar.

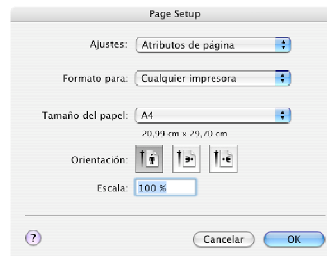


Figura 50: GMF: Opciones de impresión.

GMF permite reutilizar componentes estandarizados de alta calidad en el editor grafico de modelos.

Proceso de creación

Para el desarrollo de un editor grafico, GMF propone cuatro modelos:

- *Modelo de definición grafica:*
  Se declara que elementos aparecerán en el diagrama y para cada elemento cual será su representación grafica.
- *Modelo de herramientas:*
  Enumeración de las posibles acciones de creación que pueden ser invocadas desde la barra de herramientas. Es posible definir agrupaciones sobre ellas.
- *Modelo de asociación:*
  Para cada elemento del modelo que se desee que tenga representación grafica, se define una relación con un elemento de la definición grafica y otro del modelo de herramientas.
- *Modelo de generación:*
  Permite establecer parámetros relacionados con el editor:
  - Modificar extensiones de los archivos de diagrama y modelo.
  - Permitir imprimir los diagramas.
  - Decidir si se almacenar la información del diagrama junto al modelo.

  Como el editor grafico se generara lo hará como un plug-in de eclipse, es en este modelo donde se pueden establecer los parámetros relacionados:
  - Nombre del plug-in.
  - Nombre del desarrollador del plug-in.
  - ID con el que se registrara el plug-in.
  - Establecer la extensión para los paquetes java que se generaran.

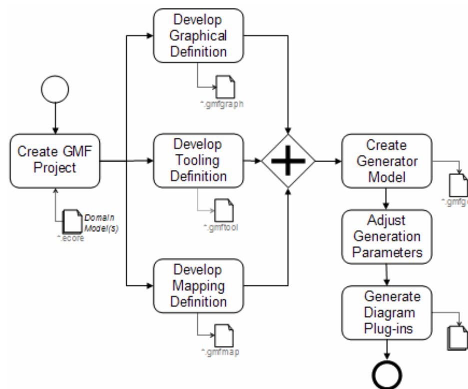


Figura 51: Esquema de modelos en GMF.

La figura 51 describe el proceso de creación del editor grafico utilizando GMF. Partiendo de un modelo Ecore ya existente se crean el modelo de definición grafica y el de herramientas. A partir de estos tres modelos,Ecore: (1) metamodelo del lenguaje, (2) definición grafica, elementos del diagrama, y (3) modelo de herramientas, iconos de la barra de herramientas, se realizaran las asociaciones entre ellos en modelo de asociación.

A partir del modelo de asociación, GMF comprueba su corrección y deriva de forma automática el modelo de generación, estableciendo por defecto, todos los parámetros relacionados con el plug-in de eclipse que se generara.

Aproximación multivista

Originalmente GMF propone la creación de un editor grafico para modificar desde un único diagrama todos los elementos de un modelo. En el metamodelo de PervML existen cuarenta y cinco elementos distintos, además de sus relaciones. Considerando que en un modelo de PervML suelen crearse varios elementos por cada elemento del metamodelo, se pueden obtener modelos de dimensiones considerables. Editar un modelo de ese tamaño a través de un único diagrama, puede reducir considerablemente la usabilidad del editor grafico.

Para abordar este problema, el lenguaje propone el uso de diferentes diagramas, donde cada diagrama será una vista sobre uno de los sub-modelos del lenguaje PervML. Aunque estos sub-modelos describen diferentes vistas de un sistema pervasivo, existen relaciones entre ellos. Es por esto que los distintos diagramas deben estar relacionados y ser consistentes entre si.

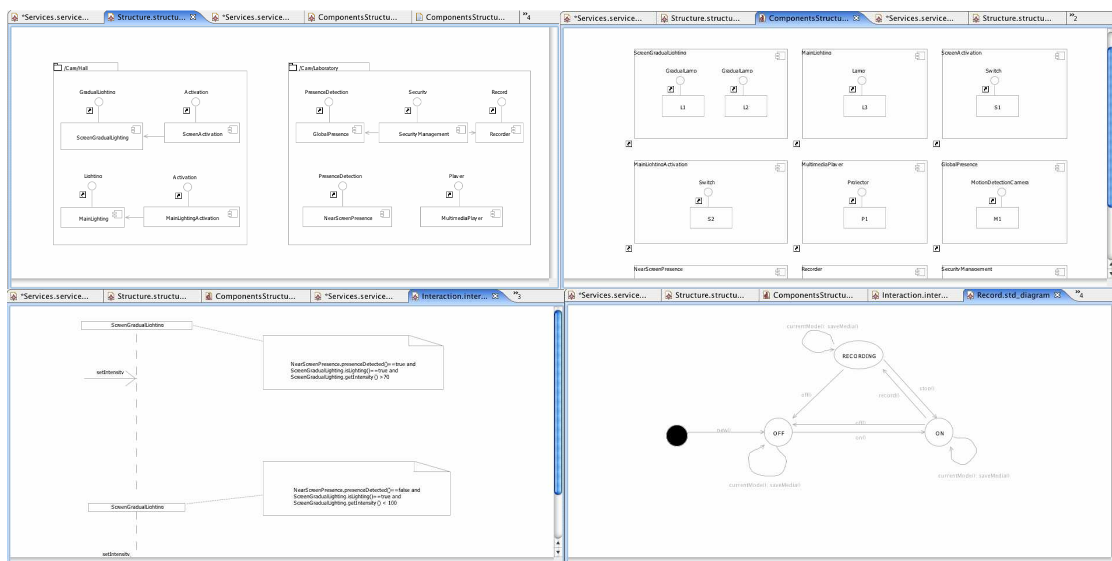


Figura 52: Diagramas relacionados.

Relaciones de dependencia entre modelos

En ocasiones los elementos de una vista necesitarán relacionarse con elementos de otra vista. Pero desde una vista sólo se pueden alcanzar otros elementos de la misma vista, o elementos del modelo al que está asociado la vista, aunque éstos no tengan representación gráfica.

Esta necesidad puede aparecer en dos casos:

I. Se necesita una representación grafica de un elemento de otra vista para establecer una relación con él.
II. Se debe completar un atributo con un elemento de otra vista, no siendo necesaria su representación gráfica.

Para dar soporte al primer caso se necesitara de una característica de GMF llamada "shortchuts". Los "shortcuts" permiten representar un elemento de otro modelo en la vista actual. Esta representación implica que el elemento no se replicará y sólo se almacenarán las características de su representación grafica: posición, tamaño, color, etc. Para poder utilizar elementos de otros modelos con este procedimiento se deben realizar las siguientes acciones:

Suponiendo que el editor A, con extensión "extA", necesita acceder a elementos del editor B, con extensión "extB".

- En el modelo .gmfgen del editor B: "Shortcuts Provided For" = "extA",
- En el modelo .gmfgen del editor A: "Contains Shortcuts To" = "extB",
- En el modelo .gmfgraph se deberá declarar una representación grafica para el elemento.
- En el modelo .gmfmap se realizara la asociación entre el elemento y su representación gráfica.

Para resolver el segundo caso únicamente se deberá utilizar una funcionalidad heredada de EMF. Desde cualquier parte del diagrama en el editor grafico se puede invocar con el menú contextual la acción "Load resource" que permitirá cargar otro modelo y a sus elementos.

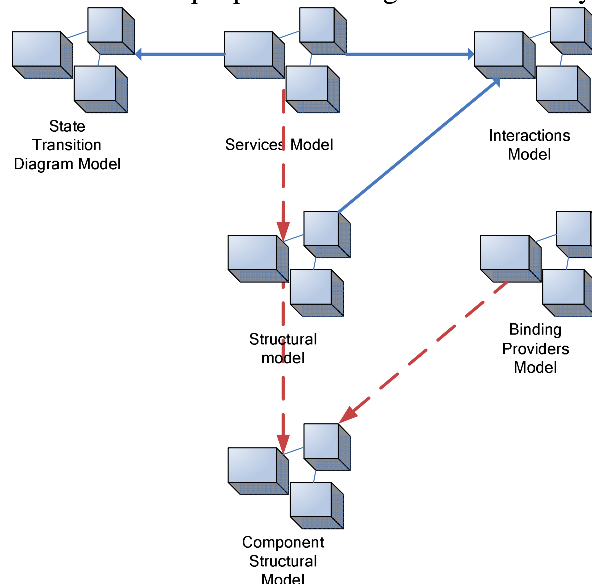


Figura 53: Relaciones entre los modelos de PervML.

Las flechas azules continuas de la figura 53 indican que el modelo apuntado por la flecha utiliza elementos sin representación visual del modelo origen de la flecha. Mientras que las flechas discontinuas rojas representan el uso de elementos del modelo origen con necesidad de representación visual.

Colisiones entre editores

Cuando un plug-in se registra en el entorno de desarrollo Eclipse, el plug-in proporciona cuatro datos acerca de él: nombre, proveedor, versión e ID. De estos cuatro datos, el nombre, proveedor y versión son únicamente descriptivos, mientras que el ID identifica al plug-in dentro del entorno eclipse, y por ello debe ser único y no nulo. En ninguno de los modelos de GMF anteriores al modelo de generación se indica el ID del plug-in. Por ello es derivado de forma automática del modelo ecore.

Siguiendo la aproximación multivista, contra un mismo modelos Ecore se crean varios editores gráficos utilizando GMF, es por ello que todos estos editores gráficos son creados con el mismo ID en el plug-in. Para evitar este comportamiento, se deberá modificar el valor del ID por defecto en el modelo de generación antes de proceder a la generación de las clases Java.

Además GMF genera un asistente para la creación de los modelos que se manipularán con el editor gráfico. De nuevo este depende de un ID, “model ID” que se deriva del modelo Ecore. Por ello esta propiedad debe ser modificada en el modelo .gmfgen tras su inicialización con el valor por defecto.

Extensión del editor EMF

Como se ha comentado en puntos anteriores, el editor grafico almacenará los atributos emergentes de las representaciones graficas (posición, colores, etc.) separados del modelo. Heredado del plug-in EMF todavía se dispone del editor simple en forma de árbol. Este editor era capaz de trabajar con recursos que cumplan dos condiciones:

1. La extensión del recurso debe ser .pervml.
2. Su elemento raíz debe ser un elemento del metamodelo de PervML.

Los nuevos archivos en los que se almacenen los modelos, estarán poblados por elementos de acuerdo a subconjuntos del lenguaje PervML, por lo que siempre el elemento raíz será de acuerdo a un elemento del metamodelo. Pero al existir diferentes vistas que pueden ser editadas de forma simultánea, las extensiones de los recursos deberán ser diferente entre sí, incumpliéndose la condición número 1. Puesto que el editor EMF esta orientado a editar un único tipo de modelo, no permite la asociación con mas de una extensión en su archivo .genmodel. El archivo .genmodel como se comentó en el capitulo *gestión de modelos* comprende todos los aspectos relacionados con la creación del plug-in Eclipse.

Para lograr que el editor EMF reconozca los archivos con las nuevas extensiones, se deberá añadir en el archivo plugin.xml del del editor generado, una entrada del siguiente estilo por cada nueva extensión:

```
<extension point = "org.eclipse.ui.editors">
    <editor
        id = "pervml.presentation.PervmlEditorID"
        name = "%_UI_PervmlEditor_label"
        icon = "icons/full/obj16/PervmlModelFile.gif"
        extensions = "servicesModel"
        class = "pervml.presentation.PervmlEditor"
   contributorClass="pervml.presentation.PervmlActionBarContributor" >
    </editor>
 </extension>
```

En la entrada extensión es donde declarar el nombre de la nueva extensión que se desea permitir editar.

6.2 Recursos graficos

Tal y como se ha visto anteriormente GMF propone un modelo en el que se declaran todas representaciones visuales que se utilizaran en el editor grafico. Los posibles recursos gráficos que se pueden utilizar son:

- Figuras predefinidas:
    - Rectángulo.
    - Rectángulo redondeado.
    - Elipse.
    - Polígono.
- Figuras personalizadas GMF.
- Figuras java 2D.
- Líneas.
- Decoradores.
- Contenedores.
- Texto.
- Layouts predeterminados.
- Layouts personalizados.

Las figuras son los representantes en el diagrama de los elementos del modelo. Pueden albergar texto en su interior o mediante los contenedores otras figuras. La disposición de los elementos dentro de una figura se realiza de acuerdo a los *layouts*. Las líneas relacionan figuras del diagrama entre sí y pueden ser complementadas gráficamente con decoradores.

En el caso de que sea necesario figuras distintas a las predefinidas, GMF proporciona primitivas para describir de forma declarativa nuevas figuras personalizadas. Para declarar las nuevas figuras se indica los puntos que unidos formaran la silueta de la figura así como el color de relleno y el del borde. Es posible formar figuras más complejas a base de componer figuras simples. La ventaja de esta aproximación es que describimos las figuras a un alto nivel de abstracción, pero como inconvenientes nos encontramos que nuestras figuras están descritas con puntos fijos por lo que no podrán ser redimensionadas.

Figura 54: Figura redimensionada.

Ante la necesidad de describir una figura que modifica su aspecto al ser redimensionada, como ocurre en la figura 54, no encontramos la expresividad suficiente en las primitivas proporcionadas por GMF. En ese caso recurriremos a java 2D, para ello extenderemos la clase Shape e implementaremos su dos métodos abstractos: fillShape y outlineShape.

```
public class myFigure extends Shape
{
      protected void fillShape(Graphics graphics)
            {…
            }
      protected void outlineShape(Graphics graphics)
            {…
            }
}
```

El objeto de las clase Graphics indica el área que ha seleccionado el usuario para dibujar la figura. En el método outlineShape se define la silueta de la figura, considerando que solo será visible la parte de la silueta contenida dentro del área delimitada por el usuario. Finalmente en el método fillShape se aplican los colores a la figura. Frente a la aproximación de describir las figuras personalizas utilizando las primitivas de GMF, Java 2D nos permite representar cualquier tipo de figura sin ninguna limitación, pero por contra debemos especificar las figuras a bajo nivel.

En el interior de las figuras pueden mostrarse atributos como cadenas de texto u otras figuras. La disposición de estos elementos viene determinada por el layout que tengan asociadas las figuras contenedoras. GMF ofrece por defecto los layouts que se pueden encontrar en Java 2D. Pero al igual que puede ser interesante definir nuevas figuras, puede ser necesaria la creación de nuevos layouts. Para la definición de nuevos layouts GMF no proporciona primitivas, pero permite realizar composición de layouts ya existentes o la utilización de clases java que contengan layouts de java 2D validos.

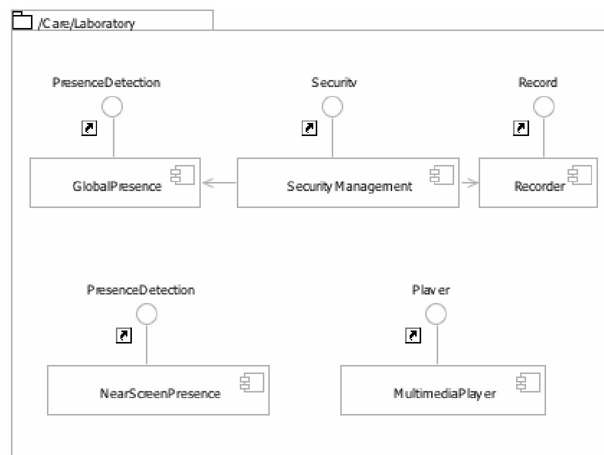


Figura 55: Localización con componentes.

En la figura 55 se ha aplicado un layout personalizado. Este layout asigna la posición central superior al primer elemento añadido a la figura, impidiendo que sea reubicado, y el resto de elementos que se añadan son colocados en el cuadrante inferior de la figura, permitiendo que sean reubicados.

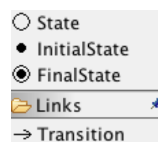


Figura 56: Iconos en la paleta.

Para la personalización de los iconos que aparecerán junto a las acciones en la barra de herramientas, se deben especificar la ruta, siguiendo la filosofía de paquetes en java y el nombre del icono.
Estos iconos deben cumplir unas restricciones para visualizarse:

- Dimensiones de 16 pixeles de alto por 16 pixeles de ancho.
- Profundidad de color de 8 bits.

Además GMF decorara las figuras del diagrama con los iconos por los cuales eran representadas en el editor EMF. Estos iconos se encuentras en el plug-in que generó EMF con los adaptadores, dentro de la carpeta '/icons/full/obj16/'. Por cada elemento del metamodelo, encontraremos un archivo con su nombre y la extensión "ico". Estos iconos deben cumplir las mismas restricciones que los iconos de la barra de herramientas.

6.3 Descripción de los Mapping Diagrama-Modelo

Para describir cada una de las vistas, se mostrara una instanciación de cada modelo en la que elementos representativos están resaltados con colores. Posteriormente sobre un fragmento del metamodelo de PervML se utilizarán los mismos colores para identificar que clases son representadas en el diagrama.

Como se ha comentado en puntos anteriores, existe una serie de atributos en los modelos de GMF que deben ser personalizados, para evitar el problema de las colisiones entre plug-ins y permitir mostrar elementos de diferentes modelos en un diagrama.
Por ellos para cada una de las vistas se especificara el valor que deben tomar estos atributos.

6.4 Vista Modelo servicios

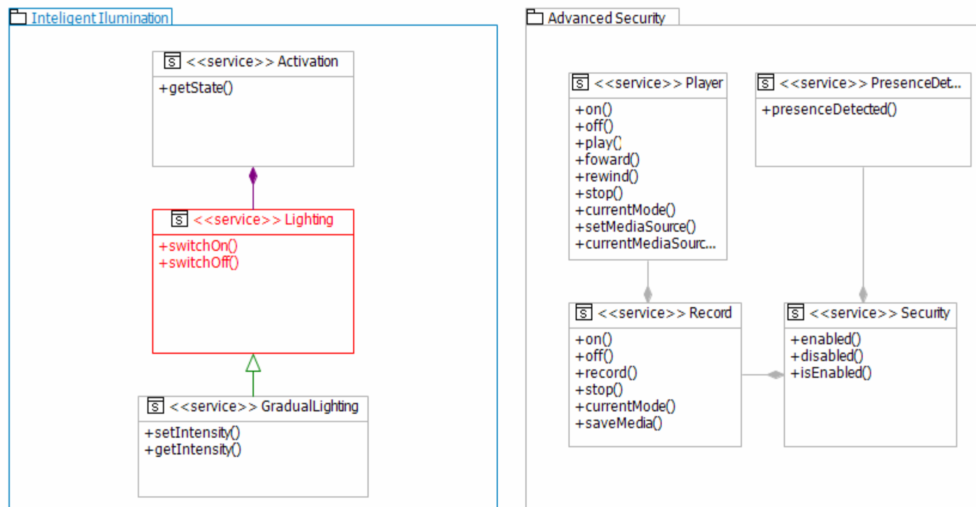


Figura 57: Modelo de servicios.

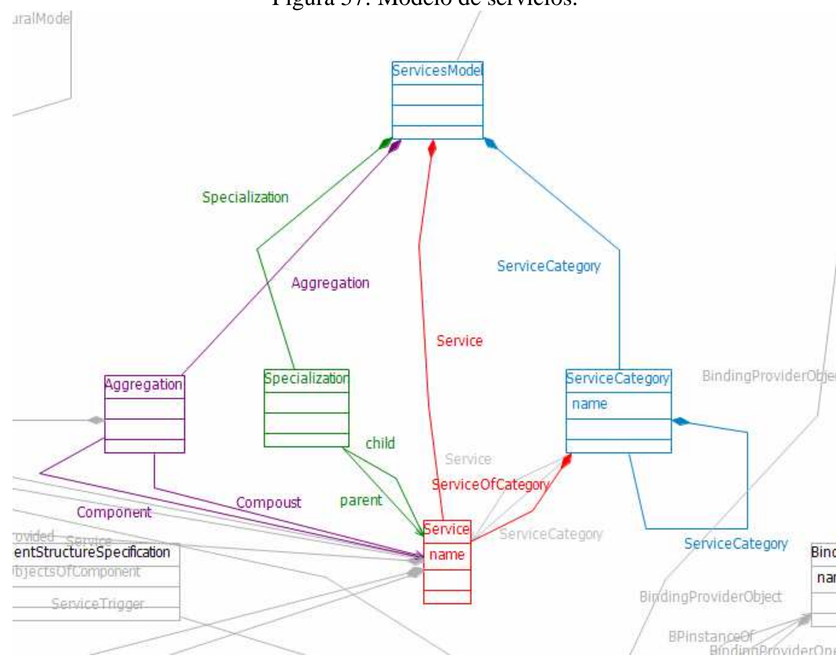


Figura 58: Subconjunto del metamodelo relacionado con el modelo de servicios.

Plug-in	
Nombre:	pervml servicesModel Plugin
Proveedor:	Pervasive
ID plug-in:	PervMLVisualEditorservicesModel.diagram
ID asistente:	servicesModel

Shortcuts	
Proporciona enlaces a:	structualModel
Consume enlaces de:	-

Ficheros	
Extensión del modelo:	servicesModel
Extensión del diagrama:	servicesModel_diagram

6.5 Vista Modelo diagrama transición de estados

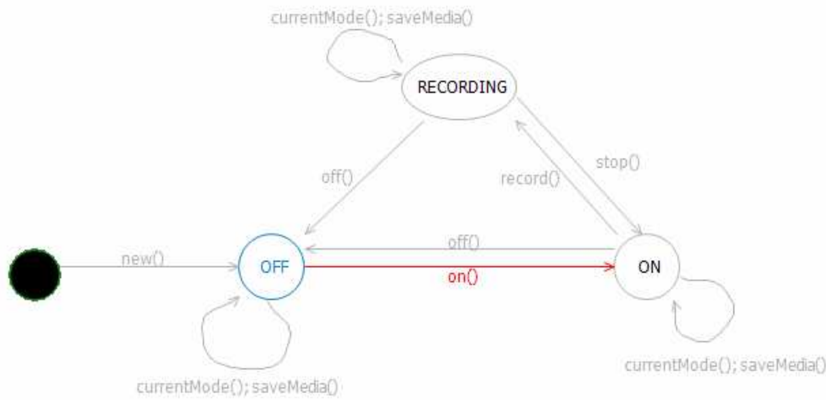


Figura 59: Diagrama de transicion de estados.

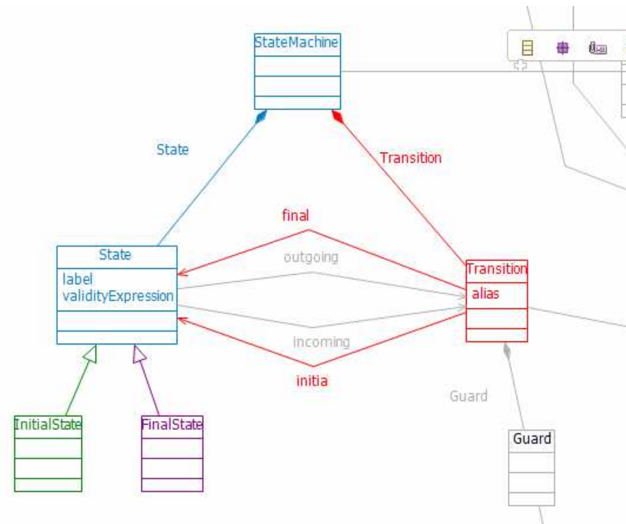


Figura 60: Subconjunto del metamodelo relacionado con los diagramas de transiciones de estados.

Plug-in	
Nombre:	pervml std Plugin
Proveedor:	Pervasive
ID plug-in:	PervMLVisualEditorstd.diagram
ID asistente:	std

Shortcuts	
Proporciona enlaces a:	-
Consume enlaces de:	-

Ficheros	
Extensión del modelo:	std
Extensión del diagrama:	std_diagram

6.6 Vista Modelo componentes

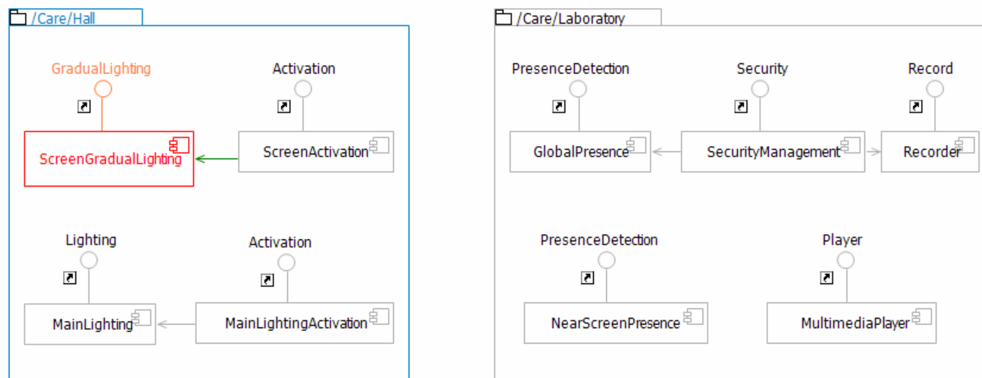


Figura 61: Modelo de componentes.

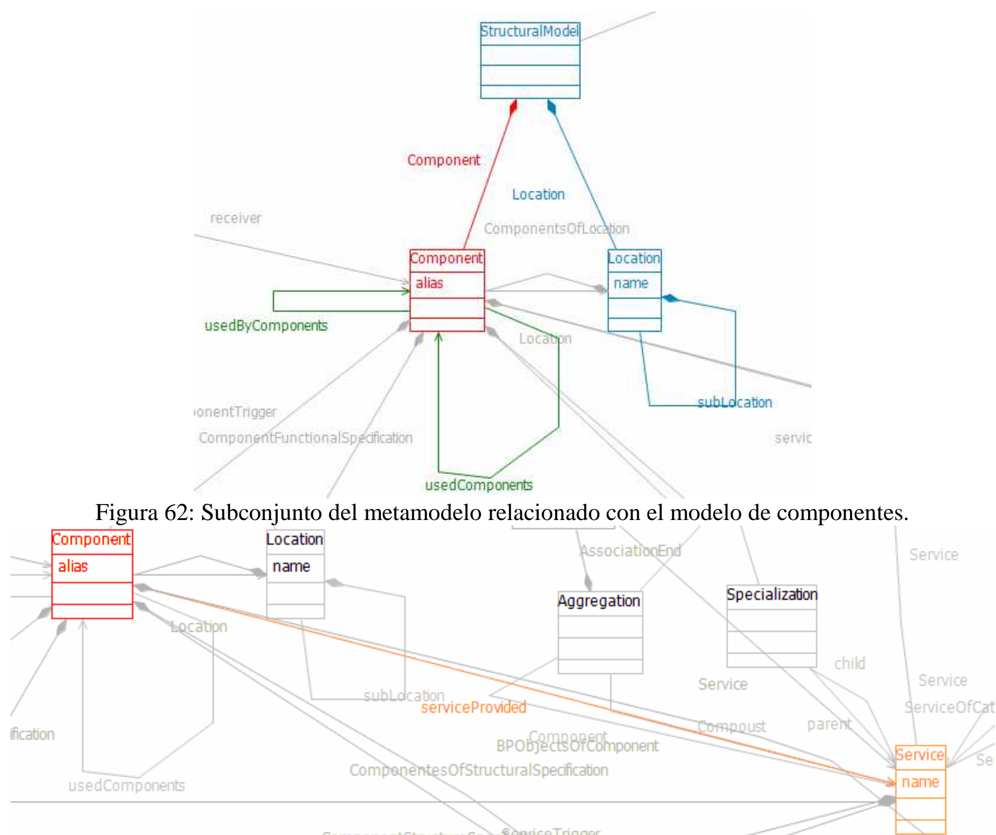


Figura 62: Subconjunto del metamodelo relacionado con el modelo de componentes.

Figura 63: Relaciones en el metamodelo con otros modelos.

Plug-in	
Nombre:	pervml structuralModel Plugin
Proveedor:	Pervasive
ID plug-in:	PervMLVisualEditorstructuralModel.diagram
ID asistente:	structuralModel

Shortcuts	
Proporciona enlaces a:	componentsStructualSpecification
Consume enlaces de:	servicesModel

Ficheros	
Extensión del modelo:	structuralModel
Extensión del diagrama:	structuralModel_diagram

6.7 Vista Modelo Binding providers

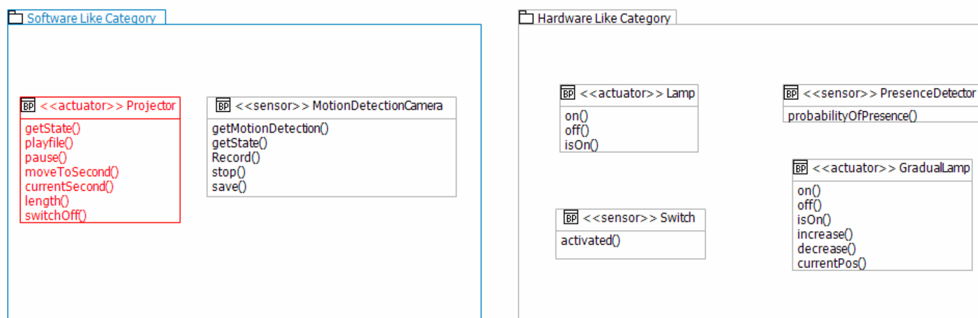


Figura 64: Modelo de binding providers.

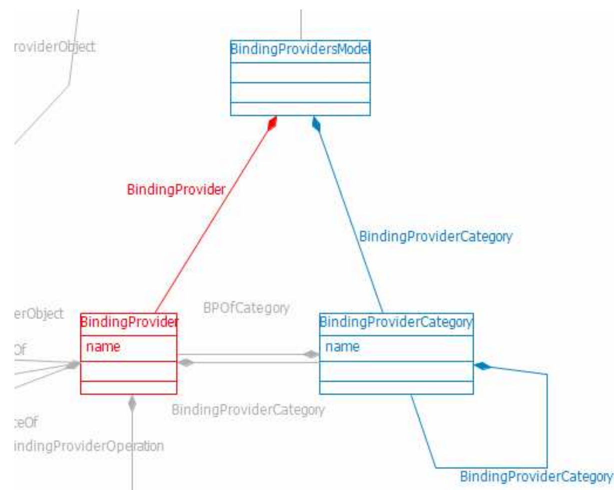


Figura 65: Subconjunto del metamodelo relacionado con el modelo de binding providers.

Plug-in	
Nombre:	pervml bindingProvidersModel Plugin
Proveedor:	Pervasive
ID plug-in:	PervMLVisualEditorbindingProvidersModel.diagram
ID asistente:	bindingProvidersModel

Shortcuts	
Proporciona enlaces a:	componentsStructualSpecification
Consume enlaces de:	-

Ficheros	
Extensión del modelo:	bindingProvidersModel
Extensión del diagrama:	bindingProvidersModel_diagram

6.8 Vista Modelo estructura componentes

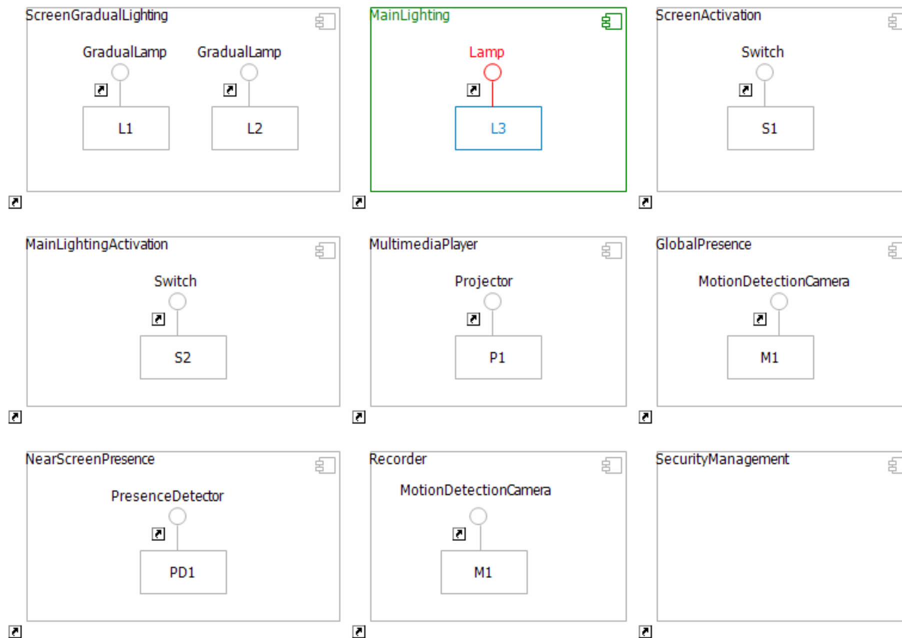


Figura 66: Modelo de estructura de componentes.

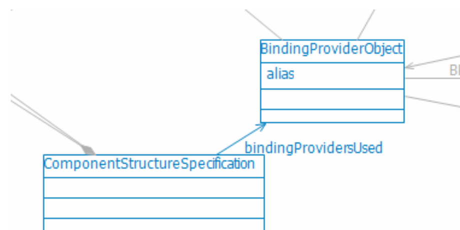


Figura 67: Subconjunto del metamodelo relacionado con el modelo de binding providers.

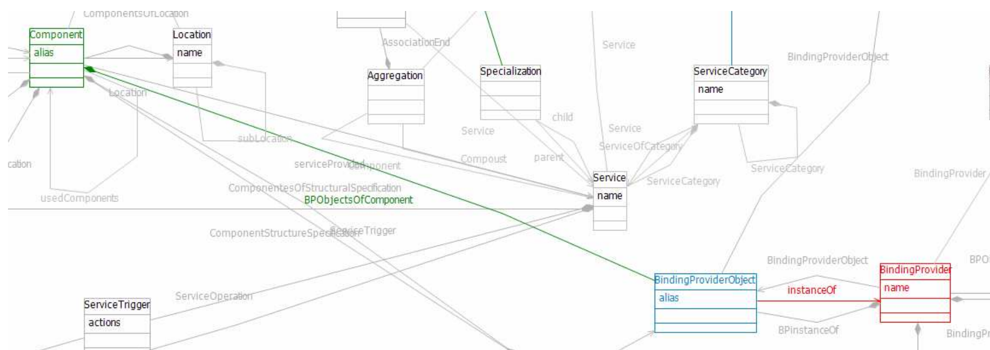


Figura 68: Relaciones en el metamodelo con otros modelos..

Plug-in	
Nombre:	pervml componentsStructualSpecification Plugin
Proveedor:	Pervasive
ID plug-in:	PervMLVisualEditorcomponentsStructualSpecification.diagram
ID asistente:	componentsStructualSpecification

Shortcuts	
Proporciona enlaces a:	-
Consume enlaces de:	bindingProvidersModel, structuralModel

Ficheros	
Extensión del modelo:	componentsStructualSpecification
Extensión del diagrama:	componentsStructualSpecification_diagram

6.8 Restricciones

Como se comento en el capitulo de *Gestión de modelos* las clases java generadas con EMF garantizan que se cumplan las restricciones definidas en el modelo Ecore. Además GMF nos permite definir nuevas restricciones tanto a nivel de modelo como de diagrama. La razón de que se permitan definir restricciones también a nivel de diagrama, se debe a que un diagrama no tiene porque tener necesariamente una relación uno a uno con los elementos del modelo que representa. Apareciendo distintos elementos o relaciones en el diagrama sobra las que se desee exigir ciertas condiciones.
Las restricciones definidas a nivel de modelo Ecore y las definidas desde GMF a nivel de modelo, pese a que obtienen el mismo resultado en el modelo, tiene distintos comportamientos. Mientras las definidas en ecore impiden la realización de la acción, las definidas en GMF permiten realizar la acción en el diagrama y es en la fase de validación cuando advierten de la inconsistencia detectada de acuerdo a la restricción.

Las restricciones en GMF se definen en los diagramas mediante “reglas de auditoria”. Una regla de auditoria consta fundamentalmente de:

- ID.
- Nombre.
- Descripción.
- Elemento sobre el que se aplica.
- Expresión.
- Lenguaje.

Pese a existir la posibilidad de seleccionar el lenguaje en el que se expresan las restricciones en GMF, actualmente solo se encuentra soportado el lenguaje OCL.

Por defecto en el modelo genModel esta desactivada la comprobación de restricciones, por ello si solo se definen las reglas de auditoria estas nunca serán comprobadas. Para activar la comprobación de restricciones se debe establecer el valor de la propiedad “Validation Enabled” a true en el modelo genModel. Adicionalmente es posible indicar al editor grafico que decore con un signo de error el elemento del diagrama que origino una violación de restricción, para ello de nuevo en el modelo genModel se debe estable el valor de “'Validation Decorators” a true.

A continuación se listan las restricciones definidas con GMF, expresandolas tanto en lenguaje natural como con sintaxis de OCL:

- Los nombre de los servicios deben ser únicos:
  context ServicesModel inv: self.Service->isUnique(s:Service | s.name)

- Los nombres de las operaciones de los servicios deben ser únicos dentro de cada servicio:
  context Service inv: self.ServiceOperation->isUnique(op:Operation | op.name)

- Los nombre de los estados del STD deben ser únicos:
  context StateMachine inv: self.State->isUnique(st:State | st.label)

- Los alias de los componentes del modelo estructural deben ser únicos: context StructuralModel inv: self.Component->isUnique(co:Component | co.alias)

- Los nombres de los tipos de binding providers deben ser únicos:
  context BindingProvidersModel inv: self.BindingProvider->isUnique(bp:BindingProvider | bp.name)

- Los nombres de las operaciones de los tipos de binding provider deben ser únicas para cada binding provider:
  context BindingProvider inv: self.BindingProviderOperation ->isUnique(bpo:BindingProviderOperation | bpo.name)

- Los alias de las instancias de binding providers deben ser únicos: context BindingProvidersInstances inv: self.BindingProviderObjects->isUnique(bpo:BindingProviderObject | bpo.alias)

6.9 Unión de las vistas

El fin de los modelos en una herramienta CASE de generación de código es generar el código a partir de ellos. Para ello la herramienta aplicará en un siguiente paso una serie de platillas a un único modelo. Este modelo debe contener de forma completa toda la especificación del sistema pervasivo. Tras la introducción por parte del usuario de los distintos modelos de PervML mediante el editor grafico, se obtienen varios modelos relacionados entre sí pero separados. Con el objetivo de obtener un único modelo, se ha dotado de funcionalidad a la herramienta para generar este modelo integrado a partir de los distintos modelos del lenguaje PervML.

El editor grafico almacena cada uno de los modelos de PervML en dos ficheros distinto.

- *nombreDelModelo:* almacena únicamente la información propia del modelo.
  - Ejemplo: Servicios, atributos, relaciones entre servicios, etc.

- *nombreDelModelo_diagram:* almacena la información relativa a la representación grafícaa de los elementos del modelo.
  - Ejemplo: Posición en el modelo del grafismo que representa un servicio, colores del grafismo, tamaño del grafismo, etc.

En los nombres de los ficheros *nombreDelModelo* debe remplazarse por el nombre del modelo que se encuentra almacenado en los ficheros. Pudiendo ser uno de los siguientes nombres: servicesModel, std, structuralModel, bindingProvidersModel, componentsStructualSpecification o interactionModel.

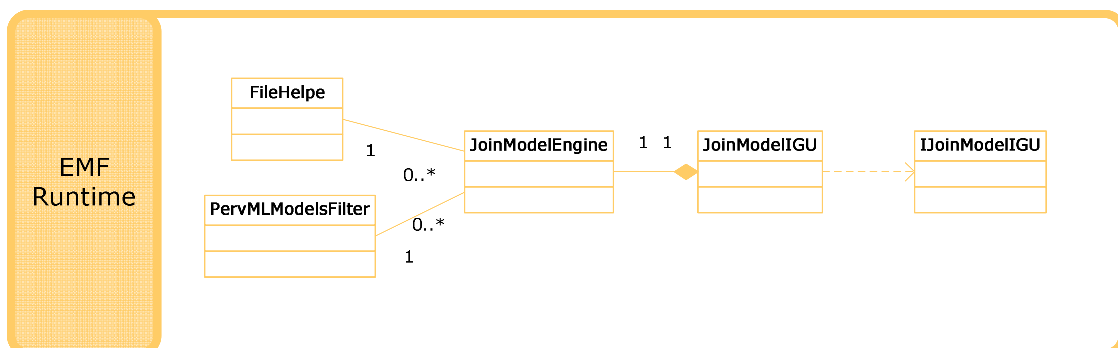


Figura 69: Diagrama de clases de la aplicación JoinModel.

Las clases accesorias FileHelper y PervMLModelsFilter de la figura 69, proporcionan funcionalidad básica relacionada con el tratamiento de ficheros. FileHelper implemeta el método copy que permite copiar ficheros.

```
static void copy(File src, File dst)
```

PervMLModelsFilter implementa la interfaz FilenameFilter. Dando como semántica al metodo accept, que devuelva cierto cuando reciba como argumento un nombre de archivo con una extensión igual a la que se paso en el constructor de la clase, o falso en caso contrario.

```
public boolean accept(File directory, String fileName)
```

La clase JoinModelEngine de la figura 69, es la responsable de materializar los diferentes modelos que forman el lenguaje PervML y fusionarlos en un único modelo.
Un modelo es un objeto raíz, que puede tener relaciones con otros objetos, almacenado en un recurso. Dado que pueden existir diferentes recursos de modelos debe indicarse que implementación se utilizara para (des)serializar los datos.

```
ResourceSet resourceSet = new ResourceSetImpl();
Registry.INSTANCE.put(PervmlPackage.eNS_URI,PervmlPackage.eINSTANCE);
resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().pu
t("pervml", new XMIResourceFactoryImpl());
```

El código muestra como se indica al resourceSet que los modelos que se carguen o almacenen, deben ser interpretados conforme al metamodelo de PervML. Un resourceSet representa una colección de resources en la que se garantiza que ningún resource es cargado más de una vez.

El (des)serializador cuenta con la peculiaridad de que solo es capaz de procesar archivos con una extensión concreta, “pervml”. Por este motivo antes de proceder a materializar los diferentes archivos de las vistas, se debe realizar una copia temporal de estos asignándoles la extensión pervml.

```
File modelsFolder = new File(modelsPath);
            String []modelFiles= modelsFolder.list(new
PervMLModelsFilter(extension));
            if (modelFiles.length>0)
            {
                  String modelFile=modelFiles[0];
                  try{
                        File tempFile=new File(modelsPath+"/"
+extension+ ".pervml");
                        temporaryFiles.add(tempFile);
                        FileHelper.copy(new File(modelsPath+"/"
+modelFile), tempFile);
                  }
```

Para la implementación se hace uso de las clases accesorias, que se encargan de filtrar los archivos que contiene modelos y realizar copias de ellos.

Una vez que los archivos pueden ser procesados por el deserializador se procede a materializar uno a uno los modelos y recuperar su objeto raíz.

```
/*ServicesModel*/
				// Get the URI of the model file.
			joinModelIGU.addVerboseMsg("Procesing: Services Model.");
			fileURI = URI.createFileURI(new File(modelsPath+"/" +
XMIFileNameServiceModel).getAbsolutePath());

		// Demand load the resource for this file.
				resource = resourceSet.getResource(fileURI, true);

				ServicesModel serviceModel = (ServicesModel)
resource.getContents().get(0);
```

Finalmente se crea un objeto pervmlModel utilizando la factoría de objetos de PervML.

```
PervmlFactoryImpl factory = new PervmlFactoryImpl();
PervMLModel pervMLModel= factory.createPervMLModel();
```

A este objeto se le asignan los distintos modelos materializados y como ultimo paso se almacena en un nuevo resource.

```
pervMLModel.setServicesModel(serviceModel);
pervMLModel.setStructuralModel(structuralModel);
pervMLModel.setBindingProvidersModel(bindingProvidersModel);
pervMLModel.setInteractionModel(interactionModel);

resourceSerializar.getContents().add(pervMLModel);
resourceSerializar.save(null);
```

Interfaz gráfica

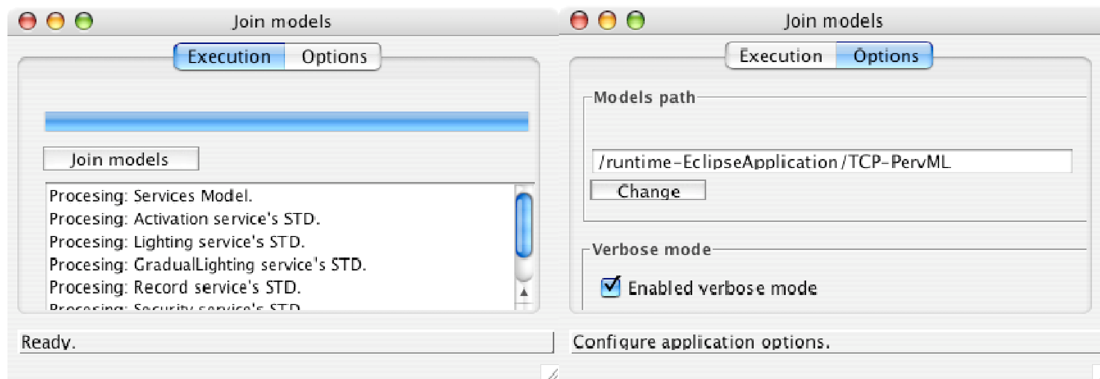


Figura 70: IGU de la aplicación JoinModel.

Parte de la funcionalidad de la interfaz grafica de usuario debe ser accesible desde el engine, la interfaz IJoinModelIGU describe los métodos que debe implementar la interfaz grafica de usuario:

```
public interface IJoinModelIGU {
	public void setProgressPercentage(int percentage);

	public void addVerboseMsg(String msg);
}
```

El método setProgressPercentage es invocado desde el engine para actualizar el valor de la barra de progreso, de acuerdo al porcentaje del proceso realizado. A través del método addVerboseMsg pueden visualizarse en la interfaz de usuario aquellos mensajes que se produzcan en el engine.

El evento clic sobre el boton Join models lanza a ejecución una instancia del engine, con una ruta para los modelos que el usuario seleccionara en la interfaz de usuario.

7 Generación

Tras la introducción del modelado del sistema pervasivo, en el editor grafico de modelos, se debe proceder a la generación del código Java que implemente lo representado en los modelos. Este último paso de generación, debe realizarse de forma completamente automática, sin intervención por parte del usuario.

7.1 Tecnología utilizada

Para la parte de generación de código, se ha utilizado, MOFScript, un lenguaje de transformación de modelos a archivos de texto. Una transformación MOFScript contiene una mezcla entre bloques de texto y lógica de control, para combinar los bloques de texto con información procedente de un modelo.

```
<%package org.pervml.application.bproviders.%> self.name

<%;import org.pervml.framework.ActivatorHelper.BProviderActivator;

public class Activator extends BProviderActivator {

      public void specificStart(){
            String driverPID;
            try{
              %>
self.BindingProviderObject ->forEach(bp:pervml.BindingProviderObject)
{
   <%
      driverPID = "CHANGE_THIS"; // TODO: Change the driver PID
      registerBProviders(driverPID,"BP%> bp.alias
      <%",new BProvider(driverPID,this.context));
            %>
}
<%
            }
      catch(Exception e) {
            }
      }
}
%>
```

En el fragmento de regla anterio, las zona delimitadas por los paréntesis angulares y el porcentaje, <% %>, son bloques de texto que aparecerán sin modificaciones en el archivo de texto generado. Las sentencias no delimitadas por <% %> son las sentencias de control. Su función es doble:

- Determinar que bloques de texto aparecen en el código resultante.
- Resolver valores de variables en el modelo.

El bloque de texto contenido en el recuadro de línea continua, aparece en el código Java generado ya que no depende de ninguna sentencia de control. Por otro lado el bloque de texto delimitado por el cuadrado de línea discontinua, aparecerá tantas veces como iteraciones se produzcan el bucle *forEach* que lo contiene.

Como se ha comentado anteriormente, desde las reglas de transformación se pueden consultar los elementos de un modelo. Para que el motor que aplica las reglas de transformación conozca la estructura del modelo y cuales son lo elementos validos, debe determinarse con una directiva cual es el metamodelo conforme al cual están los elementos del modelo, que se pretende recorrer con las platillas.

```
texttransformation
PervML2OSGiFramework (in pervml:"http:///org/oomethod/pervml.ecore")
```

Reglas

El cuerpo de la transformación está formado por un conjunto de reglas. Las reglas se ejecutan dentro del contexto de un elemento del metamodelo. Pueden tener también un valor de retorno para ser utilizado en expresiones o en otras reglas. Al igual que en las funciones, pueden definirse parámetros, tanto de tipos predefinidos como de tipos pertenecientes al metamodelo.

```
pervml.Location::getLocationString():String
{
  if (  self.superLocation == null )
    result =  "/" + self.name
  else
      result = self.superLocation.getLocationString() + "/"+ self.name
 }
```

La anterior regla se ejecuta en el contexto de un elemento *Location* del metamodelo de PervML. Esto implica que:

- La regla solo podrá ser invocada sobre elementos de la clase Location.
- La palabra reservada self en el cuerpo de la regla se referirá a la instancia de Location sobre la que se invoco.

El resto de la sintaxis de MOFScript que puede ser empleada en las sentencias de control, no se describirá, dada su gran similitud con la sintaxis típica de los lenguajes de programación.

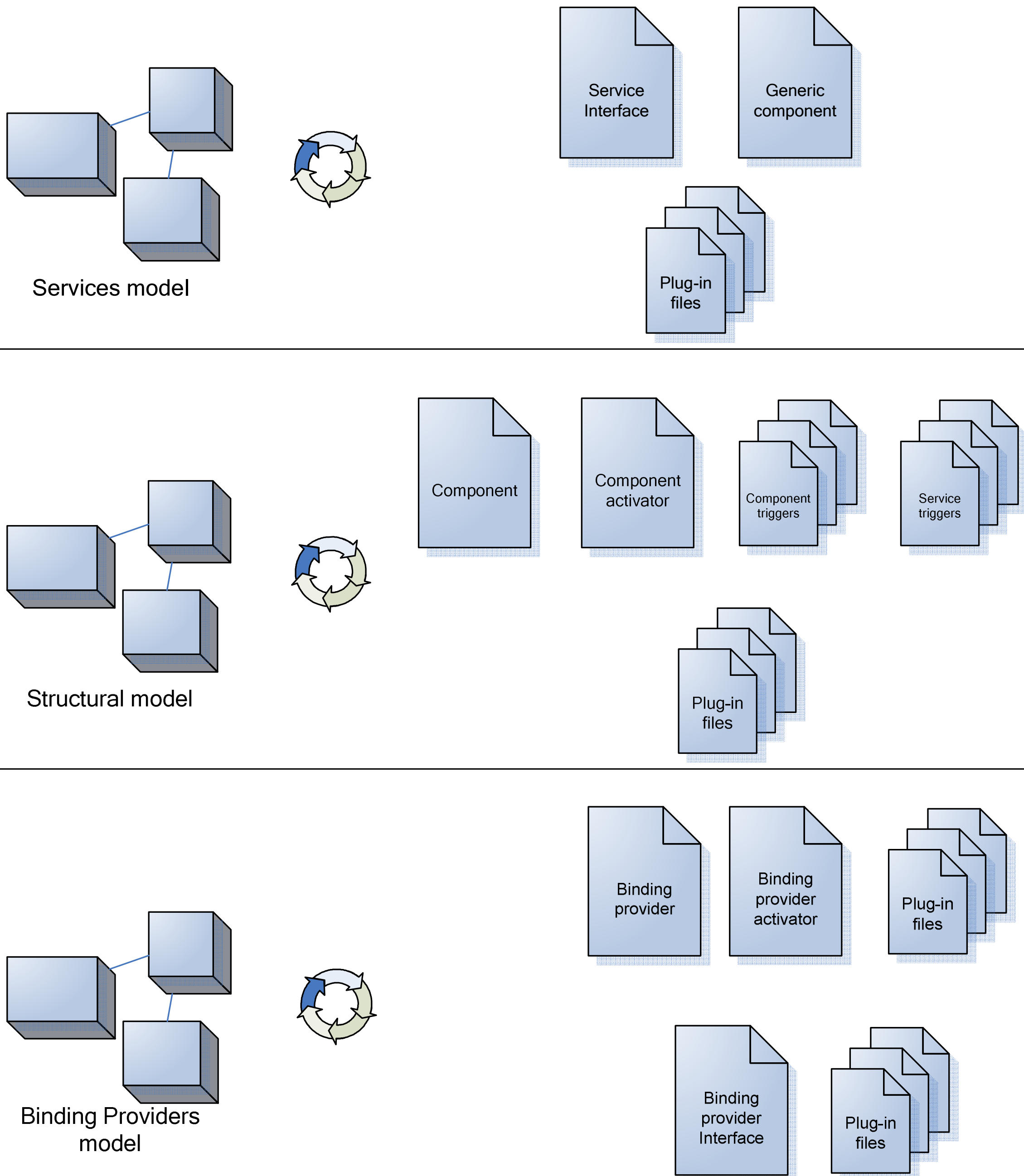
Acceso a elementos del modelo

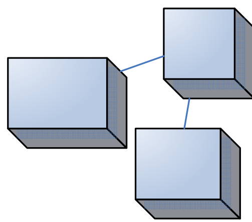
El proceso de generación de código consiste en recorrer a través de las reglas de transformación los elementos del modelo, aplicando sobre estos las reglas definidas. La sintaxis para navegar por los elementos del modelo, tanto en las estructuras de control, como dentro de las reglas es la misma. Se dispone de un objeto self que representa el objeto de contexto sobre el cual se aplica la regla. Para acceder a sus atributos se utiliza el operador de acceso "." seguido del nombre del atributo. La navegación por las relaciones se realiza de forma similar, con el operador de acceso "." seguido esta vez del nombre de la relación. A través de una relación es posible alcanzar una colección de elementos o un único elemento, dependiendo de la cardinalidad. En el caso de recuperar una colección de elementos, se pueden procesar de forma individual por medio del iterador forEach.

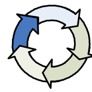
```
self.BindingProviderObject ->forEach(bp:pervml.BindingProviderObject)
```

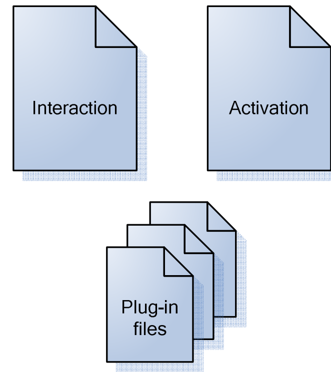
7.2 Ficheros derivados

En este punto se va a especificar que ficheros se generan de cada modelo del lenguaje PervML.









7.3 Punto de entrada a la ejecución

Una regla de Punto de entrada define dónde empieza la ejecución de la trasformación. Es similar al método main en Java. Para definir una regla como punto de entrada se debe indicar el elemento del metamodelo sobre el que se aplicara la reglas, seguido de ::main.

```
pervml.PervMLModel::main()
```

A continuación se muestra la platilla que contiene el punto de entrada, donde se inicia la ejecución de la generación. En esta regla se recorren los elementos del modelo y sobre ellos se aplica una platilla por cada fichero que debe ser generado.

```
/**
 * transformation PervML2OSGiFramework
 *
 *
 */

import "ServiceInterface.m2t"

import "GenericComponent.m2t"
import "GenericComponentResources.m2t"

import "Component.m2t"
import "ComponentActivator.m2t"
import "ComponentResources.m2t"
import "Trigger.m2t"

import "BProvider.m2t"
import "BProviderActivator.m2t"
import "BProviderResources.m2t"

import "BProviderInterface.m2t"
import "BProviderInterfaceResources.m2t"

import "Interaction.m2t"
import "InteractionActivator.m2t"
import "InteractionResources.m2t"

texttransformation PervML2OSGiFramework (in
pervml:"http:///org/oomethod/pervml.ecore") {
```

```
  var firstEntry = true;

  pervml.PervMLModel::main() {

              var counter = 1

              var generateServices = true
              var generateComponents = true
              var generateBProviders = true
              var generateInteractions = true

              if ( generateServices == true ){
                stdout.println("-- Processing Services --")
                self.ServicesModel.ServiceCategory-
>forEach(c:ServiceCategory)
                      {
                         stdout.println("Found Service Category: " +
c.name)
                          c.ServiceOfCategory->forEach(s:Service)
                      {
                         stdout.println("Found Service: " + s.name)
                           s.generateFromService()
                      }

                      }


              }

              if ( generateComponents == true ){
                 stdout.println("-- Processing Components --")
                    self.StructuralModel.Location-
>forEach(l:pervml.Location){
                         stdout.println("Found Components Location: " +
l.name)
                         l.ComponentsOfLocation-
>forEach(c:pervml.Component){
                           stdout.println("Found Component: " + c.alias)
                         c.generateFromComponent()
                      }
                      }
              }

              if ( generateBProviders == true ){
                stdout.println("-- Processing Binding Providers --")
                self.BindingProvidersModel.BindingProviderCategory-
>forEach(bpc:pervml.BindingProviderCategory){
                   stdout.println("Found Binding Providers Category: " +
bpc.name)
                    bpc.BPOfCategory->forEach(b:pervml.BindingProvider){
                       stdout.println("Found Binding Provider: " +
b.name)
                         b.generateFromBindingProvider()
                      }
                }

self.BindingProvidersModel.generateFromBProvidersModel()
              }

              if ( generateInteractions == true ){
                      stdout.println("-- Processing Interactions --")
```

```
                    self.InteractionModel.Interaction-
>forEach(i:pervml.Interaction) {
                         stdout.println("Found Interaction of: " +
i.msgite.Component)
                         i.generateFromInteraction(counter)
                         counter = counter + 1
                    }
              }

  }


   pervml.Service::generateFromService() {

            property packageName = "org.pervml.application.services"
            property packageDir = "src/org/pervml/application/services"
            property baseDir = "Services/"


            /** Generating the Service Interface **/
            file serviceInterfaceFile
(baseDir+"Serv_"+self.name+"/"+packageDir+"/"+self.name+"/Interface.ja
va")
            self.generateServiceInterface()

            /** Generating the Generic Component **/
            file genericComponentFile
(baseDir+"Serv_"+self.name+"/"+packageDir+"/"+self.name+"/GenericCompo
nent.java")
            self.generateGenericComponent()

            /** Generating the Manifest **/
            file componentManifest (baseDir+"Serv_"+self.name+"/meta-
inf/manifest.mf")
            self.generateGenericComponentManifest()

            /** Generating the Eclipse .classpath **/
            file componentManifest
(baseDir+"Serv_"+self.name+"/.classpath")
            self.generateGenericComponentClasspath()
            /** Generating the Eclipse .project **/
            file componentManifest
(baseDir+"Serv_"+self.name+"/.project")
            self.generateGenericComponentProject()

  }

   pervml.Component::generateFromComponent() {

            property packageName = "org.pervml.application.components"
            property packageDir =
"src/org/pervml/application/components"
            property baseDir = "Components/"
            property projectName = "Comp_"+ self.alias
            property projectRoot = baseDir + projectName

            property dir = projectRoot+"/"+packageDir+"/"+ self.alias

            var counter = 1

            /** Generating the Component **/
```

```
            file componentFile (dir+"/Component.java")
            self.generateComponent()

            /** Generating the Component Activator **/
            file componentActivatorFile (dir+"/Activator.java")
            self.generateComponentActivator()

            /** Generating the Component Triggers **/
            self.ComponentTrigger->forEach(trig:pervml.Trigger) {
                  file triggerFile (dir+"/Trigger"+ counter +".java")
                  trig.generateTrigger(counter,self)
                  counter = counter + 1
            }

      //    stdout.println("DEMO-BUG:" +self.serviceProvided.name)
            /** Generating the Service Triggers **/
            self.serviceProvided.ServiceTrigger-
>forEach(trig:pervml.Trigger)  {
                  file triggerFile (dir+"/Trigger"+ counter +".java")
                  trig.generateTrigger(counter,self)
                  counter = counter + 1
            }

            /** Generating the Manifest **/
            file componentManifest (projectRoot+"/meta-
inf/manifest.mf")
            self.generateComponentManifest()

            /** Generating the Eclipse .classpath **/
            file componentManifest (projectRoot+"/.classpath")
            self.generateComponentClasspath()
            /** Generating the Eclipse .project **/
            file componentManifest (projectRoot+"/.project")
            self.generateComponentProject()


  }

   pervml.BindingProvider::generateFromBindingProvider() {

            property packageName = "org.pervml.application.bproviders"
            property packageDir =
"src/org/pervml/application/bproviders"
            property interfacesDir =
"src/org/pervml/bproviders/interfaces"
            property baseDir = "BProviders/"
            property BPInterfacesRoot =
baseDir+"BPInterfaces/"+interfacesDir

            property dir = baseDir+"BP_"+ self.name
+"/"+packageDir+"/"+ self.name
            property BPInterfaces = BPInterfacesRoot + "/"+ self.name


            /** Generating the BindingProvider **/
            file bProviderFile (dir+"/BProvider.java")
            self.generateBProvider()

            /** Generating the BindingProvider Activator**/
            file bProviderActivatorFile (dir+"/Activator.java")
            self.generateBProviderActivator()
```

```
            /** Generating the BindingProvider Project**/
            file bProviderProjectFile (baseDir+"BP_"+
self.name+"/.project")
            self.generateBProviderProject()
            /** Generating the BindingProvider  Classpath**/
            file bProviderClasspathFile (baseDir+"BP_"+
self.name+"/.classpath")
            self.generateBProviderClasspath()
            /** Generating the BindingProvider Manifest**/
            file bProviderManifestFile (baseDir+"BP_"+
self.name+"/meta-inf/manifest.mf")
            self.generateBProviderManifest()

            /** Generating the BindingProvider Interface**/
            file bProviderInterfaceFile
(BPInterfaces+"/Interface.java")
            self.generateBProviderInterface()



   }

   pervml.BindingProvidersModel::generateFromBProvidersModel() {
            property baseDir = "BProviders/"

            /** Generating the BindingProvider Interfaces Project**/
                  file bProviderInterfaceProjectFile
(baseDir+"BPInterfaces/"+"/.project")
                  self.generateBProviderInterfaceProject()
                  /** Generating the BindingProvider Interfaces
Classpath**/
                  file bProviderInterfaceClasspathFile
(baseDir+"BPInterfaces/"+"/.classpath")
                  self.generateBProviderInterfaceClasspath()
                  /** Generating the BindingProvider Interfaces
Manifest**/
                  file bProviderInterfaceManifestFile
(baseDir+"BPInterfaces/"+"/meta-inf/manifest.mf")
                  self.generateBProviderInterfaceManifest()
   }

     pervml.Interaction::generateFromInteraction(counter:Integer) {

            property packageName =
"org.pervml.application.interactions"
            property packageDir =
"src/org/pervml/application/interactions"
            property baseDir = "Interactions/"

            property dir = baseDir+"Int_"+ counter
+"/"+packageDir+"/interaction"+ counter


            /** Generating the Interactions **/
//          file interactionFile (dir+"/Interaction"+counter+".java")
            file interactionFile (dir+"/Interaction.java")
            self.generateInteraction(counter)

            /** Generating the BindingProvider Activator**/
            file interactionActivatorFile (dir+"/Activator.java")
```

```
            self.generateInteractionActivator(counter)

            /** Generating the Interaction Project**/
            file interactionProjectFile (baseDir+"Int_"+ counter
+"/.project")
            self.generateInteractionProject(counter)
            /** Generating the Interaction Classpath**/
            file interactionClasspathFile (baseDir+"Int_"+ counter
+"/.classpath")
            self.generateInteractionClasspath()
            /** Generating the Interaction Manifest**/
            file interactionManifestFile (baseDir+"Int_"+ counter
+"/meta-inf/manifest.mf")
            self.generateInteractionManifest(counter)

   }
```

7.4 Parte común (operaciones)

```
/**
 * transformation SharedRules
 *
 *
 */

texttransformation SharedRules (in
pervml:"http:///org/oomethod/pervml.ecore") {


pervml.Operation::generateOperation(preName:String)
{
      var opReturnType: String
      if (self.returnValue!=null)
      {
        opReturnType=self.returnValue.name
      }
      else
      {
       opReturnType= ""
      }
      print(opReturnType+"  "+preName+self.name+"(")
      self.Argument->forEach(a:Argument){
        print(a.DataType.name+"  "+a.name)
        if ( self.Argument.last().name != a.name ) print(",  ")
      }
      print(")  ")

}

pervml.StateMachine::returnInitialState(): pervml.State {

  self.State->forEach(state:State) {
      state.incoming->forEach(c:pervml.Transition |
c.alias.indexOf("nitial") != -1 ){
            result = state;
      }
  }
}
```

```
pervml.Operation::returnDefaultValue(): String {

  var opDataType = self.returnValue.name

  if ( opDataType == "int" )
      result = "0"
  else if ( opDataType == "boolean" )
      result = "false"
  else
      result = "null"
}

pervml.DataType::returnToObjectPre(): String {
  var dtypeName = self.name

  if ( dtypeName == "int" )
      result = "new Integer("
  else if ( dtypeName == "boolean" )
      result = "new Boolean("
  else
      result = ""
}

pervml.DataType::returnToObjectPost(): String {
  var dtypeName = self.name

  if ( dtypeName == "int" )
      result = ")"
  else if ( dtypeName == "boolean" )
      result = ")"
  else
      result = ""
}

pervml.DataType::returnToBasicPre(): String {
  var dtypeName = self.name

  if ( dtypeName == "int" )
      result = "((Integer)"
  else if ( dtypeName == "boolean" )
      result = "((Boolean)"
  else if ( dtypeName == "String" )
      result = "(String)"
  else
      result = ""
}
pervml.DataType::returnToBasicPost(): String {
  var dtypeName = self.name

  if ( dtypeName == "int" )
      result = ").intValue()"
  else if ( dtypeName == "boolean" )
      result = ").booleanValue()"
  else
      result = ""
}

pervml.StateMachine::transTriggeredBy(state:pervml.State ,
op:pervml.Operation): List {
  var selected:List
```

```
    self.Transition->forEach(tr:pervml.Transition){
    if ( tr.initial.label == state.label and tr.triggerOperation.name
== op.name )
    {
            selected.add(tr)
    }
      result = selected
  }
}

pervml.Location::getLocationString():String {

  if (  self.superLocation == null )
    result =  "/" + self.name
  else
      result = self.superLocation.getLocationString() + "/" +
self.name

}

}
```

7.5 Modelo servicios

```
/**
 * transformation ServiceInterface
 *
 *
 */

import "SharedRules.m2t"

texttransformation ServiceInterface (in
pervml:"http:///org/oomethod/pervml.ecore") {

pervml.Service::generateServiceInterface(){
<% package org.pervml.application.services.%>self.name<%;


public interface Interface {

 %> self.ServiceOperation->forEach(op:Operation){
   <% //Precondition: %>
   if (op.Precondition!=null)
   {
   println("      // "+op.Precondition.expression)
   }
     else
   {
     println("    // true ")
   }
    <% //Postcondition: %>
     if (op.Postcondition!=null)
   {
       println("  // "+op.Postcondition.expression)
   }
   else
   {
     println("    // true ")
```

```
    }
    print("  public ")
    op.generateOperation("")
    println(";\n")
  }<%

}

%>
}

}


/**
 * transformation GenericComponent
 *
 *
 */

import "SharedRules.m2t"


texttransformation GenericComponent (in
pervml:"http:///org/oomethod/pervml.ecore") {

pervml.Service::generateGenericComponent(){

var selectedTrans:List


<%package org.pervml.application.services.%>self.name<%;

import org.pervml.framework.WireHelper.PervMLComponent;
import org.pervml.framework.WireHelper.State;

import org.osgi.framework.BundleContext;
import java.util.HashMap;
import java.util.TreeMap;

public abstract class GenericComponent extends PervMLComponent
implements Interface {

      public GenericComponent(BundleContext c, String componentPid){
                  super(c,componentPid);
      }

      // Initialization
      protected void setKindOfService(){
            this.kindOfService = "%>self.name<%";
      }

      protected void setInitialSTDState(){
            this.STDState="%>
            /* self.StateMachine.returnInitialState().label*/
            // Get de label of the initial satte
      /*    self.StateMachine.State->forEach(state:State) {
                  state.incoming->forEach(c:pervml.Transition |
c.alias.indexOf("nitial") != -1 ){
                        print( state.label )
                  }
```

```
            }*/

             self.StateMachine.Transition-
>forEach(transition:Transition) {

      if(transition.initial.oclIsTypeOf(pervml.InitialState)==true)
                        print(transition.final.label)


             }



             <%";
      }

      protected void initializeSTD(){
             %>self.StateMachine.State->forEach(state:State | not
state.incoming.isEmpty() ) {
             <%this.STD.put("%>state.label<%",new
State_%>state.label<%());
             %>
             }
             <%
      }

      protected void setInitializePersitentVariables(){
        // TODO
      }

      protected void initializeOperationsAndArguments(){
        %> self.ServiceOperation->forEach(op:pervml.Operation){
          if ( op.returnValue.name == "void" or not
op.Argument.isEmpty() ){
            <%HashMap arguments_%>op.name<% = new HashMap();
            %> op.Argument->forEach(arg:pervml.Argument){
            <%arguments_%>op.name<%.put("%>arg.name<%","%>
arg.DataType.name <%");
            %>
                  }

      <%operationsAndArguments.put("%>op.name<%",arguments_%>op.name<%
);

            %>}}<%
      }


      // Service related operations
      %> self.ServiceOperation->forEach(op:pervml.Operation){

      <%protected abstract %>
op.generateOperation("Implementation_")<% ;

      public %> op.generateOperation("")<% {
        %>if ( op.returnValue!=null and op.returnValue.name != "void"
) print(op.returnValue.name+" returnValue;")
        <%

          if(pre_%>op.name<%()){
                try{
```

```
                        disableNotificacions();
                        %>if ( op.returnValue.name != "void" )
print("returnValue = ")<%
                        Implementation_%>op.name<%(%> op.Argument-
>forEach(arg: pervml.Argument){
                             print( arg.name );
                             if ( op.Argument.last() != arg ) print(",
")
                          }
                          <%);
                        if (post_%>op.name<%()){
                             HashMap parameters = null;
                             %>if (op.Argument.size() != 0 ){
                                  println("parameters = new
HashMap();")
                                  op.Argument-
>forEach(arg:pervml.Argument){
                                    println("
      parameters.put(\""+arg.name+"\","+
arg.DataType.returnToObjectPre() +
arg.name+arg.DataType.returnToObjectPost()+");")
                                  }
                             } // if end
                             <%

      this.updateSTDState("%>op.name<%",parameters);
                             }
                  } finally {
                        enableNotifications();
                        %> if ( op.returnValue.name == "void" or not
op.Argument.isEmpty() ){ <%
                        if( changeState() == true )
                                    notifyConsumers();
                        %>}<%
                  }
                  %>if ( op.returnValue.name != "void" ) print("return
returnValue;")
        <%
            }%>if ( op.returnValue.name != "void" ) {
            println("else return "+ op.returnDefaultValue() +";")
            }<%
      }

      public boolean pre_%>op.name<%(){
            // Expression: %> op.Precondition.expression <%
            return true;
      }

      public boolean post_%>op.name<%(){
            // Expression: %> op.Postcondition.expression <%
            return true;
      }

%>    }
<%

    public TreeMap getState() {
            TreeMap hashMapComparable= new TreeMap();
            %> self.ServiceOperation->forEach(op:pervml.Operation){
            if (op.returnValue!=null and  op.returnValue.name != "void"
and op.Argument.isEmpty() ){
```

```
			<%
			hashMapComparable.put("%>op.name<%",%>

		println(op.returnValue.returnToObjectPre()+op.name+"()"+op.retur
nValue.returnToObjectPost()+".toString());")
			} // if end
			} // forEachEnd
			<%
			return hashMapComparable;
		}

%> self.StateMachine.State->forEach(state:State | not
state.incoming.isEmpty()  )  { <%

		private class State_%> state.label <% extends State {

			public State_%> state.label <%(){
				super();
				this.name="%> state.label <%";

				%> self.ServiceOperation-
>forEach(op:pervml.Operation) {
						selectedTrans =
self.StateMachine.transTriggeredBy(state,op)
						if ( not selectedTrans.isEmpty() ){
							println("		DestinationTransition
[] transition_"+op.name+" = {")
							selectedTrans-
>forEach(tr:pervml.Transition) {
							print("\t\t\t new "+tr.alias+"()")
							if ( selectedTrans.last().alias !=
tr.alias )
								print(", ")
							}<%			};

		transitions.put("%>op.name<%",transition_%>op.name<%);

					%>} /* del if not selectedTrans.isEmpty */<%
				%>}<%
			}

			public boolean satisfy() {
				boolean returnValue = true;
				// ValidityExpression: %> state.validityExpression <%
				return returnValue;
			}

		   %> state.outgoing->forEach( trans:pervml.Transition) { <%
			public class %>trans.alias<% extends DestinationTransition
{
				public %>trans.alias<%(){
					super();
					finalState="%> trans.final.label <%";
				}

				public boolean guardCondition(HashMap parameters)
				{
					boolean returnValue = true;

					%> trans.Guard->forEach( g:pervml.Guard ){
					println("// GuardCondition: "+ g.expression )
```

```
                        }<%
                        return returnValue;
                  }
            }
            %>}// end forEach outgoing Transition
            <%
      }
%>} // ForEach State
<%

}

%> // Final
}


}

/**
 * transformation Generic component resources
 *
 *
 */

texttransformation GenericComponentResources (in
pervml:"http:///org/oomethod/pervml.ecore") {


pervml.Service::generateGenericComponentManifest(){
<%Manifest-Version: 1.0
Bundle-Category: PervML_Services
Created-By: PervML2OSGi Generator
Bundle-Name: Generic Component %> self.name <%
Bundle-Vendor: SEAPS Project
Bundle-Version: 1.0.0
Import-Package: org.pervml.framework.WireHelper,
org.osgi.service.wireadmin
Export-Package: org.pervml.application.services.%> self.name <%
%>
}

pervml.Service::generateGenericComponentClasspath(){
<%<?xml version="1.0" encoding="UTF-8"?>
<classpath>
      <classpathentry kind="src" path="src"/>
      <classpathentry kind="con"
path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
      <classpathentry kind="src" path="/PervMLFramework"/>

      <classpathentry kind="output" path="bin"/>
</classpath>%>
}

pervml.Service::generateGenericComponentProject(){
<%<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
      <name>Serv_%> self.name <%</name>
      <comment></comment>
      <projects>
      </projects>
      <buildSpec>
```

```
		<buildCommand>
			<name>org.eclipse.jdt.core.javabuilder</name>
			<arguments>
			</arguments>
		</buildCommand>
	</buildSpec>
	<natures>
		<nature>org.eclipse.jdt.core.javanature</nature>
	</natures>
</projectDescription>%>
}
}
```

7.6 Modelo componentes

```
/**
 * transformation Component
 *
 *
 */

import "SharedRules.m2t"

texttransformation Component (in
pervml:"http:///org/oomethod/pervml.ecore") {


pervml.Component::generateComponent(){

var idCounter:Integer = 1

<%package org.pervml.application.components.%>self.alias<%;


import org.osgi.framework.BundleContext;

// TODO: Include the imports from the required BProviderInterfaces o
GenericComponents

public class Component extends org.pervml.application.services.%>
self.serviceProvided.name   <%.GenericComponent {

	public Component(BundleContext c, String componentPid){
		super(c,componentPid);
%>
		self.ComponentTrigger->forEach(trig:pervml.Trigger) {
		println("\t\t this.listaTriggers.insertar(new
Trigger"+idCounter+"(this));")
		idCounter = idCounter + 1
		}
		self.serviceProvided.ServiceTrigger-
>forEach(trig:pervml.Trigger)  {
		println("\t\t this.listaTriggers.insertar(new
Trigger"+idCounter+"(this));")
		idCounter = idCounter + 1
		}
		<%
	}

	public void initializeProperties(){
```

```
            this.serviceName="%>self.alias<%";
            this.location="JAVI - Modelo localizaciones%>
/*self.Location.getLocationString()*/ <%";
      }
%>

        self.ComponentFunctionalSpecification.Method-
>forEach(me:pervml.Method) {

            print("\t protected ")
            me.ServiceOperation.generateOperation("Implementation_")
            println("{")

            if ( me.ServiceOperation.returnValue.name != "void")
                  println("\t\t"+me.ServiceOperation.returnValue.name+"
returnValue = "+ me.ServiceOperation.returnDefaultValue() +";")

            println("\t\t// "+me.body)
            if ( me.ServiceOperation.returnValue.name != "void")
                  println("\t\t return returnValue;")
            println("\t\t }")
            }
      <%

}
%>
}

}


/**
 * transformation Component Activator
 *
 *
 */

import "SharedRules.m2t"

texttransformation ComponentActivator (in
pervml:"http:///org/oomethod/pervml.ecore") {


  pervml.Component::generateComponentActivator(){


<%package org.pervml.application.components.%>self.alias<%;

import org.pervml.framework.ActivatorHelper.ComponentActivator;

import org.osgi.service.wireadmin.WireAdmin;


public class Activator  extends ComponentActivator {
      protected void initialize(){
            componentPid = "%>self.alias<%";
          implLocation = "JAVI - Modelo
lozalizaciones%>/*self.Location.getLocationString()*/ <%";
            componentInstance       = new
Component(this.context,componentPid);
```

```
	}
	protected  void createAllWires(WireAdmin wa){
	  %>

	  self.BPObjectsOfComponent-
>forEach(bp:pervml.BindingProviderObject){
	    println("\t\t
this.createWire(wa,\"BP"+bp.alias+"\",componentPid);")
	  }
	  self.usedComponents->forEach(comp:pervml.Component) {
	      println("\t\t
this.createWire(wa,\""+comp.alias+"\",componentPid);")
	  }
<%
	}

}
%>
}

}

/**
 * transformation Triggers
 *
 *
 */

import "SharedRules.m2t"

texttransformation Trigger (in
pervml:"http:///org/oomethod/pervml.ecore") {


pervml.Trigger::generateTrigger(id:Integer, comp:Object){


<%package org.pervml.application.components.%>comp.alias<%;

import org.pervml.application.services.%> comp.serviceProvided.name
<%.Interface;

import org.pervml.framework.WireHelper.Trigger;


public class Trigger%>id<% extends Trigger{

	private Interface concreteComponent;

	public Trigger%>id<%(Component newComponent){
		concreteComponent = newComponent;
	}


	protected  boolean getConditions(){

		boolean returnValue = false;

		// Pasarlo a Java
	       // returnValue = %>self.condition<%;
```

```
            return returnValue;
        }

        protected  void doAction(){
              // Pasarlo a Java
              // Del diagrama de secuencia
        }
}
%>
}

}

/**
 * transformation Component resources
 *
 *
 */

texttransformation ComponentResources (in
pervml:"http:///org/oomethod/pervml.ecore") {


pervml.Component::generateComponentManifest(){
<%Manifest-Version: 1.0
Bundle-Category: PervML_Components
Created-By: PervML2OSGi Generator
Bundle-Name: Component %> self.alias <%
Bundle-Vendor: SEAPS Project
Bundle-Version: 1.0.0
Bundle-Activator: org.pervml.application.components.%> self.alias
<%.Activator
Export-Package:  org.pervml.application.components.%> self.alias <%
Import-Package:  org.pervml.framework.WireHelper,
org.pervml.framework.ActivatorHelper,
org.pervml.application.services.%> self.serviceProvided.name <%, %>
self.ComponentStructureSpecification.bindingProvidersUsed-
>forEach(bp:BindingProviderObject){
print("org.pervml.bproviders.interfaces."+bp.instanceOf.name+", "); }
<%org.osgi.service.wireadmin,
org.pervml.framework.ActivatorHelper.list
%>
}

pervml.Component::generateComponentClasspath(){
<%<?xml version="1.0" encoding="UTF-8"?>
<classpath>
      <classpathentry kind="src" path="src"/>
      <classpathentry kind="con"
path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
      <classpathentry kind="src" path="/PervMLFramework"/>
      <classpathentry kind="src" path="/Serv_%>
self.serviceProvided.name  <%"/>
      <classpathentry kind="src" path="/BPInterfaces"/>

      <classpathentry kind="output" path="bin"/>
</classpath>%>
}

pervml.Component::generateComponentProject(){
<%<?xml version="1.0" encoding="UTF-8"?>
```

```
<projectDescription>
      <name>Comp_%> self.alias <%</name>
      <comment></comment>
      <projects>
      </projects>
      <buildSpec>
            <buildCommand>
                  <name>org.eclipse.jdt.core.javabuilder</name>
                  <arguments>
                  </arguments>
            </buildCommand>
      </buildSpec>
      <natures>
            <nature>org.eclipse.jdt.core.javanature</nature>
      </natures>
</projectDescription>%>
}
}
```

7.7 Modelo Binding providers

```
/**
 * transformation Binding provider
 *
 *
 */

import "SharedRules.m2t"

texttransformation BProvider (in
pervml:"http:///org/oomethod/pervml.ecore") {


pervml.BindingProvider::generateBProvider(){
  var counter = 0

<%package org.pervml.application.bproviders.%> self.name <%;

import org.pervml.framework.WireHelper.BindingProvider;
import org.osgi.framework.BundleContext;

public class BProvider extends BindingProvider implements
org.pervml.bproviders.interfaces.%>self.name<%.Interface
 {

      public BProvider(String driverPID, BundleContext c){
            super(driverPID,c);
      }

%>
        self.BindingProviderOperation->forEach(op:pervml.Operation) {
            print("\t public ")
            op.generateOperation("")
            println("{")


            println("\t\t Object [] args = new Object["+
op.Argument.size()+"];")
                  op.Argument->forEach(arg:pervml.Argument) {
```

```
                    println("\t\t args["+counter+"] = "+
arg.DataType.returnToObjectPre()+ arg.name
+arg.DataType.returnToObjectPost()+";");
                    counter = counter + 1
                  }

            if ( op.returnValue.name != "void" )
                  print("\t\t return ");
            else
                  print("\t\t");
            if (op.returnValue!=null)
                  print(op.returnValue.returnToBasicPre())
            print( "runMethod(\"" + op.name +"\",args)" )
            if (op.returnValue!=null)
                  print( op.returnValue.returnToBasicPost() )
            print(";")

            println("\t }")
            }
      <%

}
%>
}

}


/**
 * transformation Binding provider Activator
 *
 *
 */

import "SharedRules.m2t"

texttransformation BProviderActivator (in
pervml:"http:///org/oomethod/pervml.ecore") {


pervml.BindingProvider::generateBProviderActivator(){


<%package org.pervml.application.bproviders.%>self.name<%;

import org.pervml.framework.ActivatorHelper.BProviderActivator;


public class Activator extends BProviderActivator {

      public void specificStart(){

            String driverPID;

            try{
              %> self.BindingProviderObject -
>forEach(bp:pervml.BindingProviderObject) {<%
                  driverPID = "CHANGE_THIS"; // TODO: Change the driver
PID
                    registerBProviders(driverPID,"BP%> bp.alias <%",new
BProvider(driverPID,this.context));
```

```
                  %>}<%
            }
            catch(Exception e) {
            }
      }

}

%>
}

}

/**
 * transformation Binding provider resources
 *
 *
 */

import "SharedRules.m2t"

texttransformation BProviderResources (in
pervml:"http:///org/oomethod/pervml.ecore") {


pervml.BindingProvider::generateBProviderManifest(){
<%Manifest-Version: 1.0
Bundle-Category: PervML_BProviders
Created-By: PervML2OSGi Generator
Bundle-Name: BindingProvider %> self.name <%
Bundle-Vendor: SEAPS Project
Bundle-Version: 1.0.0
Import-Package: org.pervml.bproviders.interfaces.%> self.name <%,
org.osgi.service.wireadmin, org.pervml.framework.ActivatorHelper,
org.pervml.framework.WireHelper
Bundle-Activator: org.pervml.application.bproviders.%> self.name
<%.Activator

%>
}

pervml.BindingProvider::generateBProviderClasspath(){
<%<?xml version="1.0" encoding="UTF-8"?>
<classpath>
      <classpathentry kind="src" path="src"/>
      <classpathentry kind="con"
path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
      <classpathentry kind="src" path="/PervMLFramework"/>
      <classpathentry kind="src" path="/BPInterfaces"/>

      <classpathentry kind="output" path="bin"/>
</classpath>%>
}

pervml.BindingProvider::generateBProviderProject(){
<%<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
      <name>BP_%> self.name <%</name>
      <comment></comment>
      <projects>
      </projects>
```

```
	<buildSpec>
		<buildCommand>
			<name>org.eclipse.jdt.core.javabuilder</name>
			<arguments>
			</arguments>
		</buildCommand>
	</buildSpec>
	<natures>
		<nature>org.eclipse.jdt.core.javanature</nature>
	</natures>
</projectDescription>%>
}

}

/**
 * transformation Binding provider Interface
 *
 *
 */

import "SharedRules.m2t"

texttransformation BProviderInterface (in
pervml:"http:///org/oomethod/pervml.ecore") {

pervml.BindingProvider::generateBProviderInterface(){
<% package org.pervml.bproviders.interfaces.%> self.name <%;


public interface Interface {

 %>  self.BindingProviderOperation->forEach(op:Operation){
   print("  public ")
   op.generateOperation("")
   println(";\n")
 }<%

}

%>
}


}

/**
 * transformation Binding Interface resources
 *
 *
 */

import "SharedRules.m2t"

texttransformation BProviderInterfaceResources (in
pervml:"http:///org/oomethod/pervml.ecore") {


pervml.BindingProvidersModel::generateBProviderInterfaceManifest(){
<%Manifest-Version: 1.0
Bundle-Category: PervML_BProviders
```

```
Created-By: PervML2OSGi Generator
Bundle-Name: BP Interfaces
Bundle-Vendor: SEAPS Project
Bundle-Version: 1.0.0
Export-Package:  %>
      self.BindingProvider->forEach(b:pervml.BindingProvider){
            print(" org.pervml.bproviders.interfaces."+ b.name);
            if ( self.BindingProvider.last().name != b.name ) {
              print(",")
            }
            println("");

      }
}

pervml.BindingProvidersModel::generateBProviderInterfaceClasspath(){
<%<?xml version="1.0" encoding="UTF-8"?>
<classpath>
      <classpathentry kind="src" path="src"/>
      <classpathentry kind="con"
path="org.eclipse.jdt.launching.JRE_CONTAINER"/>

      <classpathentry kind="output" path="bin"/>
</classpath>%>
}

pervml.BindingProvidersModel::generateBProviderInterfaceProject(){
<%<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
      <name>BPInterfaces</name>
      <comment></comment>
      <projects>
      </projects>
      <buildSpec>
            <buildCommand>
                  <name>org.eclipse.jdt.core.javabuilder</name>
                  <arguments>
                  </arguments>
            </buildCommand>
      </buildSpec>
      <natures>
            <nature>org.eclipse.jdt.core.javanature</nature>
      </natures>
</projectDescription>%>
}

}
```

7.8 Modelo Interacción

```
/**
 * transformation Interaction
 *
 *
 */

import "SharedRules.m2t"

texttransformation Interaction (in
pervml:"http:///org/oomethod/pervml.ecore") {
```

```
pervml.Interaction::generateInteraction(id:Integer){
   var component:String =self.msgite.Component

<%package org.pervml.application.interactions.interaction%>id<%;

import org.osgi.framework.BundleContext;

import org.pervml.framework.WireHelper.PervMLInteraction;

import java.util.ArrayList;

//Si algun componente recibe mas de un mensaje, estara repetido
%> self.msgite->forEach(m:pervml.Message) {
println("import org.pervml.application.components."+
m.Component+".Component;")
}<%

public class Interaction extends PervMLInteraction {

      //Faltan componentes
      //Si algun componente recibe mas de un mensaje, estara repetido
%> self.msgite->forEach(m:pervml.Message) {

println("private org.pervml.application.components."+
m.Component+".Component "+ m.Component+";")
}<%

      public Interaction(BundleContext c){
            super(c);
      }

      public boolean checkCondition(){
      //Faltan componentes
      //Si algun componente recibe mï¿½s de un mensaje, estarï¿½
repetido

%> self.msgite->forEach(m:pervml.Message) {
println("\t"+ m.Component+" = (org.pervml.application.components."+
m.Component+".Component)")
println("\t\t
frameworkSearcher.getComponent(org.pervml.application.components."+
m.Component+".Component.class.getName(), \""+ m.Component+"\");")
}<%

          boolean returnValue = false;

            // TODO: Pasar la condicion a Java
          //condition = %> self.condition<%

          return returnValue;
      }

      public void  doActions(){

          //El orden de los parametros puede ser incorrecto
%>

   self.msgite.Parameter->forEach(p:pervml.Parameter)
   {
print("\t"+ component+"."+p.expression+"();")
      /*      if ( param.expression != m.Parameter.last().expression )
```

```
            print(", ")
        println("); ")*/
      }
<%
            //Falta el tratamiento de mensaje blocks, condiciones de
ejecucion y repeticion.

      }
}
%>
}

}


/**
 * transformation Interaction Activator
 *
 *
 */

import "SharedRules.m2t"

texttransformation InteractionActivator (in
pervml:"http:///org/oomethod/pervml.ecore") {


pervml.Interaction::generateInteractionActivator(id:Integer){


<%package org.pervml.application.interactions.interaction%>id<%;

import org.pervml.framework.ActivatorHelper.InteractionActivator;

import org.osgi.service.wireadmin.WireAdmin;


public class Activator extends InteractionActivator {

      protected void initialize(){
            interactionPID = "Interaction%>id<%";
            instance = new Interaction(this.context);
      }

      protected  void createAllWires(WireAdmin wa){

            //Si algun componente recibe mas de un mensaje, estara
repetido
            %> self.msgite->forEach(m:pervml.Message) {
println("this.createInteractionLink(\""+ m.receiver.alias+"\");")
}<%
            // Faltan: (1) los que estan en bloques internos
            //          (2) los que estan en la condician

      }

}
%>
}

}
```

```
/**
 * transformation Interaction resources
 *
 *
 */

import "SharedRules.m2t"

texttransformation Interaction (in
pervml:"http:///org/oomethod/pervml.ecore") {


pervml.Interaction::generateInteraction(id:Integer){
   var component:String =self.msgite.Component

<%package org.pervml.application.interactions.interaction%>id<%;

import org.osgi.framework.BundleContext;

import org.pervml.framework.WireHelper.PervMLInteraction;

import java.util.ArrayList;

//Si algun componente recibe mas de un mensaje, estara repetido
%> self.msgite->forEach(m:pervml.Message) {
println("import org.pervml.application.components."+
m.Component+".Component;")
}<%

public class Interaction extends PervMLInteraction {

      //Faltan componentes
      //Si algun componente recibe mas de un mensaje, estara repetido
%> self.msgite->forEach(m:pervml.Message) {

println("private org.pervml.application.components."+
m.Component+".Component "+ m.Component+";")
}<%

      public Interaction(BundleContext c){
            super(c);
      }

      public boolean checkCondition(){
      //Faltan componentes
      //Si algun componente recibe mï¿½s de un mensaje, estarï¿½
repetido

%> self.msgite->forEach(m:pervml.Message) {
println("\t"+ m.Component+" = (org.pervml.application.components."+
m.Component+".Component)")
println("\t\t
frameworkSearcher.getComponent(org.pervml.application.components."+
m.Component+".Component.class.getName(), \""+ m.Component+"\");")
}<%

          boolean returnValue = false;

            // TODO: Pasar la condicion a Java
```

```
          //condition = %> self.condition<%

          return returnValue;
      }

      public void  doActions(){

          //El orden de los parametros puede ser incorrecto
%>

   self.msgite.Parameter->forEach(p:pervml.Parameter)
   {
print("\t"+ component+"."+p.expression+"();")
      /*      if ( param.expression != m.Parameter.last().expression )
            print(", ")
        println("); ")*/
      }
<%
            //Falta el tratamiento de mensaje blocks, condiciones de
ejecucion y repeticion.

      }
}
%>
}

}
```

8 Caso de estudio

Este capitulo presenta un caso práctico de aplicación de Perv-ML desarrollado con la herramienta de generación automática. En primer lugar se describe brevemente el sistema que se pretende especificar. A continuación se construyen los modelos propuestos por Perv-ML para especificar este sistema.

8.1 Requisitos del Sistema

Se describirá un sistema pervasivo que proporcione servicios en el ámbito de una sala de reuniones. La iluminación de la sala se podrá controlar manualmente mediante algún tipo de interruptor. La sala dispondrá de un reproductor de elementos multimedia (video, audio, etc.). Cuando alguien se acerque al lugar donde aparecen las imágenes, las luces que se encuentren sobre él se atenuarán para permitir una mejor visibilidad. Además, un sistema de seguridad permitirá grabar lo que sucede en la sala si se detecta que hay alguien dentro cuando no debería. Las grabaciones de la cámara podrán ser mostradas en el reproductor de multimedia.

8.2 Identificación de los Servicios del Sistema

A partir de la breve descripción de los requisitos del sistema, se pueden identificar los siguientes servicios:

- **Reproducción de multimedia:** permite reproducir contenidos multimedia, tanto audio como video. También proporciona funcionalidad para controlar la reproducción (avanzar rápido, rebobinar, pausar la reproducción, etc.). En la sala existirá un servicio de este tipo.
- **Iluminación**: proporciona y controla la iluminación de una ubicación. Existirá un servicio de iluminación general de la sala.
- **Iluminación gradual**: además de la funcionalidad de un servicio de Iluminación básico, este servicio permite controlar la intensidad de la iluminación. Un servicio de este tipo estará ubicado junto al reproductor de multimedia.
- **Activación**: permite al usuario activar y desactivar otros servicios del sistema. En nuestro caso, existirán dos servicios de este tipo para controlar los dos servicios de iluminación de la sala.
- **Detección de presencia**: informa sobre la existencia de presencia en una ubicación determinada. Existirán dos servicios de este tipo: uno que indicará si existe presencia en la sala, y otro que indicará si existe presencia cerca del reproductor de multimedia.
- **Grabación de video**: graba en un archivo un video con imágenes que captura. El sistema tendrá un servicio de este tipo que capturará las imágenes de la sala.
- **Seguridad**: es un servicio compuesto que, cuando se encuentra activado, se encarga de iniciar la grabación de video si se detecta presencia en una ubicación. En el sistema existirá un servicio de este tipo que activará el servicio de grabación de video cuando se detecte presencia en la sala.

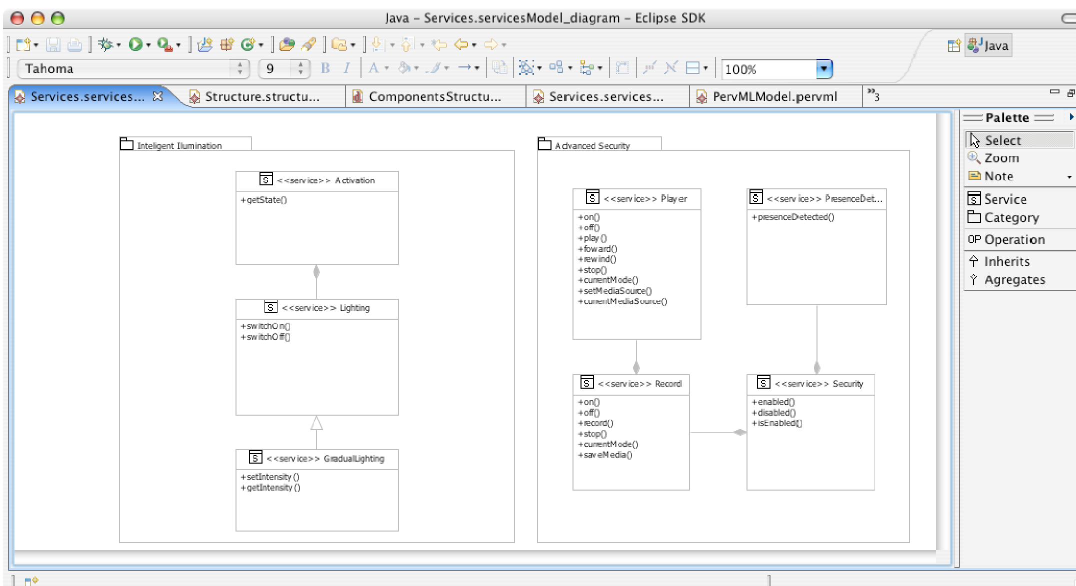
La funcionalidad proporcionada por los servicios descritos hasta ahora puede (o debe) ser controlada por el/los usuario/s mediante algún tipo de interfaz (web, pda, voz, etc.) Por ejemplo, el servicio de grabación de video puede controlarse a petición del usuario, el servicio de seguridad puede activarse o desactivarse, etc. Pero una de las funcionalidades descritas en los requisitos no requiere ni permite participación directa del usuario. Se trata de la *disminución de la intensidad de la iluminación cuando se detecta presencia cerca del reproductor de*

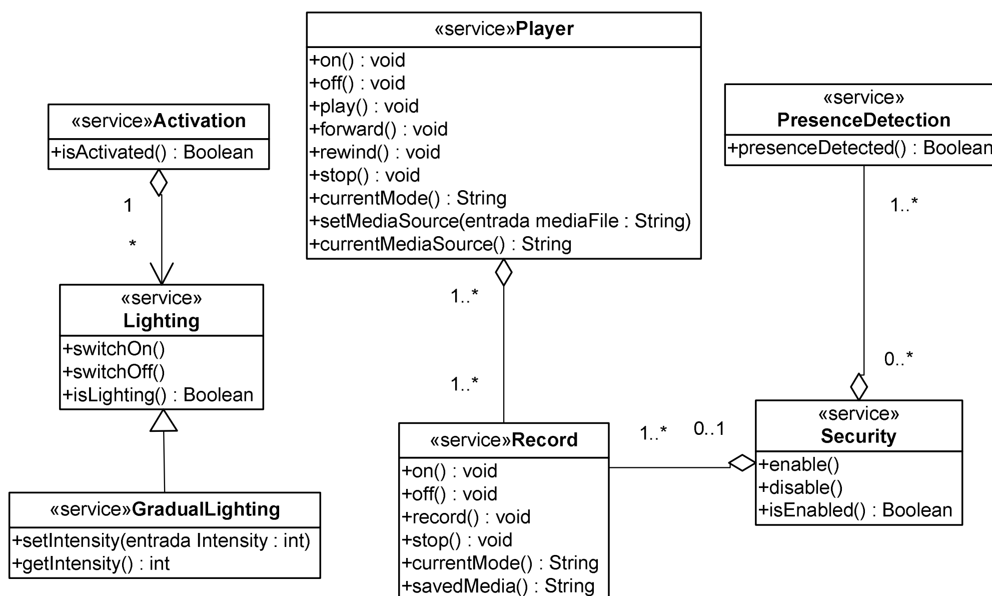
multimedia. Por ello, esta funcionalidad se describe como una **interacción**, en lugar de como un servicio.

8.3 El Modelo de Servicios (Descripción de los Tipos de Servicios)

En el Modelo de Servicios de PervML se describen los distintos tipos de servicios que puede haber en el sistema en desarrollo. Un servicio está compuesto de una serie de operaciones, cada una de ellas puede tener disparadores, pre y post condiciones. Además cada servicio tiene asociada una máquina de transición de estados para describir la secuencia válida de invocación de las operaciones.

A continuación se muestra el Modelo de Servicios utilizando la herramienta de generación de código. Como el objetivo del proyecto es el de producir código fuente y no el de generar documentación a partir de los modelos, no se ha optimizado la presentación de los modelos en documentos. Por ello se presentan también los modelos realizados con una herramienta orientada a producir documentación (Microsoft vision) para favorecer la legibilidad.





Se ha optado por relacionar los servicios de activación (*Activation*) con los servicios de iluminación (*Lighting*). De esta manera, desde un servicio de activación en concreto sólo podrá accederse (para activar o desactivar) a servicios de iluminación. En principio, esto limita la funcionalidad del servicio de activación, ya que así no podría ser utilizado, por ejemplo, para controlar el servicio de seguridad. En realidad, esta posibilidad continúa existiendo mediante el uso de interacciones, donde se puede acceder a cualquiera de los servicios del sistema. Se ha elegido la opción de la relación porque, en este caso de estudio, refleja explícitamente los requisitos del sistema.

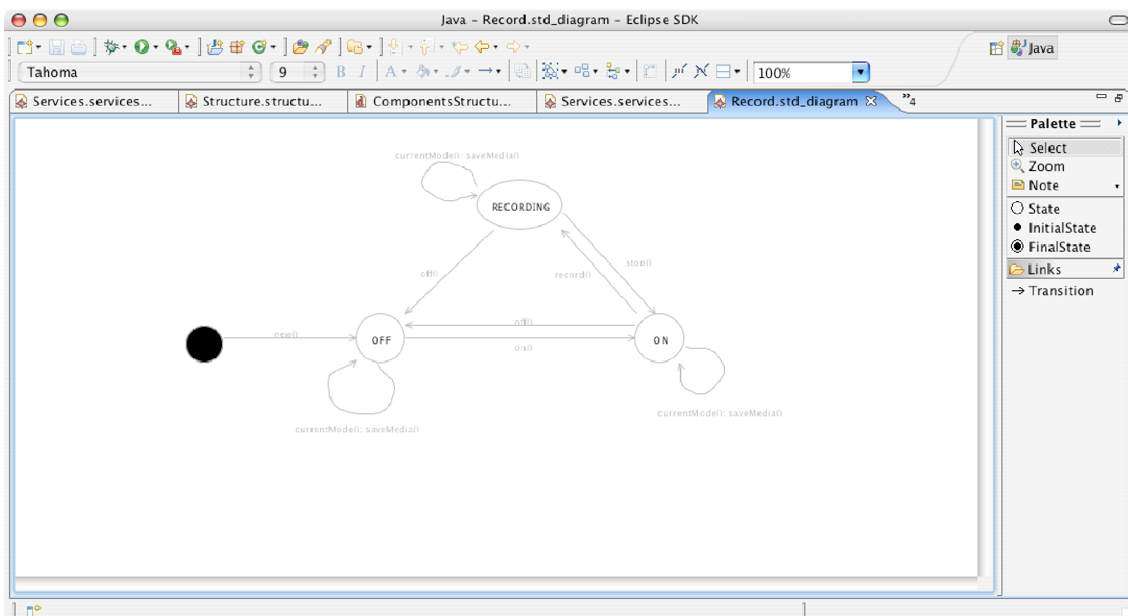
Tipos de Servicios del Sistema

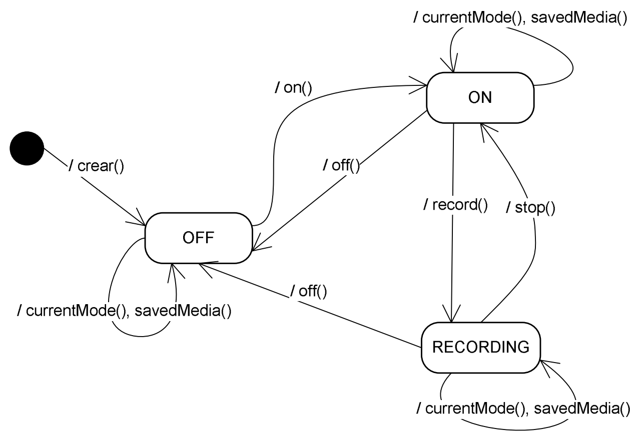
A continuación se describirá cada uno de los tipos de servicios que aparecen en el modelo de servicios. Para ello se describirá textualmente su funcionamiento, se mostrará su diagrama de transición de estados y se especificarán las pre y post condiciones de cada operación.

Record

Mediante las operaciones "on()" y "off()" el servicio se activa y desactiva respectivamente. Cuando el servicio está activado es posible iniciar la grabación ("record()") y detenerla ("stop()"). La operación "currentMode()" devuelve la situación actual del servicio, que puede ser: apagado ("off"), activado pero detenido ("on") o grabando ("recording"). La operación "savedMedia()" devuelve la ruta del archivo multimedia sobre el que se está realizando la grabación.

Se presentan el modelo del diagrama de transición de estados utilizando la herramienta de generación de código y como se ha comentado antes se acompaña del mismo modelo creado con una herramienta orientada a producir documentación. En los sucesivos diagramas de transición de estados únicamente se mostrara el modelo creado específicamente para la documentación.





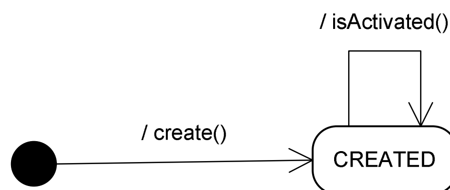
Operación	Pre	Post
on()	true	currentMode () = 'ON'
off()	true	currentMode () = 'OFF'
record()	true	currentMode () = 'RECORDING'
stop()	true	currentMode () = 'ON'
currentMode()	true	true
savedMedia()	true	true

Fórmulas de los Estados

Estado	Fórmula
ON	currentMode() = "ON"
OFF	currentMode() = "OFF"
RECORDING	currentMode() = "RECORDING"

Activation

La operación "isActivated()" devuelve la situación actual del servicio, es decir, si el usuario lo ha activado (true) o desactivado (false).



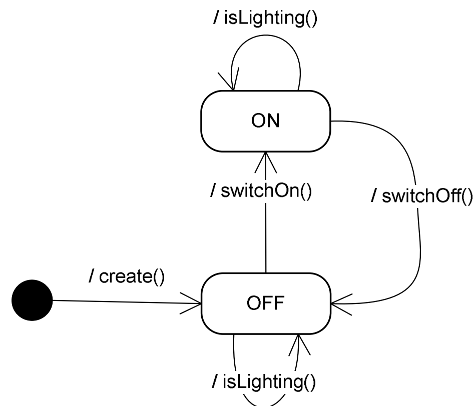
Operación	Pre	Post
getState()	true	true

Fórmulas de los Estados

Estado	Fórmula
CREATED	true

Lighting

Las operaciones "switchOn()" y "switchOff()" encienden y apagan respectivamente el servicio de iluminación. La operación "isLighting()" devuelve el estado actual del servicio (true si está encendido y false si está apagado).



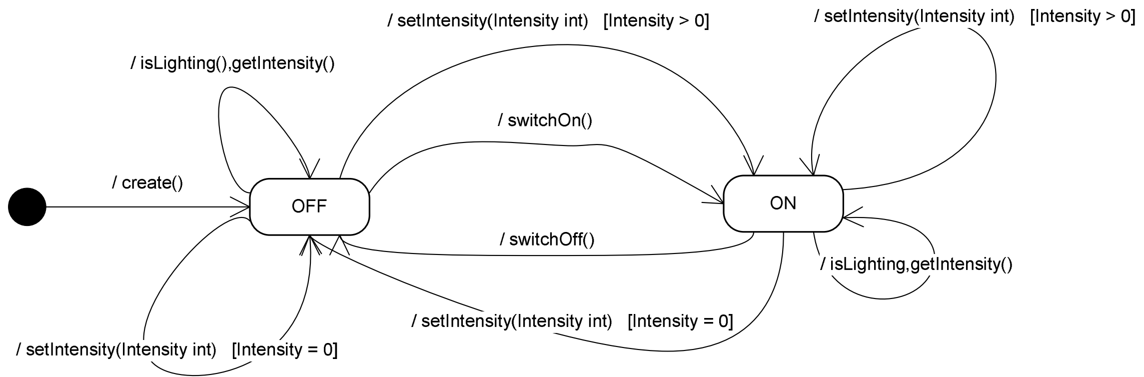
Operación	Pre	Post
switchOn()	true	isLighting() = true
switchOff()	true	isLighting() = false
isLighting()	true	true

Fórmulas de los Estados

Estado	Fórmula
ON	isLighting() = true
OFF	isLighting() = false

GradualLighting

Además de las operaciones de encendido y apagado heredadas de Lighting, este servicio permite fijar (y consultar) un porcentaje de intensidad lumínica. Tal y como se indica en las post-condiciones, encender el servicio utilizando "switchOn" equivale a fijar la intensidad al 100%



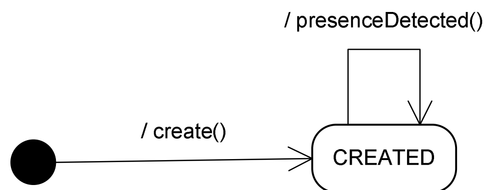
Operación	Pre	Post
switchOn()	true	(isLighting() = true) and (getIntensity() = 100)
switchOff()	true	(isLighting() = false) and (getIntensity() = 0)
isLighting()	true	true
setIntensity(Intensity)	true	getIntensity() = Intensity
getIntensity()	true	true

Fórmulas de los Estados

Estado	Fórmula
ON	isLighting() = true
OFF	isLighting() = false

PresenceDetection

La operación "presenceDetected()" devuelve un valor booleano que indica si el servicio detecta presencia (true) o si no la detecta (false).



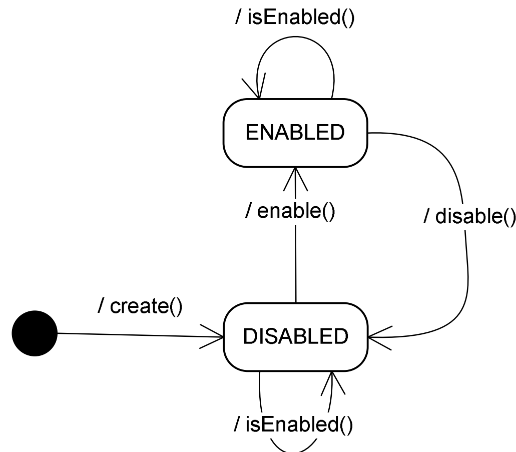
Operación	Pre	Post
presenceDetected()	true	true

Fórmulas de los Estados

Estado	Fórmula
CREATED	true

Security

Las operaciones "enable()" y "disable()" activan y desactivan respectivamente este servicio. La operación "isEnabled()" devuelve el estado actual del servicio (true si está activado y false si está desactivado).



Operación	Pre	Post
`enable()`	`true`	`isEnabled() = true`
`disable()`	`true`	`isEnabled() = false`
`isEnabled()`	`true`	`true`

Fórmulas de los Estados

Estado	Fórmula
`ENABLED`	`isEnabled() = true`
`DISABLED`	`isEnabled() = false`

Triggers

when PresenceDetector->exists(p | p.presenceDetected() = true) and isEnabled() = true **do**

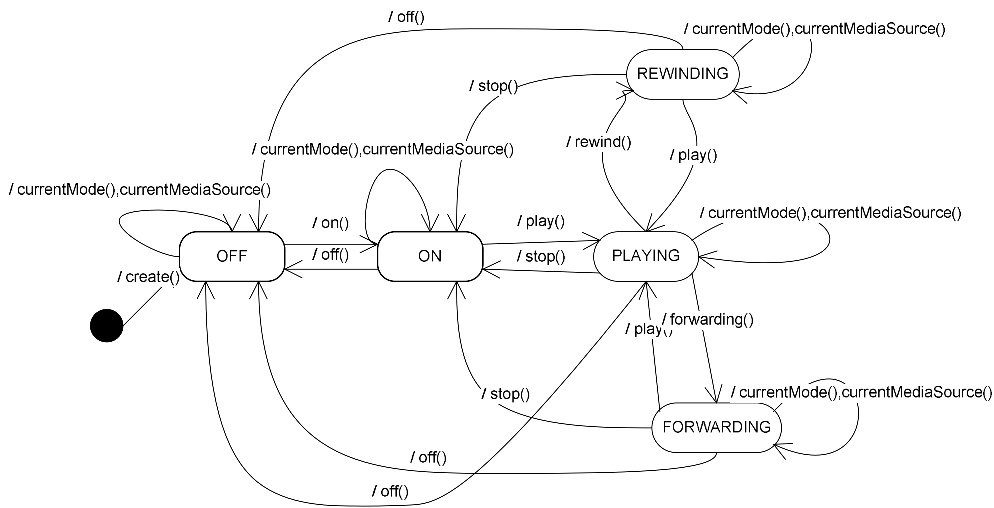
getRecord()->forAll(r | r.off() and r.on() and r.record())

when PresenceDetector->forAll(p | p.presenceDetected() = false) or isEnabled() = false **do**

getRecord()->forAll(r | if r.currentModel() = 'recording" then r.stop())

Player

Mediante las operaciones "on()" y "off()" el servicio se activa y desactiva respectivamente. Cuando el servicio está activado es posible iniciar la reproducción ("play()") y detenerla ("stop()"). Durante la reproducción de un archivo multimedia es posible acelerar la reproducción ("forward()") o rebobinarla ("rewind()"). La operación "currentMode()" devuelve la situación actual del servicio, que puede ser: apagado ("off"), activado pero detenido ("on") o reproduciendo ("playing"), avanzando ("forwarding") o rebobinando ("rewinding"). La operación "setMediaSource(mediaFile: String)" fija la ruta del archivo multimedia que se va a reproducir. Mediante la operación "currentMediaSource()" se puede consultar este valor.



Operación	Pre	Post
on()	true	currentMode () = ‘ON’
off()	true	currentMode () = ‘OFF’
play()	currentMediaSource() <> ‘ ’	currentMode () = ‘PLAYING’
forward()	true	currentMode () = ‘FORWARDING’
rewind()	true	currentMode () = ‘REWINDING’
stop()	true	currentMode () = ‘ON’
currentMode()	true	true
setMediaSource(mediaFile:String)	currentMode () = ‘ON’	currentMediaSource() = mediaFile
currentMediaSource()	true	true

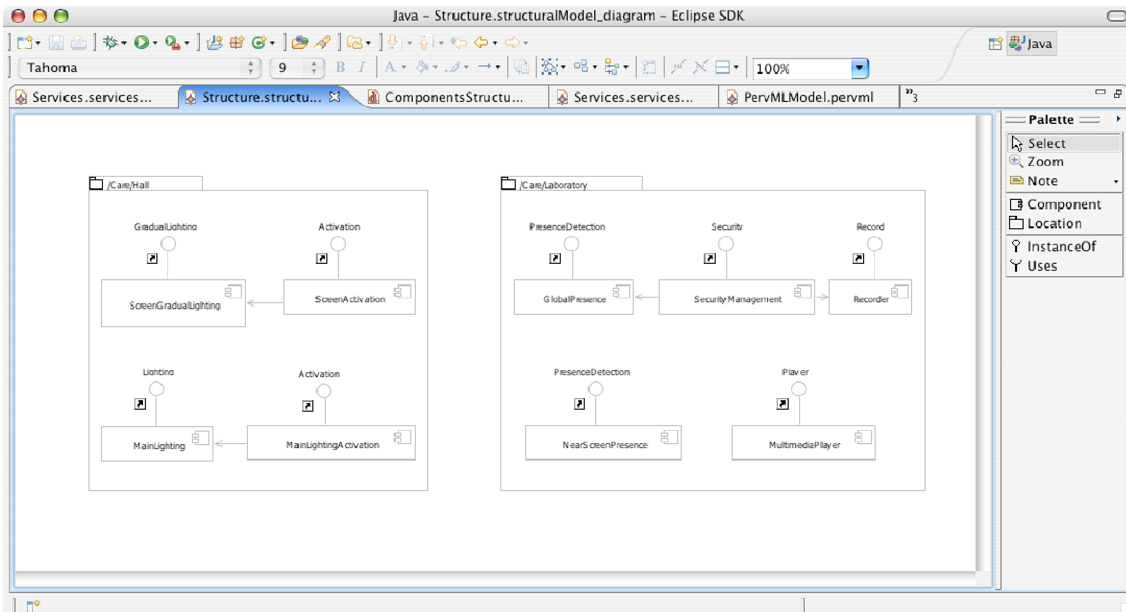
Fórmulas de los Estados

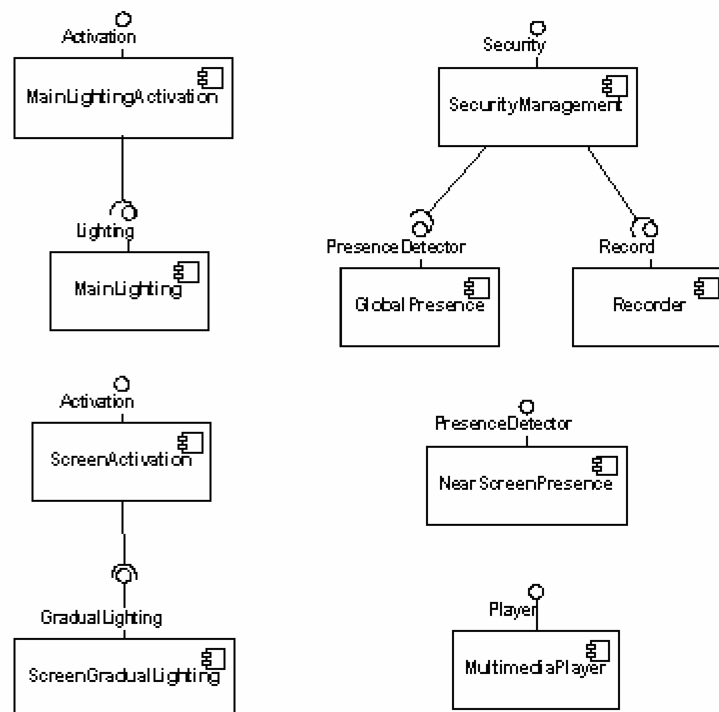
Estado	Fórmula
ON	currentMode() = “ON”
OFF	currentMode() = “OFF”
PLAYING	currentMode() = “PLAYING”
FORWARDING	currentMode() = “FORWARDING”
REWINDING	currentMode() = “REWINDING”

8.4 El Modelo Estructural (Los Servicios del Sistema)

En el Modelo Estructural de PervML se especifican los distintos servicios que existirán en el sistema. Cada servicio está representado por un componente de UML 2.0. El servicio que proporciona se representa mediante una interfaz. Se especifican los servicios de cada ubicación del sistema en paquetes distintos.

Se presentan el modelo de estructuras utilizando la herramienta de generación de código y como se ha comentado antes se acompaña del mismo modelo creado con una herramienta orientada a producir documentación.





Además, sobre estos componentes se han definido los siguientes triggers:

MainLightingActivation

when isActivated() = true **do**

MainLighting.switchOn()

when isActivated() = false **do**

MainLighting.switchOff()

ScreenActivation

when isActivated() = true **do**

ScreenGradualLighting.switchOn()

when sActivated() = false **do**

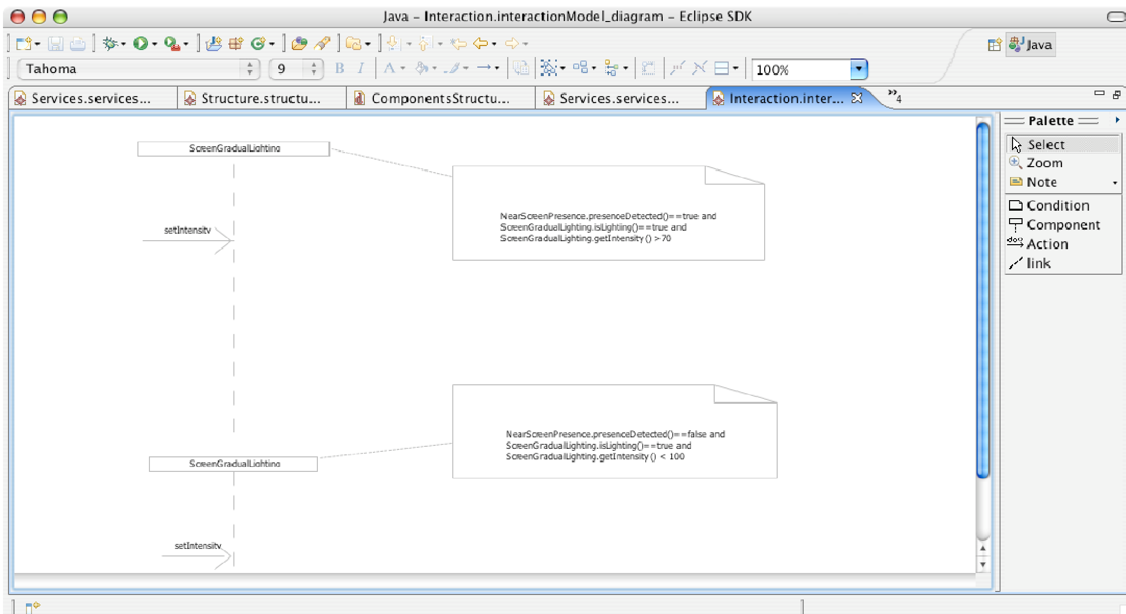
ScreenGradualLighting.switchOff()

Estos disparadores se han asociados a los componentes (en lugar de al tipo de servicio "Activation"), en previsión de que, en un futuro, los servicios de activación puedan utilizarse para controlar otro tipo de servicios que no sean de iluminación.

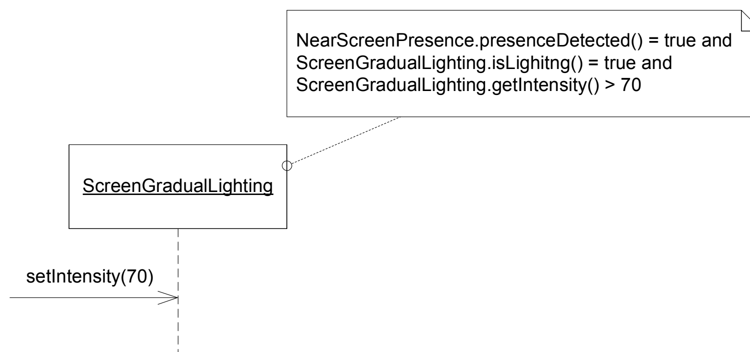
8.5 El Modelo de Interacción

En el Modelo de Interacción de PervML se especifica la funcionalidad del sistema resultante de combinar varios servicios y que se realizará a partir de ciertas situaciones o condiciones del sistema y no a partir de una invocación del usuario. El Modelo de Interacción está compuesto por interacciones, en la que se describe la condición que inicia la interacción y las acciones que se realizarán cuando la interacción tenga lugar.

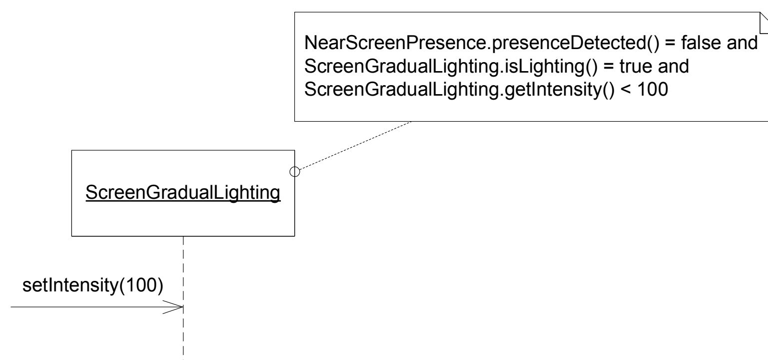
Se presentan el modelo de interacción utilizando la herramienta de generación de código y como se ha comentado antes se acompaña del mismo modelo creado con una herramienta orientada a producir documentación con el fin de facilitar la legibilidad de los modelos



En nuestro sistema hay dos interacciones. La primera de ellas se encarga de fijar la intensidad de la iluminación gradual a una intensidad del 70%. Esto ocurre cuando: se detecta presencia cerca del servicio de reproducción, la iluminación se encuentra encendida y la intensidad es mayor del 70%.



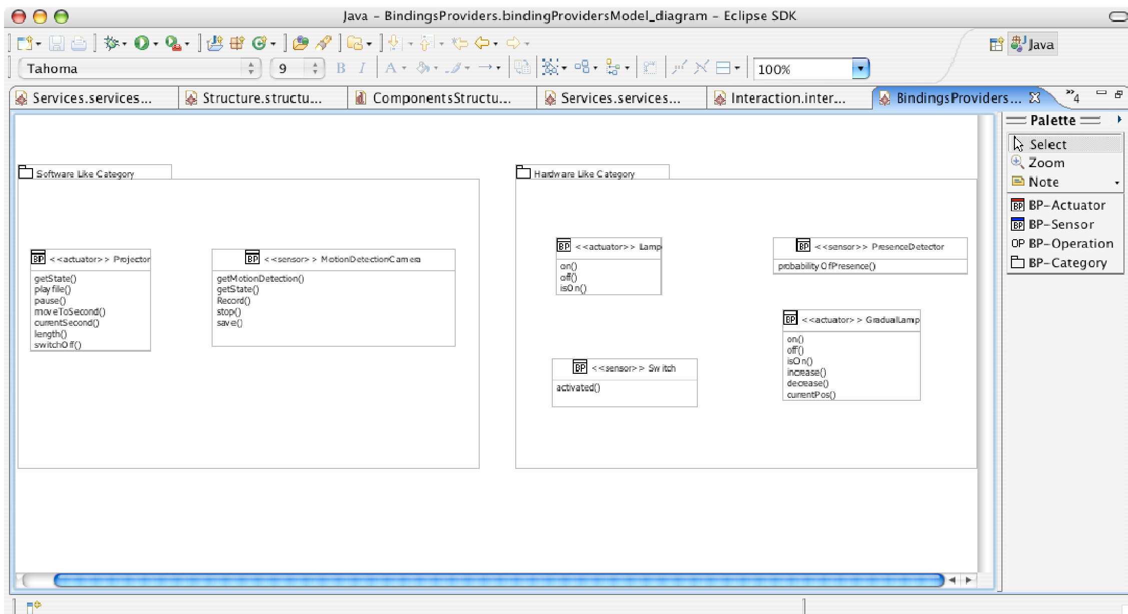
La segunda de las interacciones se encarga de fijar la intensidad al máximo (100%) cuando ya no hay nadie cerca de la pantalla.

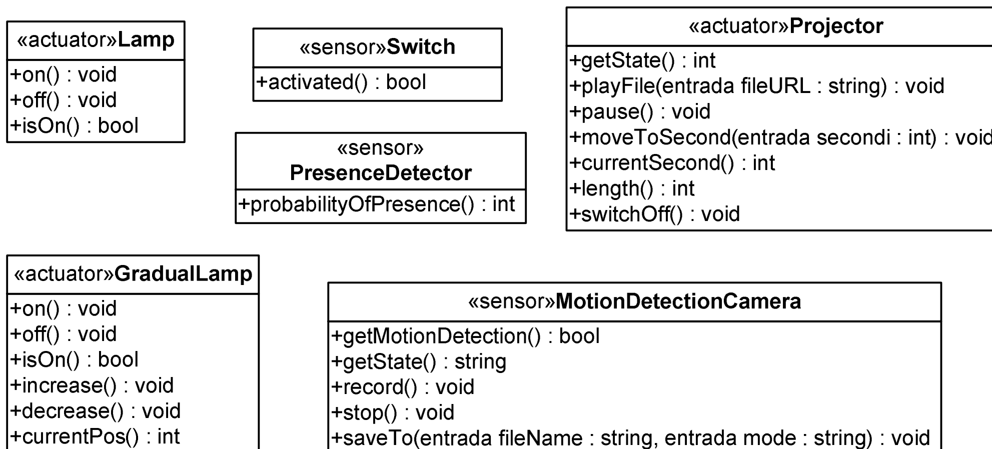


8.6 El Modelo de Proveedores de Enlace

Con los anteriores modelos se han especificado los requisitos del sistema utilizando primitivas de alto nivel de abstracción, como servicio e interacción. Para poder implementar la funcionalidad requerida es necesario utilizar ciertos dispositivos o sistemas software externos. En el Modelo de Proveedores de Enlace se describen las interfaces de estos elementos que permitirán al sistema comunicarse con su entorno.

Se presentan el modelo de proveedres de enlace utilizando la herramienta de generación de código y como se ha comentado antes se acompaña del mismo modelo creado con una herramienta orientada a producir documentación con el fin de facilitar la legibilidad de los modelos





A continuación se describe cada uno de los tipos de dispositivitos que se utilizarán para implementar la funcionalidad del sistema.

Lamp (Lámpara o Bombilla)

Se trata de una única bombilla o de una lámpara (quizá con múltiples fuentes de luz pero que funcionan como un todo).

- **on()**: enciende la lámpara.
- **off()**: apaga la lámpara.
- **isOn()**: Si la lámpara está encendida devuelve true y si está apagada devuelve false.

GradualLamp (Lámpara gradual)

Se trata de una lámpara de intensidad graduable con 7 posiciones, o de una lámpara con 7 bombillas.

- **on()**: enciende la lámpara a la máxima intensidad.
- **off()**: apaga la lámpara.
- **isOn()**: Si la lámpara está apagada devuelve false, en caso contrario devuelve true.
- **increase():** aumenta, si es posible, una posición la intensidad de la lámpara.
- **decrease():** disminuye, si es posible, una posición la intensidad de la lámpara.
- **currentPos()**: devuelve la posición de la intensidad de la lámpara como un número del 0 al 7, donde 0 significa que la lámpara está apagada y 7 que está encendida a la máxima intensidad.

Switch (Interruptor)

Se trata de un interruptor que puede estar encendido o apagado.

- **activated()**: Si el interruptor está pulsado devuelve cierto, si no lo está devuelve falso.

PresenceDetector (Detector de presencia)

Se trata de un dispositivo que detecta presencia en una zona determinada.

- **probabilityOfPresence()**: Devuelve un entero de 0 a 100 que indica la probabilidad de que haya alguien o algo en la zona que monitoriza.

Projector (Proyector)

Se trata de un dispositivo que permite reproducir y proyectar archivos multimedia.

- **getState()**: Devuelve un entero con el estado actual de funcionamiento. Los valores significan: 0 = reproduciendo, 1 = pausa, 2 = error.
- **playFile(fileURL : string)**: Inicia el dispositivo, si está apagado, y reproduce desde el inicio el archivo que se encuentra en “fileURL”.
- **pause()**: Detiene la reproducción de un archivo congelando la imagen.
- **length()**: Devuelve el número de segundos que dura el documento multimedia que se está reproduciendo.
- **currentSecond()**: Devuelve el segundo en el que se encuentra el documento que se está reproduciendo.
- **moveToSecond(second : int)**: Mueve el momento de reproducción al segundo indicado en la variable “second”. Si el valor es mayor que la longitud del archivo, se detiene la reproducción mostrando una imagen en negro.

- **switchOff()**: Apaga el dispositivo, el cual deja de emitir una imagen.

MotionDetectionCamera (Cámara con detección de presencia)

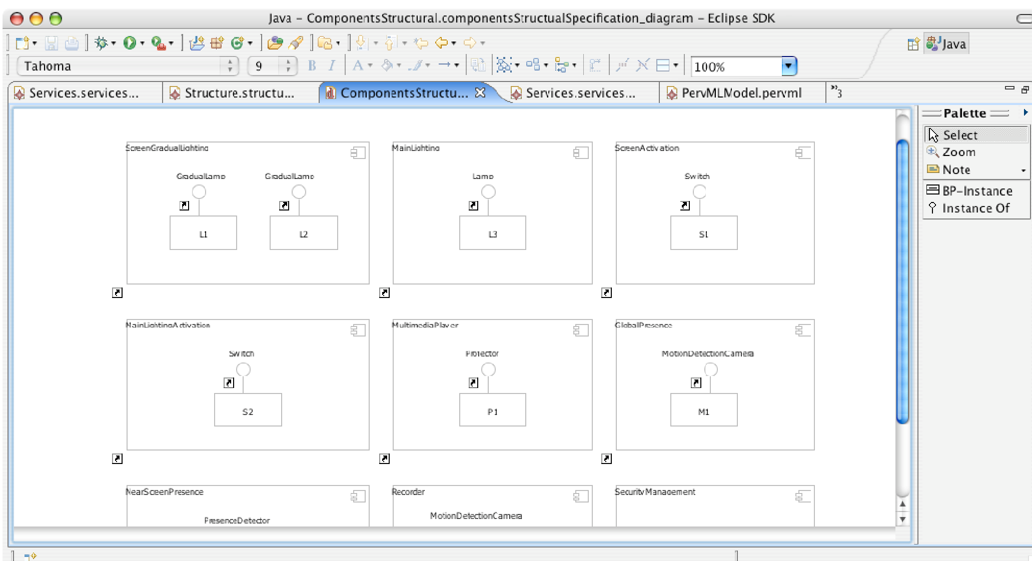
Se trata de una cámara de video capaz de indicar si hay presencia a través del análisis de las imágenes que capta.

- **getMotionDetection()**: Devuelve un valor que indica si detecta presencia (true) o no (falso).
- **getState()**: Devuelve una cadena de texto con el estado en que se encuentra el dispositivo. Los valores son: 'RECORDING' y 'STOPPED'.
- **saveTo(fileName : string, mode : string)**: Configura el destino de la grabación de video. El primer parámetro indica la ruta del archivo donde se guardará el video. El segundo parámetro es una cadena de texto que indica el modo de grabación. Los valores válidos para el modo son: "w" para sobrescribir el archivo si existe, y "a" para añadir la grabación al archivo actual.
- **record()**: Inicia la grabación de video. Es necesario haber fijado previamente un nombre de archivo. En caso contrario no se inicia la grabación y el dispositivo continúa en estado 'STOPPED'.
- **stop()**: Detiene la grabación de video.

8.7 La Especificación Estructural de Componentes

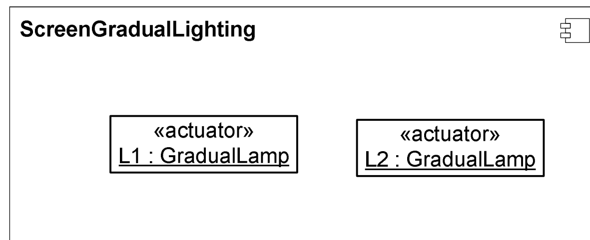
En la Especificación Estructural de Componentes se describen aquellos proveedores de enlace que forman un componente.

Se presentan el modelo estructural de componentes utilizando la herramienta de generación de código y como se ha comentado antes se acompaña del mismo modelo creado con una herramienta orientada a producir documentación, con el fin de facilitar la legibilidad de los modelos



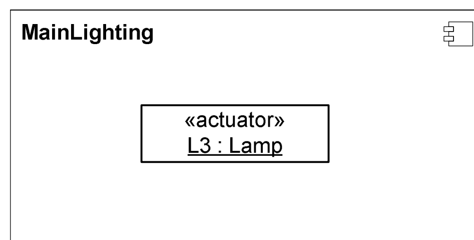
ScreenGradualLighting

Este servicio será proporcionado por dos lámparas graduales.



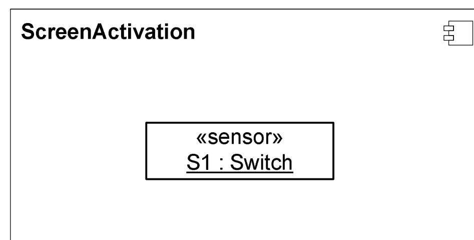
MainLighting

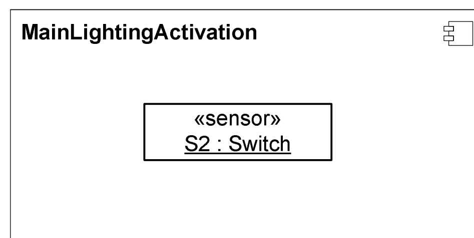
La iluminación principal está formada un una única lámpara.



ScreenActivation y MainLightingActivation

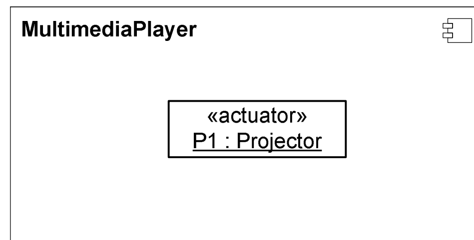
Los servicios de activación manual son implementados por sendos interruptores.





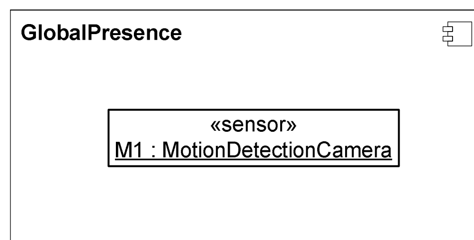
MultimediaPlayer

Para implementar este servicio se ha elegido un proyector.



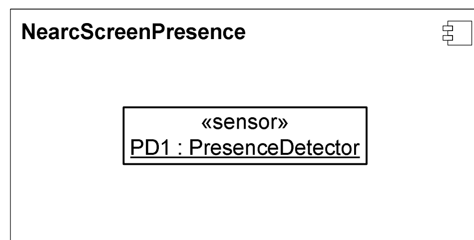
GlobalPresence

Este servicio será proporcionado por una cámara de video con capacidad de detectar presencia.



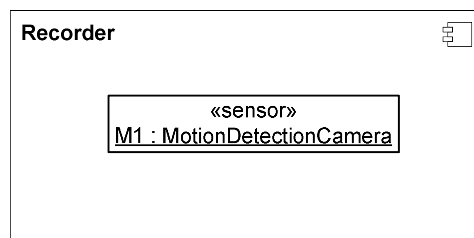
NearScreenPresence

Para implementar este servicio se utiliza un detector de presencia que abarca toda la sala.



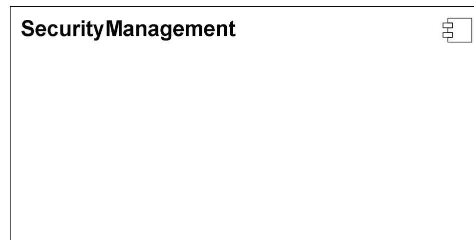
Recorder

Este servicio es proporcionado por una cámara. Nótese que se trata de la misma cámara utilizada en el servicio GlobalPresence (tiene el mismo identificador “M1”).



SecurityManagement

El servicio de seguridad utiliza directamente la funcionalidad de otros servicios. Por lo tanto, no necesita disponer de proveedores de enlace propios.



8.8 La Especificación Funcional de Componentes

En la Especificación Funcional de componentes de indican las acciones que se llevarán a cabo cuando se invoque una operación de un componente. Estas acciones podrán invocar operaciones de sus proveedores de enlace o de otros componentes con los que se encuentren relacionados.

Para la especificación funcional de componentes se utilizan las Acciones de UML 2.0. Debido a que estas acciones no tienen una sintaxis concreta, actualmente estamos utilizando la propuesta por Kennedy Carter.

Por comodidad, se utilizan las siguientes notaciones:

- "_NOMBRE_" para representar "this->BBProviders where nombre = 'NOMBRE'"
- "^NOMBRE^" para representar "this->Components where name = 'NOMBRE'"

Además, se considera que aquellas operaciones que devuelven un valor tienen definida una variable llamada "returnValue". Finalmente, aquellas variables que deben conservar su valor entre invocaciones de las operaciones se precederán del símbolo "@"; por ejemplo "@file".

Es importante destacar que, en ocasiones, la funcionalidad implementada en un componente no es o no puede ser la descrita de utilizando lenguaje natural debido al modo de funcionamiento de los proveedores de enlace utilizados. Pese a eso, siempre se cumplen las pre y postcondiciones especificadas en el Modelo de Servicios.

ScreenGradualLighting

void switchOn()

```
for lamp in { this->BProviders.Lamp } do
   lamp.on();
endfor
```

void switchOff()

```
for lamp in { this->BProviders.Lamp } do
   lamp.off();
endfor
```

bool isLighting()

```
returnValue = _L1_.isOn() or _L2_.isOn();
```

void setIntensity(Intensity : int)

```
bulbsOn = 0; // Para que bulbsOn sea de tipo entero
bulbsOn = ( 7 * Intensity ) / 100;

for lamp in { this->BProviders.Lamp } do
```

```
   difference = lamp.currentPos() - bulbsOn;
   if difference > 0 then
         loop
              break if difference = 0;
              lamp.decrease();
              difference = difference - 1;
          endloop
   else if difference < 0 then
          loop
              break if difference = 0;
              lamp.increase();
              difference = difference + 1;
          endloop
        endif
   endif
endfor
```

int getIntensity()

```
returnValue = ( _L1_.currentPos() + _L2_.currentPos() ) / 14 ;
```

MainLighting

void switchOn()

```
_L3_.on();
```

void switchOff()

```
_L3_.off();
```

bool isLighting()

```

returnValue = _L3_.isOn();
```

ScreenActivation

bool isActivated()

```
returnValue = _S1_.activated ();
```

MainLightingActivation

bool isActivated()

```
returnValue = _S2_.activated();
```

MultimediaPlayer

void on()

```
_S2_.play("");

@fakeState = "";
```

void off()

```
_P1_.switchOff();

@fakeState = "";
```

void play()

```
_P1_.playFile(@file);

@fakeState = "";
```

void forward()

```
if (( _P1_.currentSecond() + 60 ) < _P1_.length() ) then
   _P1_.moveToSecond(_P1_.currentSecond() + 60 );
endif

@fakeState = "FORWARDING";
```

void rewind()

```
if ( _P1_.currentSecond() > 60 ) then
   _P1_.moveToSecond(_P1_.currentSecond() - 60 );
endif

@fakeState = "REWINDING";
```

void stop()

```
_P1_.moveToSecond( 0 );
_P1_.pause();
```

String currentMode()

```
if @fakeState != "" then

   returnValue = @fakeState;

else

   state = _P1_.getState();

   switch state
   case 0
          returnValue = "PLAYING";
   case 1
```

```
            returnValue = "ON";
    case 2
            returnValue = "OFF";
    endswitch

endif
```

void setMediaSource(mediaFile : String)

```
@file = mediaFile;
```

String currentMediaSource()

```
returnValue = @file;
```

GlobalPresence

bool presenceDetected()

```
returnValue = _M1_.getMotionDetection();
```

NearScreenPresence

bool presenceDetected()

```
returnValue = _PD1_.probabilityOfPresence > 80 ;
```

Recorder

void on()

```
// La cámara permanece encendida pero sin grabar.
_M1_.stop();
@fakeState = "";
```

void off()

```
_M1_.stop();
@fakeState = "OFF";
```

void record()

```
_M1_.saveTo( "smb://servidor/privado/grabacion.avi" , "a" );
_M1_.record();
@fakeState = "";
```

void stop()

```
_M1_.stop();
@fakeState = "";
```

String currentMode()

```
if @fakeState = "OFF" then
```

```
   returnValue = "OFF";
else

  state = _M1_.getState();

  switch state
  case "RECORDING"
      returnValue = "RECORDING"
  case "STOPPED"
      returnValue = "ON"
  default
      returnValue = "OFF"
  endswitch

endif
```

String savedMedia()

```
returnValue = "smb://servidor/privado/grabacion.avi";
```

SecurityManagement

void enable()

```
@enabled = TRUE;
```

void disable()

```
@enabled = FALSE;
```

bool isEnabled()

```
returnValue = @enabled ;
```

8.9 Despliegue

Tal y como se contó en el capitulo *Contexto* los pasos a seguir para la implantación de un sistema pervasivo son:

1) El analista capturaría los requisitos del sistema y construiría los modelos gráficos que describen el sistema pervasivo.
2) Se aplicaría el motor de trasformación a los modelos del sistema, generando código java y otros recursos como resultado.
3) Se desarrollarían los drivers que proporcionan acceso a los dispositivos o al software externo al sistema.
4) Se deben ajustar los archivos java para poder utilizar los drivers deseados. El ajuste únicamente implicaría indicar el ID del driver a utilizar.
5) Finalmente el código java se compila y empaqueta en bundles (archivos jar).Para el funcionamiento del sistema, en el servidor debe haber una implementación del framework y los drivers necesarios.

Tras el modelado del sistema pervasivo y su posterior generación de código, se obtienen proyectos de eclipse en los que se encuentra implementada la parte software del sistema. Con esto se habrían realizado los pasos 1) y 2). El siguiente paso consistiría en 3) desarrollar los drivers para acceder a los dispositivos. dado que el objetivo de este proyecto no es la implementación de drivers, se seleccionaran aquellos que sean necesarios de un repositorio de drivers ya existentes.

Los dispositivos disponibles para utilizar en el caso de estudio son:

- Multipusador Transcend del fabricante Merten.
- Actuador eib duo del fabricante Lingg & janke.
- Regulador de intensidad del fabricante Lingg & janke.
- Dos bombillas.

Dado que no se disponen de todos los dispositivos reales necesarios para implementar el caso de estudio completamente, se ha recurrido a un simulador de dispositivos. Es posible instalar drivers en este simulador que reproducen el comportamiento de los dispositivos reales.

Los drivers simulados utilizados son:

- Simulador de presencia.
- Interruptor.
- Cámara con detección de movimiento.
- Projector.

En el paso 4) se debe actualizar los IDs del código generado para que coincida con el de los drivers que se utilizan. Para ello se tiene que modificar el archivo que contiene el activador de cada binding provider, cambiando el valor de la variable ”driverPID” por el del ID del driver con el que se relaciona.

Finalmente en el paso 5) solo es necesario exportar los proyecto generados en bundles. Para ello seleccionaremos la opción *export* en el menú contextual de cada proyecto y tras indicar que se exportara como un archivo *jar* seleccionaremos el archivo manifiesto que ha generado la herramienta dentro del proyecto Eclipse. Tras repetir el proceso con todos los proyectos generados, solo será necesario cargar los bundles en un servidor.

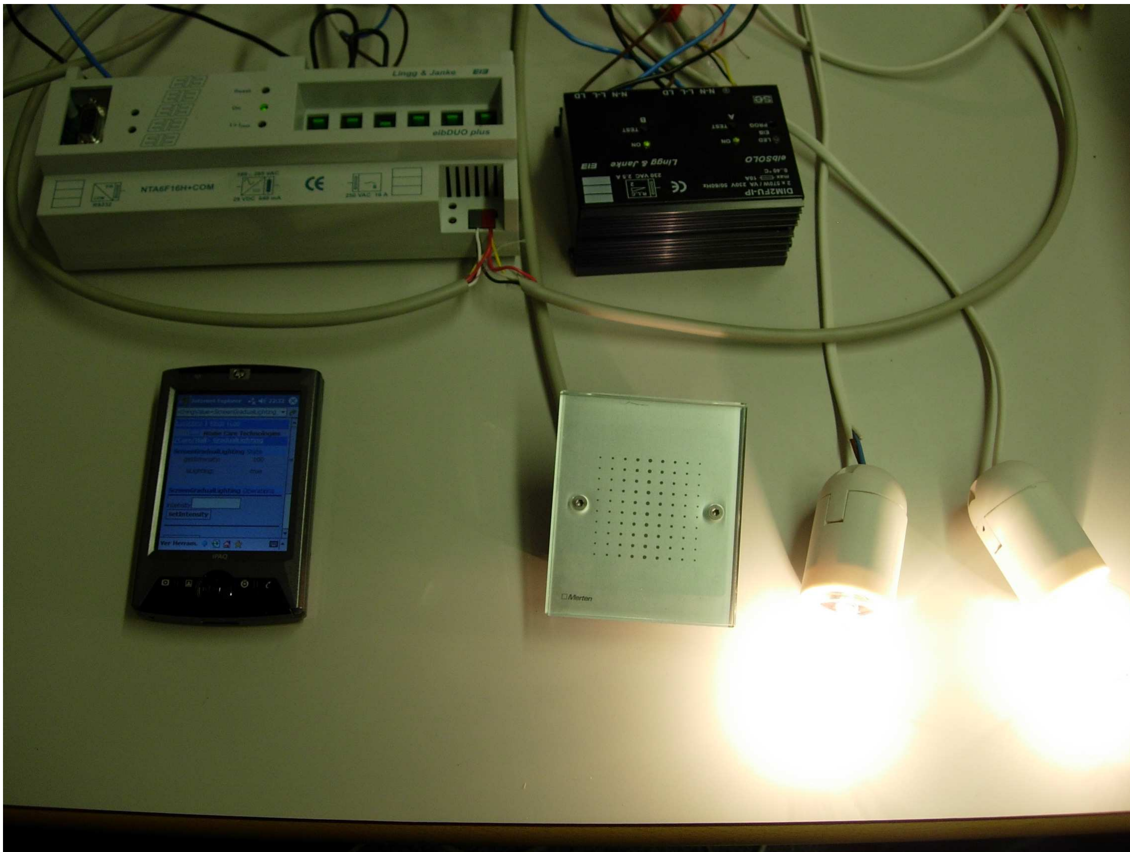


Figura 71: Dispositivos reales.

En el servidor se dispone de acceso a una red domótica por medio de la conexión RS232 del computador. También estará instalada una implementación comercial de OSGi llamada Prosyst, en necesaria para el acceso a la red domótica. Los bundles del sistema generado deberán ser cargados en el servidor junto al simulador de dispositivos y todos los drivers, tanto simulados como reales.

Los pasos para realizar la carga de bundles son:

- Ejecutar la consola grafica de Prosyst.
- Conectarse al servidor de Prosyst local.
- Seleccionar en el menú “Bundles\Install bundle”
- Tras seleccionarse la opción jar se debe indicar la ruta del bundle.
- Una vez instalado el bundle, debe iniciarse, para ello se utiliza la opción “Start” de su menú contextual.

Al existir dependencias entre los bundle generados es necesario realizar la carga siguiendo un orden que garantice ante la carga de un bundle concreto, la disponibilidad de sus bundles prerrequisito.

Un posible orden seria:

- Interfaces de binding providers.
- Drivers
- Binding providers.
- Componentes genéricos.
- Componentes concretos.
- Interacciones.

8.10 Sistema generado en funcionamiento

Una vez que todos los bundles se encuentran instalados y en ejecución es posible interactuar con el sistema por medio de los dispositivos reales o a través de la interfaz del simulador de dispositivos. Como se describe en los modelos, existe una interacción definida por la cual si se detecta presencia cerca de la pizarra y se encuentran las luces encendidas, éstas reducen su luminosidad gradualmente para mejorar la visibilidad de la pizarra.

Para reproducir este comportamiento sería necesario realizar dos pasos:

1. Establecer en el simulador de dispositivos que el interruptor simulado pasa a estar activado, por lo que se satisfacería la condición de unos de los disparadores definidos en los modelos, disparándose la acción de encender completamente la luz gradual.
2. De nuevo en el simulador, se modificará el estado del detector de presencia para que simule la presencia cerca de la pizarra. Este acto desencadenará la ejecución de la interacción, decrementando la luminosidad para mejorar la visibilidad de la pizarra.

Como se ha visto en la carga de los bundles, el sistema pervasivo permite la carga en caliente de nuevos bundles, explotando esta característica. Es posible cargar un bundle con una interfaz grafica de usuario web para PDA. En esta nueva interfaz se descubrirán los servicios PervML existentes en el sistema y permitirá acceder a su funcionalidad.

Es posible navegar por el sistema con la nueva interfaz de usuario y comprobar como los bundles generados se comportan fielmente a los modelos.

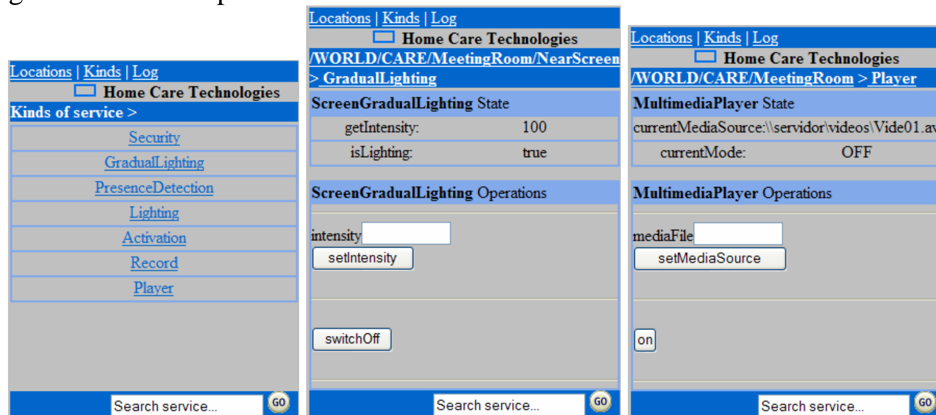


Figura 72: Interfces del sistema en funcionamiento.

9 Conclusiones

En este capítulo se abordara el proyecto desde los puntos de vista de calidad y esfuerzo. Además se remarcara las conclusiones derivadas de la experiencia del proyecto que servirán para futuros proyectos. Finalmente se listan las publicaciones en las que se ha participado, a raíz de la participación en el grupo de investigación donde se realizo el proyecto.

9.1 Resumen del trabajo realizado

En este proyecto fin de carrera se ha desarrollado una herramienta de generación de código automático para sistemas pervasivos. La herramienta desarrollada en este proyecto final de carrera proporciona un editor grafico para la creación y manipulación de modelos PervML, un lenguaje de modelado para especificar este tipo de sistemas. Para implementar la herramienta se ha hecho uso de las tecnologías EMF, GEF y GMF, las cuales son plug-ins de la plataforma Eclipse.

9.2 Calidad

El modelo de calidad establecido en la primera parte del estándar ISO 9126-1, clasifica la calidad del software en un conjunto de seis factores:

- Funcionalidad.
- Fiabilidad.
- Eficiencia.
- Portabilidad.
- Mantenimiento.
- Amigabilidad.

A continuación se intentan describir algunas de las características de la herramienta que pueden dar soporte a estos factores:

La utilización de las tecnologías EMF, GEF y GMF ha proporcionado a la herramienta una buena arquitectura de código, ya que estas tecnologías embeben buenas prácticas de programación, tal y como se ha mostrado en detalle al describir cada tecnología. Una buena arquitectura de código facilita el **mantenimiento** de la herramienta frente a posibles cambios en los requisitos.

Para la elección de las metáforas graficas que representarían los elementos del metamodelo, se consideraron aquellas opciones empleadas en las herramientas más utilizadas para el modelado de sistemas software. Con el objetivo de facilitar el aprendizaje de la herramienta en pro de la **amigabilidad** de esta.

La utilización de la tecnología Java dota a la herramienta de lo **posibilidad de ser portada** a otras plataformas, sin la necesidad de realizar modificaciones. Únicamente se exige que exista una implementación de la máquina virtual de Java para la máquina destino.

El desarrollo del caso de estudio descrito en el capitulo *Caso de estudio* tenía como objetivo el modelado y la generación de código del sistema, para demostrar la **funcionalidad** de la herramienta.

Considerando que no se han realizado pruebas para establecer en que grado la herramienta cubre los factores de fiabilidad y eficiencia se puede concluir que de acuerdo al estándar ISO 9126-1se ha desarrollado una herramienta que cubre los principales factores de calidad.

9.3 Esfuerzo

El proyecto final de carrera según el plan de estudios tiene asignados 15 créditos, equivalente a 150 horas lectivas, que exigirían en la práctica aproximadamente unas 300 horas de dedicación. El desarrollo desde cero de una herramienta con características similares a las obtenidas en este proyecto final de carrera sin el uso de frameworks de de ayuda sobrepasaría ampliamente las 300 horas de dedicación. La utilización de los plug-in EMF, GEF, GMF y MOFScript ha permitido abordar el desarrollo del proyecto dentro del tiempo asignado a un proyecto final de carrera de 15 creditos.

9.4 Conclusiones para futuros proyectos

A continuación se describen las conclusiones obtenidas, estructuradas de acuerdo a los principales módulos de la herramienta:

Gestión de modelos:

- La expresividad del lenguaje ecore de EMF ha sido suficiente para soportar completamente el lenguaje PervML.
- Los asistentes de EMF para importar modelos Ecore a partir de otros formatos permiten trabajar con otras herramientas más potentes para la definición de modelos, como Racional Rose, e importar sus resultados a Ecore.

Edición grafica de modelos:

- La expresividad proporcionada por GMF es suficiente para definir una representación visual de un modelo.
- Es posible crear editores multivista sobre un mismo modelo, utilizando el concepto de shortcut proporcionado por GMF.

Generación de código

- La facilidad de integración del plug-in MOFScript con EMF junto a las funciones predefinidas que proporciona, aconseja su uso en futuros proyectos de generación de código en los que se utilice EMF como soporte para gestión de modelos.

9.5 Publicaciones

Como ya se ha comentado en varios puntos de este documento, el Proyecto Final de Carrera se ha desarrollado en el contexto del grupo de investigación OO-Method, lo cual ha posibilitado al estudiante la participación en varias publicaciones de carácter científico que se enumeran a continuación

Applying a Model-Driven Method to the Development of a Pervasive Meeting Room
Javier Muñoz, Estefanía Serral, Carlos Cetina and Vicente Pelechano
ERCIM News
April 2006. vol. 65, pp. 44-45, ISSN: 0926-4981

Implementing a Pervasive Meetings Room: A Model Driven Approach
Javier Muñoz, Vicente Pelechano, Carlos Cetina
International Workshop on Ubiquitous Computing (IWUC 2006), Paphos, Cyprus
23 May 2006. pp. 13 - 20, ISBN: 972-8865-51-1

Un Framework basado en OSGi para el Desarrollo de Sistemas Pervasivos
Javier Muñoz, Carlos Cetina, Estefanía Serral, Vicente Pelechano
9 Workshop Iberoamericano de Ingenieria de Requisitos y Ambientes Software (IDEAS2006), La Plata (Argentina)
24 - 28, Apr 2006. pp. 257 - 270

Software Engineering for Pervasive Systems. Applying Models, Frameworks and Transformations.
Javier Muñoz, Vicente Pelechano, Carlos Cetina
I International Workshop on Software Engineering for Pervasive Services (SEPS 2006), Lyon (France)
29 June 2006

Software Engineering for Pervasive Systems. Applying Models, Frameworks and Transformations.
Javier Muñoz, Vicente Pelechano, Carlos Cetina
I International Workshop on Software Engineering for Pervasive Services (SEPS 2006), Lyon (France)
29 June 2006

Un framework para la simulación de sistemas pervasivos
Javier Muñoz, Idoia Ruiz, Vicente Pelechano, Carlos Cetina
Simposio sobre Computación Ubicua e Inteligencia Ambiental (UCAmI'05), Granada (Spain)
September 2005
pp. 181 - 190, ISBN: 84-9732-442-0

Referencias

[1] M. Weiser. The computer for the 21st century. Scientific American, 265(3):94–104, Sept. 1991.

[2] Manoli Albert, Vicente Pelechano, Joan Fons, Marta Ruiz, and Oscar Pastor. Implementing UML ssociation, Aggregation and Composition. A Particular Interpretation based on a Multidimensional Framework. In Johan Eder and Michele Missikoff, editor, CAiSE 2003, pages 143–158, 2003.

[3] V. Pelechano. Managing Taxonomic Relationships in Automatic Software Production Enviroments: A Pattern based Approach. UMI Pro-Quest Digital Dissertationns, 2001.

[4] J. Muñoz and V. Pelechano, “Building a Software Factory for Pervasive Systems Development,” in CAiSE 2005, Porto, Portugal, June 13-17, ser. LNCS, vol. 3520, May 2005, pp. 329–343.

[5] Jack Greenfield, Keith Short, Steve Cook, Stuart Kent, John Crupi, Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, 2004, ISBN: 0471202843.

[6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad y Michael Stal, Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley & Sons, 1996 ISBN: 0471958697.

[7] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995 ISBN: 0201633612.