UNIVERSIDAD DE CANTABRIA
Facultad de Ciencias

Proyecto Fin de Carrera

DESCRIPCIÓN DE SISTEMAS BASADOS EN COMPONENTES UTILIZANDO UML 2

(Component-based systems description using UML 2)

Para acceder al Titulo de

INGENIERO EN INFORMÁTICA

Autor: José María Martínez Lanza

Septiembre - 2011

Facultad de Ciencias

Facultad de Ciencias

INGENIERÍA EN INFORMÁTICA

CALIFICACIÓN DEL PROYECTO FIN DE CARRERA

Realizado por: José María Martínez Lanza
Director del PFC: Patricia López Martínez
Título: "Descripción de sistemas basados en componentes utilizando UML 2"
Title: "Component-based systems description using UML 2"

Presentado a examen el día:

para acceder al Título de

INGENIERO EN INFORMÁTICA

Composición del Tribunal:
Presidente (Apellidos, Nombre):
Secretario (Apellidos, Nombre):
Vocal (Apellidos, Nombre):
Vocal (Apellidos, Nombre):
Vocal (Apellidos, Nombre):

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: Vocal

Fdo.: Vocal

Fdo.: Vocal

Fdo.: El Director del PFC

Facultad de Ciencias

Tabla de contenidos

Tabla de figuras

1. Resumen

El desarrollo de aplicaciones basadas en componentes es una metodología de la ingeniería del software, que enfatiza en la composición de sistemas a partir de componentes funcionales o lógicos reutilizables con interfaces bien definidas usadas para la comunicación entre ellos.

Uno de sus principios básicos es la opacidad: los diseñadores de aplicaciones deben manejar el código de los componentes de forma opaca, sin acceso a sus detalles internos ni posibilidad de modificarlo. Para ello se requiere que los desarrolladores de componentes incluyan, como parte del componente, los metadatos que permiten hacer un uso opaco de los mismos. La especificación "*Deployment and Configuration of Component-based Distributed Application*" define un metamodelo de soporte para la especificación de dichos metadatos y su utilización en el proceso de despliegue y configuración de aplicaciones distribuidas basadas en componentes.

Una de las principales formas de modelar los elementos de una aplicación, sea en la metodología que sea, es el lenguaje de modelado UML. Pero éste no aporta la funcionalidad necesaria para los programadores que despliegan aplicaciones basadas en componentes, por lo que necesitan una base en la cual sustentarse. Esa base puede ser aplicada como un perfil UML que de soporte a los descriptores definidos en la especificación D&C.

En este trabajo se define dicho perfil, que se compondrá de todos los elementos que forman parte de esta especificación, divididos en:

- Modelo de componentes: Describe la especificación, implementación y empaquetado de un componente.
- Modelo de plataformas: Describe la información sobre los recursos de la plataforma.
- Modelo de despliegue: Describe las instancias, conexiones, despliegue y configuración de una aplicación construida por ensamblado de componentes.

A continuación, el perfil será aplicado sobre una aplicación en concreto para ver los diferentes componentes que la conforman, cómo aplicar los estereotipos y cómo desplegarla en una plataforma en concreto. El resultado constituye una metodología de modelado de sistemas basados en componentes, donde se definen las pautas de modelado que se han de seguir para ser capaces de llevar a cabo el proceso automático de despliegue de una aplicación.

Asimismo, se ha definido una extensión del perfil estándar para dar soporte al desarrollo de aplicaciones de tiempo real basadas en componentes. La naturaleza de tiempo real de componentes y aplicaciones requiere el manejo de un nuevo tipo de información, que al igual que en el caso anterior, debe ser gestionada de forma opaca, lo que requiere una ampliación de los metadatos asociados a componentes y aplicaciones.

1.1 Abstract

Component-based application development is a software engineering methodology that emphasizes the composition of systems from functional or logical reusable components with well-defined interfaces used for communication between them.

One of the basic of component-based development is opacity: the application developers must manage the code of the components in an opaque way, with no access to internal details nor modifications. This requires component developers to include metadata that allow an opaque use of the components. The "*Deployment and Configuration of Component-based Distributed Application*" specification defines a metamodel to specify component metadata, and the way they have to be handled along the deployment and configuration process.

One of the main ways to model elements of an application, independently of the underlying methodology, is the Unified Modeling Language (UML). But this language doesn't support the whole functionality required for component-based application developers. UML Profile supporting the concepts defined by the D&C specification can be used with that purpose.

This profile is defined in this project, composed of all the elements defined in the specification is divided in:

- Component model: It describes the information about specification, implementation and packaging of components.
- Target model: It describes all the information about platforms.
- Deployment model: It describes instances, connections, deployment and configuration of an application.

This profile will be applied in a particular application to distinguish what components compose it, how to apply stereotypes on them, and how to deploy it on a specific target. As a result, a methodology on component-based systems modeling is established, where the patterns to follow are described.

An extension to the profile has also been defined to support the development of real-time component-based applications. The nature of these applications requires new types of information, that must be managed also in an opaque way, requiring an extension of the metadata associated to components and applications.

2. Introducción

2.1. Desarrollo basado en componentes

Actualmente, los sistemas informáticos van aumentando gradualmente su complejidad. Se demandan tiempos de desarrollo de aplicaciones más cortos y procesos mucho más ágiles. Una de las claves para solucionar este problema está en el paradigma del desarrollo de aplicaciones basado en componentes.

El desarrollo basado en componentes [1] [2] es considerado por la industria actual como una de las soluciones más eficientes para abordar la creciente complejidad de los sistemas informáticos y adaptar el desarrollo de nuevas aplicaciones a la velocidad de evolución del mercado.

El desarrollo de aplicaciones basadas en componentes tiene por objetivo la composición de éstas mediante piezas reutilizables y conectadas entre sí, es decir, el desarrollo de una aplicación por composición e interconexión de componentes software reutilizables, que han sido previamente implementados por terceras personas con independencia de la aplicación en la que vayan a ser utilizados.

La adopción de una estrategia de desarrollo orientada a componentes introduce cambios claros en el proceso de desarrollo de aplicaciones. De hecho, en una metodología de desarrollo orientada a componentes se distinguen claramente dos procesos independientes, pero a su vez complementarios, pues los resultados de uno son las piezas constructivas del otro [3]:

- Desarrollo de componentes software reutilizables: El objetivo del desarrollo de componentes es el diseño, implementación y empaquetado de estos componentes, para su posterior almacenamiento en repositorios donde puedan ser utilizados por los desarrolladores de aplicaciones.
- Desarrollo de aplicaciones basadas en componentes: Para el desarrollo de aplicaciones, la configuración y ensamblado de los componentes almacenados en los repositorios, elección de las implementaciones más adecuadas, y ejecución en la plataforma apropiada son los objetivos primordiales.

La clave para entender lo que es el desarrollo de aplicaciones basadas en componentes es profundizar en lo que es un componente y cómo estos se interconectan entre sí formando componentes de mayor complejidad e importancia. Sin embargo, no existe una definición consensuada de lo que es un componente software reutilizable. Los diferentes autores que han trabajado sobre el tema han dado sus propias definiciones, entre las que se encuentran las siguientes:

- Microsoft define un componente como "una pieza de software compilado que ofrece un servicio". [4]
- El Object Management Group define un componente en su especificación UML [5] como "una parte modular de un sistema que encapsula su contenido y cuya manifestación es reemplazable en su entorno. Un componente define su comportamiento en razón a sus interfaces ya sean proporcionadas o requeridas. Grandes partes de la funcionalidad de un sistema pueden ser ensambladas reutilizando componentes como partes de un componente mayor o un ensamblaje de componentes, uniéndolos todos mediante sus respectivos puertos requeridos y proporcionados."
- Una de las definiciones más comunes y universalmente aceptadas es la de Szyperski: "Un componente software es una unidad de composición con interfaces contractualmente especificadas y solo dependencias explícitas de contexto. Un componente software puede desplegarse independientemente y es implementado por terceras personas." [2]
- Por último, la definición de Heineman y Councill, que no dista de la anterior pero introduce el concepto de modelo de componentes: "Un componente es un elemento software que es conforme a un modelo de componentes, que puede ser desplegado independientemente, y compuesto sin ser modificado de acuerdo a una composición estándar." [6]

De todas estas definiciones y el resto que pueden encontrarse en la literatura, se pueden extraer las principales bases del paradigma orientado a componentes, entre las que destacamos dos:

- Obviamente, el concepto principal es el de la reutilización. El código de los componentes se desarrolla de manera independiente de la aplicación, e incluso de la plataforma, en la que vayan a ser utilizados. Son posteriormente almacenados en repositorios o librerías, a veces de acceso totalmente público, para que tengan un uso recurrente sin necesidad de que sean implementados de nuevo, evitando de esta forma su fase de desarrollo.
- Otra característica obvia es la composición. Los componentes deben poder conectarse entre sí para formar aplicaciones. Como afirma la última de las definiciones enumeradas anteriormente, para que el proceso de composición pueda llevarse a cabo de manera satisfactoria, es necesario que los componentes se desarrollen de acuerdo a algún modelo de componentes, que establezca una serie de reglas preestablecidas para definir y conectar componentes. Ejemplos de modelos de componentes son EJB [7], .NET [8], CCM [9], etc.
- Otra característica es la opacidad. Cuando se va a hacer uso de un componente en una aplicación, éste debe manejarse como un elemento de caja negra, es decir, se conocen su especificación, sus interfaces, sus entradas, sus salidas y las respuestas que produce, pero nunca su funcionamiento o implementación interna. Dicho de otro modo, cuando un componente es incluido en una aplicación, los diseñadores de la misma deben poder manejarlo sin tener nunca acceso a su código interno.

Para poder reutilizar un componente en una aplicación verificando el principio de opacidad es necesario gestionar el componente únicamente a través de un conjunto de metadatos proporcionados por el propio componente, que describen de modo abstracto aquellas características del componente que se necesitan conocer en cada fase del proceso de desarrollo de una aplicación. Por lo tanto, como muestra la figura 2.1, un componente se distribuye como un paquete en el que se incluyen tanto los ficheros de

código que constituyen su implementación, como los metadatos que permiten utilizarlo de modo opaco.

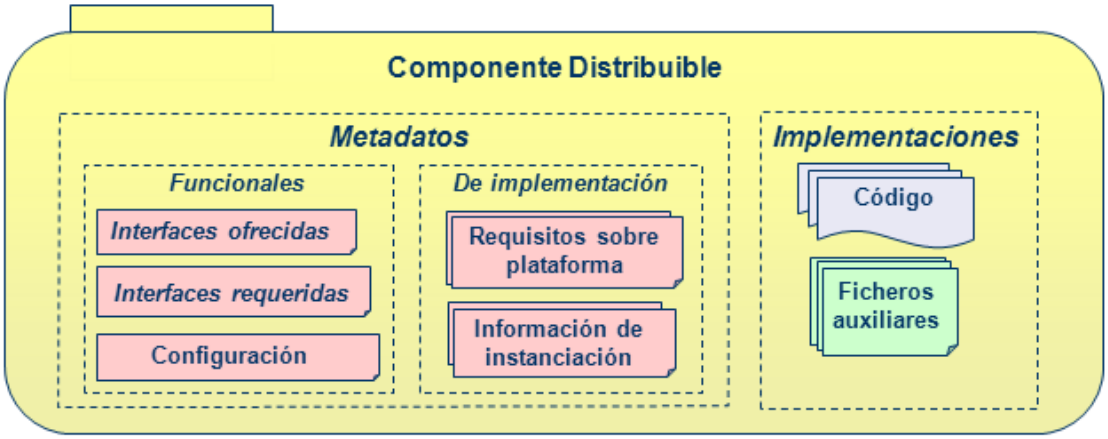

Figura 2.1: Paquete de distribución de un componente software

Otra idea importante es la de recursión, ya que una aplicación diseñada como un ensamblado –un conjunto de componentes interconectados- puede verse, a su vez, como un componente, y ser reutilizado como un ensamblado que implementa una interfaz de componente específica dentro de otro ensamblado.

A modo de resumen, se puede afirmar que un componente se basa en:

- Una especificación. Un componente incluye una descripción de los servicios que ofrece para actuar como vínculo o contrato con los futuros clientes. Además, también incluye información sobre los servicios que requiere y las interfaces que otros componentes le deben ofertar para poder operar.
- Al menos una implementación. Cada componente debe proporcionar una o más implementaciones, conformes a su especificación. Estas implementaciones, en el caso en el que no sean únicas, irán siendo descartadas por el diseñador de la aplicación hasta conseguir la que más se aproxime a las necesidades o requisitos del sistema en el que vaya a desarrollarse la aplicación.
- Una estandarización de sus elementos. Todos los componentes de un software se desarrollan dentro de un entorno definido o modelo de componentes, que se rige de unas reglas que deben de tomarse en cuenta para aprovechar todos los servicios que el propio componente oferta.
- La idea del empaquetado. Los componentes deben agruparse de diferentes formas para proveer ciertos servicios y facilidades. Este agrupamiento se llama paquete. Lo más común en el desarrollo de aplicaciones basado en componentes es que estos paquetes se adquieran de terceras personas ajenas o no al proyecto que se está desarrollando.
- La idea del despliegue. Una vez empaquetados e instalados los componentes en un entorno operacional, estos se pueden desplegar. El despliegue implica una instanciación de los componentes y las interacciones con los que les rodean.

2.2. Especificación D&C de OMG

Como se ha comentado anteriormente, una de las principales características del desarrollo basado en componentes, a la que en este trabajo se le da especial importancia, es la opacidad con la que los componentes deben ser manejados por los desarrolladores de aplicaciones. Para conseguir esta opacidad es necesario gestionar el componente únicamente a través de metadatos que describan las características del componente que se necesitan conocer en cada fase del proceso de desarrollo de una aplicación que utilice el componente. Para la formulación de estos metadatos es deseable utilizar algún tipo de notación estándar, por lo que se han desarrollado especificaciones que definen los formatos y la semántica de los diferentes tipos de metadatos que se pueden asociar a un componente.

La especificación utilizada en este proyecto es "*Deployment and Configuration of Component-based Distributed Applications Specification*", en su versión 4.0, aprobada el 2 de abril de 2006 por el OMG (Object Management Group) [19]. Este grupo, fundado en 1989, y formado por importantes compañías relacionadas con la industria de la computación como son Microsoft Corporation, IBM, Eclipse Fundation o Hewlett-Packard, entre otras, se encarga del establecimiento, publicación y mantenimiento de diferentes estándares y especificaciones para aplicaciones interoperables, portables y reutilizables desarrolladas en entornos distribuidos y heterogéneos, todo ello sin ningún ánimo de lucro, como remarcan en los prefacios de sus publicaciones.

La especificación está disponible en la web del OMG [10] y su objetivo principal es el de definir los formatos de los modelos y metadatos que se utilizan para describir componentes y aplicaciones, facilitando un proceso estándar de despliegue y configuración de aplicaciones basadas en componentes en plataformas distribuidas heterogéneas. De esta especificación se toman los conceptos básicos de componente y de aplicación desarrollada como ensamblado de componentes, así como la definición de los procesos complementarios de desarrollo de componentes y aplicaciones.

En términos más específicos, el documento define:

- Los metadatos que permiten describir componentes y ensamblados de componentes, así como las interfaces que permiten gestionar dichos metadatos, los cuales se recogen en el "*Component Data Model*" y en el "*Component Management Model*", respectivamente.
- Los metadatos que permiten definir plataformas distribuidas, así como las interfaces que gestionan todos esos metadatos. Se recopilan en el "Target Data Model" y "*Target Management Model*".
- Los metadatos que permiten describir el despliegue de una aplicación específica en un sistema distribuido en particular y las interfaces de las herramientas que guían el despliegue se describen en el "*Execution Data Model*" y en el "*Execution Management Model*", respectivamente.
- El proceso de despliegue (instalación, configuración, preparación y lanzamiento) de una aplicación.

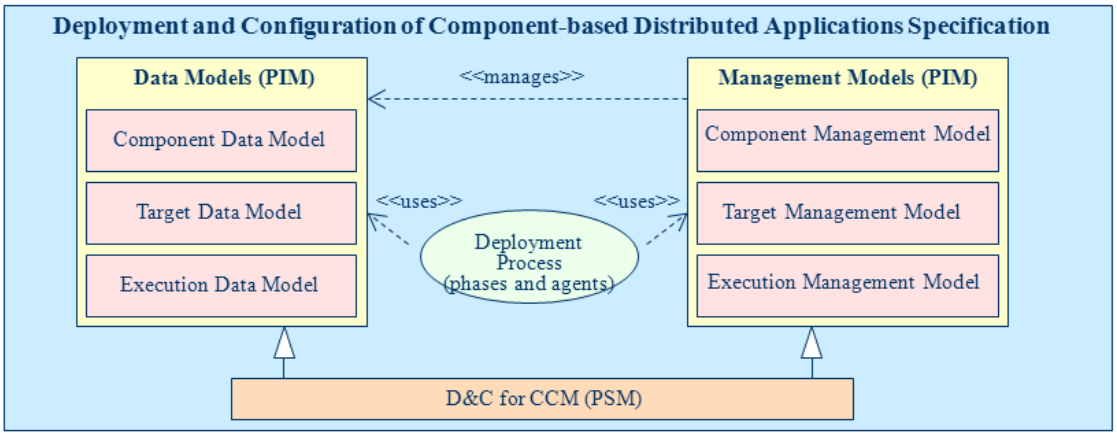

Figura 2.2: Estructura de la especificación D&C

La especificación D&C se ha formulado conforme a la especificación MDA [11]. El núcleo de la especificación se define como un modelo independiente de plataforma (PIM), dividido en los tres submodelos que se han citado: componentes, plataforma y ejecución. Todos los conceptos incluidos en estos submodelos son independientes de la tecnología de componentes, lenguaje de programación, formato de almacenamiento de información, etc., que sean empleados para el desarrollo de los componentes o de las aplicaciones. Si se quiere adaptar la especificación D&C a una tecnología de componentes específica, es necesario realizar la transformación del modelo PIM a un modelo específico de plataforma (PSM), incluyendo en él todos aquellos aspectos que sean característicos de dicha tecnología. La especificación incluye un ejemplo de dicho proceso de transformación para el caso de la tecnología CCM.

2.2.1. Proceso de desarrollo de componentes según D&C

D&C no define de manera explícita el proceso de desarrollo de componentes, sin embargo, sí define los agentes que intervienen en él y las responsabilidades de cada uno. A través de la explicación del proceso, se van a introducir los principales elementos de modelado (metadatos) que la especificación propone para describir componentes software reutilizables (formando parte del modelo de datos de componentes).
Como muestra la figura 2.3, el desarrollo de un componente software se divide en las fases de especificación, implementación y empaquetado.

Fase de especificación:
Esta fase es responsabilidad del agente *Specifier*, que define la especificación de la interfaz o contrato funcional del componente, en respuesta a una funcionalidad requerida en el dominio de aplicación correspondiente. Dicha interfaz se formula a través de un elemento de tipo *ComponentInterfaceDescription*. En él se especifican los puertos del componente, tanto ofertados como requeridos, así como sus propiedades de configuración. Cada puerto se describe a través de un elemento de tipo *ComponentPortDescription*, donde se especifica la naturaleza del puerto (la interfaz que implementa, si es requerido u ofertado, etc.). Las propiedades se definen, a su vez, por medio de un elemento *ComponentPropertyDescription*, donde se le asigna un nombre, un tipo y un valor por defecto (opcional).

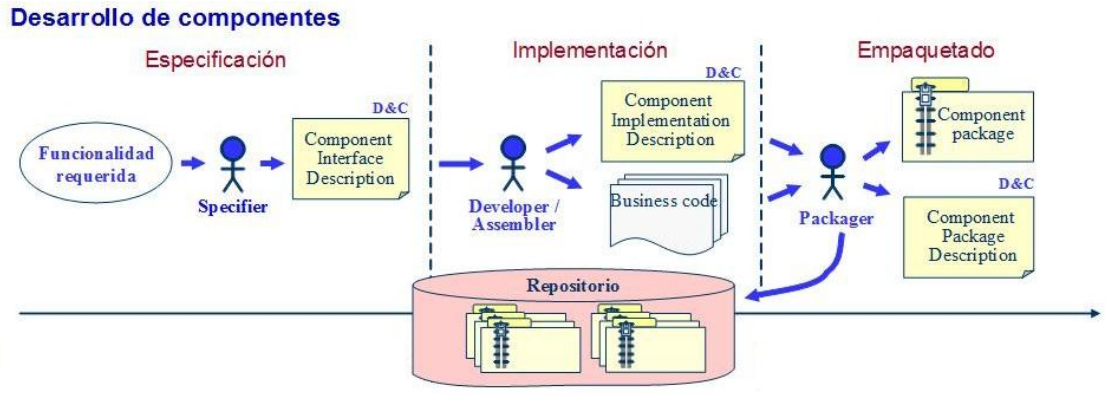

Figura 2.3: Fases del desarrollo de un componente según D&C

Fase de implementación:
Está liderada por el agente *Developer*, si se trata de implementaciones monolíticas, o *Assembler*, si se trata de implementaciones por ensamblado. En ella se crean implementaciones específicas, monolíticas o por ensamblado de otros componentes, para las interfaces de los componentes. Los metadatos más notorios que nos ofrece la especificación en cuanto a esta fase se refiere son:

- *ComponentImplementationDescription*: Describe una implementación determinada para una interfaz de componente, que puede ser monolítica o un ensamblado de varios componentes.
- *ComponentAssemblyDescription*: Describe una implementación de componente que se construye como un ensamblado de instancias de otros componentes. Contiene información acerca de las instancias de los subcomponentes que forman el ensamblado, las conexiones entre sus puertos y el mapeado de sus propiedades. Cada instancia incluida en el ensamblado se cualifica únicamente indicando la interfaz de componente que debe implementar, sin hacer referencia todavía implementaciones concretas.
- *MonolithicImplementationDescription*: Describe las características principales (archivos de código, requisitos para la instalación, etc.) de una implementación monolítica de componente.
- *ImplementationArtifactDescription*: Describe un artifact asociado a una implementación monolítica de componente en concreto. Contiene la localización del archivo de código que representa.

Fase de empaquetado:
Es la última fase del desarrollo de un componente. En ella, el agente *Packager* reúne toda la información acerca del componente (interfaz, implementaciones, descripciones de implementación, etc.) y crea un paquete, que constituye el componente distribuible para su posterior uso. Estos paquetes se almacenan en los anteriormente mencionados repositorios, y se describen a través del siguiente metadato:

- *PackageConfiguration*: Describe un paquete de componente. A través de él se puede acceder a cualquiera de los metadatos que el componente proporciona. Representa un producto de trabajo "reutilizable".

2.2.2. Proceso de desarrollo de aplicaciones según D&C

Al igual que la fase de desarrollo de componentes, el desarrollo de aplicaciones también se puede dividir en diferentes etapas: fase de configuración, fase de planificación y fase de ejecución (preparación y lanzamiento).

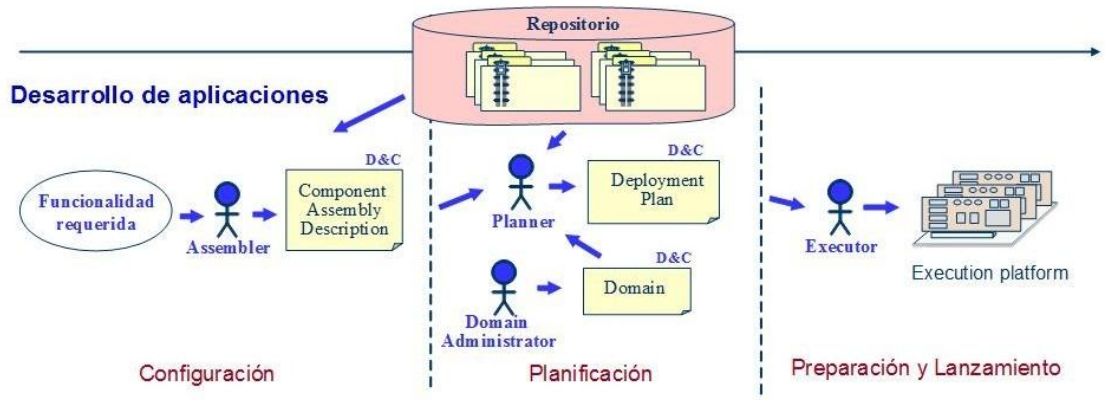

Figura 2.4: Fases del desarrollo de una aplicación según D&C

Fase de configuración:
Es la fase en la que se describe la aplicación como ensamblado de componentes. Aplicando el concepto de recursividad, esta fase está a cargo del agente *Assembler*, que al igual que en el caso anterior, escoge del repositorio las diferentes interfaces de componente y las une para crear ensamblados que cumplan los requisitos exigidos.

Fase de planificación:
Esta fase implica a dos agentes: el *DomainAdministrator*, que prepara toda la información concerniente a la plataforma en la que se va a desplegar y ejecutar la aplicación, y el *Planner*, que elige las implementaciones concretas para el despliegue, en base a las capacidades de la plataforma y los requisitos de cada componente, y las asigna a los correspondientes nodos.
Todo ello se describe a través de los siguientes metadatos:

- *Domain*: Descriptor de una plataforma. Engloba la información acerca los nodos que la forman, sus conexiones y los recursos que describen sus características principales.
- *DeploymentPlan*: Contiene toda la información necesaria para llevar a cabo el despliegue de la aplicación: las instancias de componentes que la forman (con sus correspondientes implementaciones), su información de configuración, sus conexiones, y su asignación a los nodos de la plataforma. El plan de despliegue se describe de forma plana, esto es, en él los ensamblados se han resuelto, por lo que aparecen las instancias concretas que lo forman, cada una con su propia configuración y conexiones.

Fase de ejecución:
Es la última fase. En ella, el agente *Executor* invoca las herramientas necesarias para ejecutar la aplicación. Estas herramientas trabajan exclusivamente con la información plasmada en el plan de despliegue. Desde él, deben ser capaces de acceder a toda la información necesaria para instanciar los componentes en sus nodos correspondientes, configurarlos, conectarlos y lanzar su ejecución.

2.2.3. Implementación actual de la especificación D&C

En el grupo de Computadores y Tiempo Real se ha elaborado una implementación de los modelos de datos D&C soportada por tecnología XML [12].
Se han definido una serie de pantillas W3-C Schema que mapean el metamodelo definido en la especificación. Los diferentes descriptores que se pueden utilizar para describir componentes, aplicaciones o plataformas se formulan por tanto a través de ficheros XML, validados frente a dichas plantillas. Estos ficheros se escriben a mano y, por tanto, están expuestos a faltas, fallos y demás errores humanos. Los diferentes ficheros a los que se da soporte, y que se utilizan en las diferentes fases de desarrollo son:

- Interfaz de componente (fichero .ccd.xml)
- Implementación de componente (fichero .cid.xml)
- Ensamblado de componente (fichero .cad.xml)
- Plataforma de ejecución (fichero tdm.xml)
- Plan de despliegue de la aplicación (fichero .cpd.xml)

Uno de los objetivos primordiales del proyecto era sustituir las implementaciones en XML utilizadas en el grupo CTR por modelos UML realizados a partir de una herramienta de desarrollo de modelos. De este modo, las herramientas que, actualmente, funcionan sobre ficheros XML, se lanzarán en un futuro directamente a partir de los modelos UML, tratando de reducir de esta forma el número de errores que se puedan cometer que no sean propios del computador, y facilitando, a su vez, la integración con estrategias MDA.

2.2.4. Lenguaje de modelado unificado

UML (Unified Modeling Language) [5], o lenguaje unificado de modelado, es un lenguaje gráfico de modelado respaldado por el OMG que ofrece elementos de modelado que permiten visualizar, especificar, construir y documentar sistemas software.
Cabe resaltar que UML es un lenguaje orientado a objetos por lo que es el complemento ideal para esta programación, aportando elementos como clases, componentes u objetos, aunque no solo se utilice para ella.

En la última publicación de la especificación de este lenguaje, la revisión 2.3 del OMG (Mayo 2010), UML propone una amplia variedad de elementos de modelado que se pueden emplear para modelar tanto la estructura como el comportamiento de un sistema software. Estos elementos pueden ser visualizados a través de diferentes tipos de diagramas, los cuales se pueden dividir en dos grandes grupos:

- Diagramas de estructura: definen los detalles estructurales del modelo, qué elementos deben estar presentes para que la aplicación funcione. Ellos son los diagramas de clases, de componentes, de estructuras compuestas, de despliegue, de objetos, de paquetes o de perfiles.
- Diagramas de comportamiento: define el comportamiento que debe tener el sistema, lo que debe suceder cuando la aplicación funcione. Estos son los diagramas de actividades. de casos de uso, de estados o de interacción.

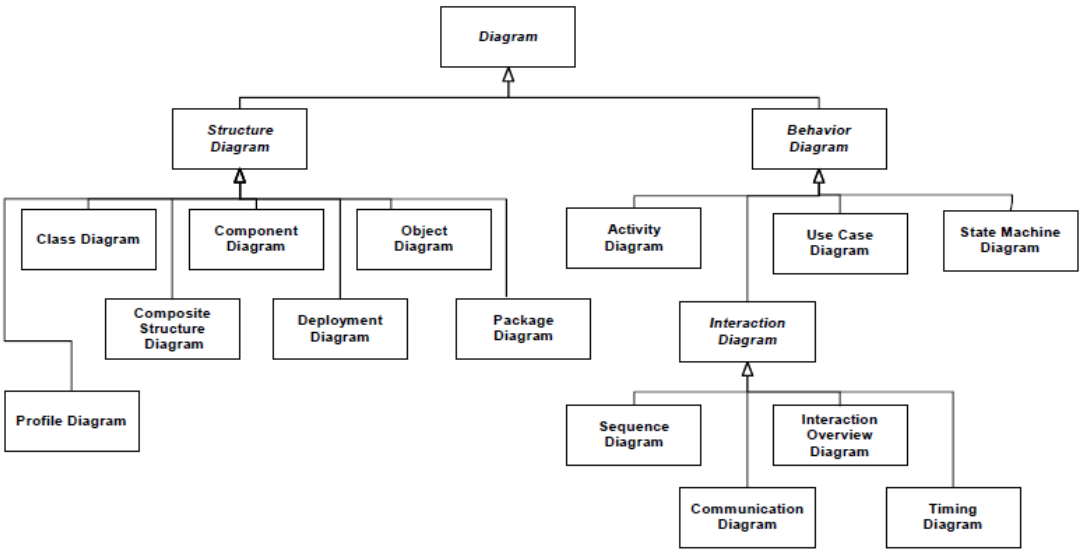

Figura 2.5: Taxonomía de diagramas de UML 2

Una de las características importantes que nos aporta UML es su capacidad de extensión. A través del uso de perfiles se puede dotar al lenguaje de mayor semántica y nuevos elementos de modelado, que lo adapten a nuevos entornos y campos de aplicación concretos.

Un perfil UML es un paquete que ofrece un mecanismo de extensión para la personalización de determinados modelos UML, proveyéndolo de una nueva semántica en sus elementos, adaptándolos a dominios y plataformas concretos.
En un perfil UML se definen principalmente estereotipos, máscaras que extienden a metaclases y que las dotan de nueva semántica, a través de nuevos atributos. También pueden definirse en un perfil nuevos tipos de datos y restricciones.

Acabamos de comentar que pueden existir perfiles UML para cualquier campo relacionado con la computación. Ejemplo de ello es que existen perfiles para:

- QoS (Calidad de servicio) y tolerancia a fallos [13].
- MARTE: Modelado y análisis de tiempo real para sistemas embebidos [14].
- CCM: Modelado de componentes CORBA [9].
- EAST-ADL: Modelado y desarrollo de sistemas electrónicos de automoción [15]

2.3. Objetivos de este proyecto

El objetivo de este proyecto es proponer una formulación y metodología de modelado, basada en la definición de un perfil UML, que permita describir sistemas basados en componentes utilizando únicamente elementos y diagramas de modelado UML 2.0. Con ello se dará soporte a todas las fases del proceso de desarrollo de aplicaciones basadas en componentes, en el cual también se incluye el proceso de desarrollo de componentes software reutilizables.

UML ofrece una gran variedad de elementos de modelado, por lo que deberán definirse y restringirse los elementos que sean más adecuados para describir una aplicación a lo largo de todas las fases del proceso de desarrollo, simplificando de esta manera la

interacción con el usuario, que únicamente deberá comprender y manejar los elementos que se definan en el perfil.

La metodología no está orientada a la descripción de la estructura interna de los componentes, sino al modelado de los metadatos que los componentes deben proporcionar con el objetivo de ser utilizados de forma opaca cuando son incluidos en una aplicación.

Los beneficios que se obtienen con los resultados de este trabajo son: por un lado, proponer una metodología de descripción de sistemas basados en componentes basada en una formulación general y actualmente conocida por cualquier desarrollador de software, como es UML; y por otro lado, hacer posible que el rico conjunto de herramientas de especificación, análisis y diseño basadas en UML que actualmente existen, se puedan también aplicar en el diseño de este tipo de aplicaciones.

3. Herramientas de diseño UML

Uno de los pasos iniciales del proyecto, fue la elección de la herramienta UML a utilizar para la definición del perfil y para el modelado de componentes y aplicaciones. Ya que el objetivo de nuestro proyecto era tanto la definición de un perfil como el modelado de una aplicación que lo utilizara, el criterio de búsqueda de la mejor herramienta UML en nuestras manos era el soporte de la herramienta para perfiles, puesto que el modelado de aplicaciones es una función que cualquier herramienta UML permite realizar.

La primera herramienta evaluada fue Altova UModel, creado [21] por Altova, en su versión más reciente Enterprise Edition v2010. Presenta un entorno de trabajo agradable, sencillo a la vista y muy intuitivo a la hora de realizar los diferentes modelos. Como en la mayoría de las herramientas UML, en este entorno podemos trabajar de dos formas de manera simultánea, la vista de árbol del proyecto o gráficamente en el modelo. Tiene la capacidad de crear todos los diagramas UML descritos en el apartado anterior, así como también diagramas SysML, o proyectos orientados a la generación de Schemas XML. Además de todos estos puntos positivos, podemos añadir que es una herramienta rápida y eficaz. No es nada pesada para el sistema y solventa los problemas velozmente. El único inconveniente de UModel es que en su versión actual, no permite definir asociaciones entre estereotipos en un perfil. Estas asociaciones son básicas en la creación de perfiles y, por tanto, esta restricción constituye un impedimento fundamental para su utilización.

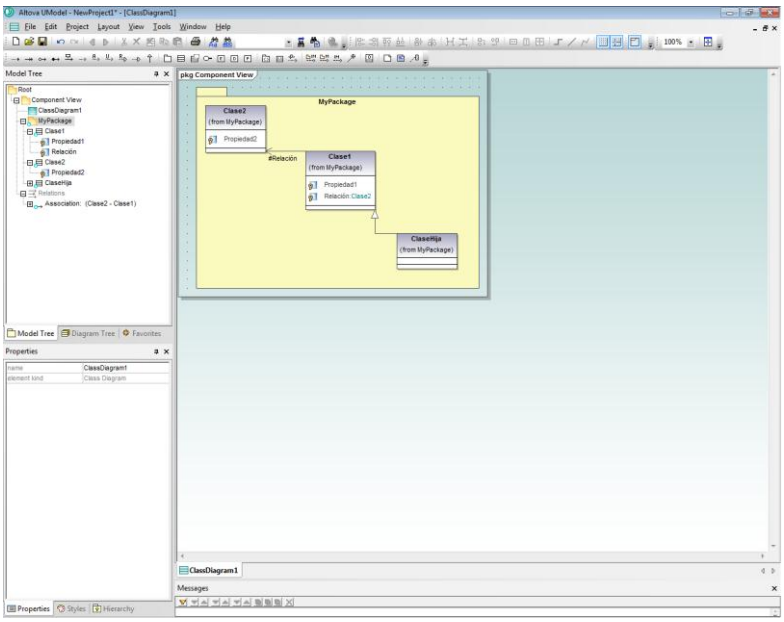

Figura 3.1: Entorno de trabajo Altova

La segunda herramienta descartada fue MagicDraw UML Personal Edition 16.9 [22] (su última versión es la 17.0). Es una herramienta desarrollada por la empresa No Magic y además de UML, también modela SysML. A primera vista, tiene un entorno de trabajo un tanto cargado y caótico, con demasiadas pestañas, pero es la herramienta más completa de todas las que hemos utilizado. El único inconveniente que se encontró en esta versión fue, la imposibilidad de importar modelos, esto es, importar un modelo en otro para hacer uso de elementos de modelado definidos en el primer modelo. Esta capacidad es básica en nuestro trabajo, pues cada componente puede ser definido en su propio modelo, haciendo uso de información común disponible en modelos externos.

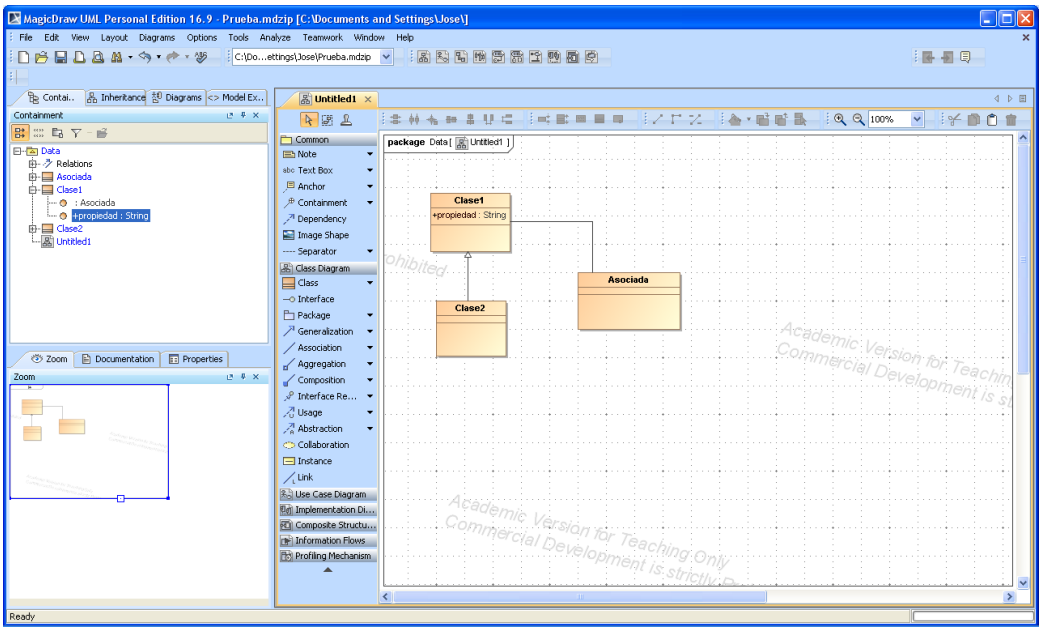

Figura 3.2: Entorno MagicDraw

La siguiente opción descartada fue Papyrus UML (Standalone version), implementado por el Comisionado francés para la Energía Atómica (CEA) [23]. Esta versión de Papyrus, aunque basada en Eclipse, no se apoya en él, es decir, es una versión completamente independiente de este entorno de desarrollo. Trabajamos con la versión 1.12, creada en 2009, por lo que queda un poco obsoleta respecto a la última especificación de UML. Como puntos positivos se puede destacar que, al igual que el Altova UModel, no produce una gran carga en el sistema y trabaja fluidamente (aunque no tanto como este primero). Además es un poco más vistoso que la versión sobre Eclipse, de la que hablaremos a continuación, a la hora de modelar. Sus aspectos negativos son dos. El primero es la falta de soporte para diagramas de despliegue, algo imprescindible en el desarrollo de aplicaciones basadas en componentes. El segundo aspecto es que los desarrolladores, junto a otras iniciativas Open Source, se unieron al Papyrus Project, para crear el add-on Papyrus para Eclipse, por lo que la versión standalone dejó de actualizarse.

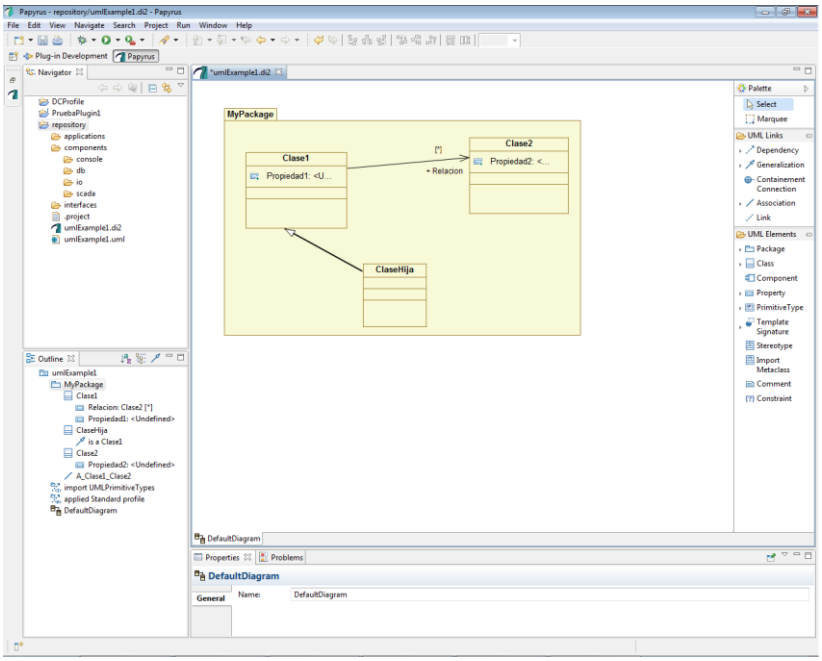

Figura 3.3: Entorno de trabajo de Papyrus UML

Por todas estas razones, finalmente se optó por utilizar Eclipse Indigo (v3.7) Modeling Tools con el add-on Papyrus, que se apoya en el plug-in UML2. A continuación analizamos esta herramienta.

3.1. Entorno Eclipse

Eclipse fue desarrollado inicialmente por IBM en 2001. Posteriormente, en 2004, tomó su relevo la Eclipse Foundation, que lo comenzó a distribuir como software libre.
Este entorno de desarrollo se caracteriza por funcionar a base de módulos o plug-ins que enriquecen su funcionalidad, a diferencia de otras herramientas que instalan su entorno como un todo. De esta forma, el cliente trabaja exclusivamente con lo que necesita, olvidándose de módulos de más y mayor tiempo de carga y trabajo.
Para nuestro proyecto hemos utilizado una versión beta de Eclipse Indigo (v3.7), cuya versión estable se lanzó a finales de junio de 2011, trabajando con ella en los últimos compases del proyecto. Además, utilizamos una versión especial de Eclipse denominada Eclipse MDT (Model Development Tools), especializada en las transformaciones de modelos. Constituye uno de los grandes proyectos existentes actualmente para este entorno de trabajo, llegando a tener una distribución propia.

El add-on Papyrus trabaja sobre el plug-in UML2 de Eclipse, que implementa el metamodelo UML 2.x englobado en el proyecto MDT. UML2 permite trabajar completamente con la especificación UML 2.x de OMG, pudiéndose llegar a realizar proyectos UML completos sin necesidad de ninguna herramienta de modelado gráfico, aunque puede ser demasiado engorroso y difícil de manejar. Por esta razón entra en juego Papyrus, que complementa UML2 dotándolo de soporte gráfico y permitiendo manejar este plug-in de una forma más cómoda y rápida.

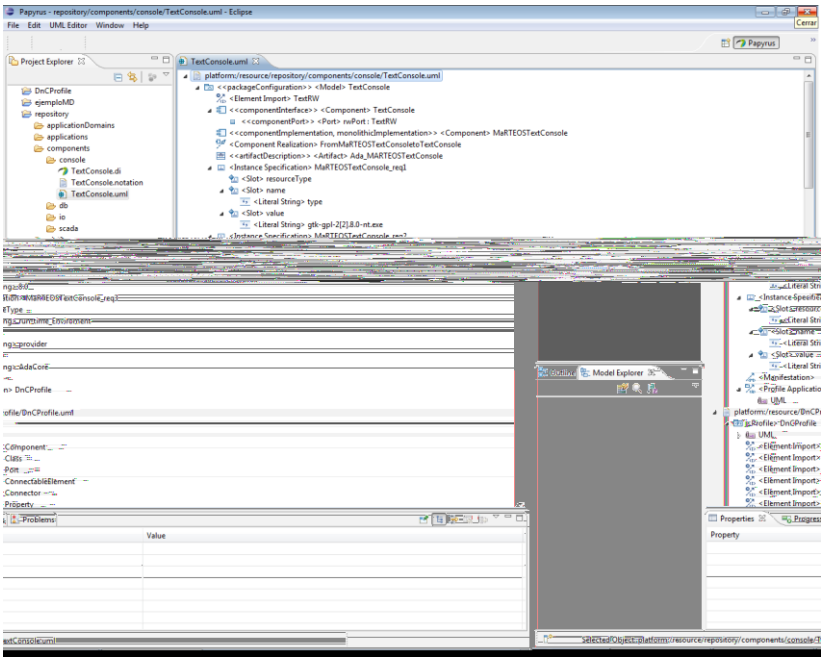

Figura 3.4: Entorno Eclipse con árbol UML2

Tanto UML2 como Papyrus tienen unos entornos de trabajo bastante pobres. El primero posee dos editores. Uno que presenta el modelo en forma de árbol, mostrando todos sus elementos, incluyendo sus propiedades, funciones, etc. El otro trabaja de modo textual (formato XMI), escribiéndose el código UML mediante el teclado, lo cual resulta bastante engorroso y, a veces, imposible, ya que cada elemento tiene una identificación no arbitraria que el cliente por sí mismo no puede asignar.

Papyrus trabaja sobre un canvas blanco en el que el usuario va añadiendo los elementos que vaya precisando en su proyecto. Estos elementos (clases, componentes, nodos…), por defecto, son cajas blancas con bordes negros y su estereotipo escrito en ellas, que pueden colorearse gracias a los menús que proporciona. Tiene numerosas vistas personalizables, gracias a las ventanas desplegables, que hacen el trabajo un poco más intuitivo, aunque generalmente las que más se utilizan, pudiéndose cambiar mediante pestañas, son Project y Model Explorer, Properties y el propio escritorio donde se modela el proyecto.

Otro de los aspectos negativos de Papyrus es la falta de elementos concretos que llegan a ser importantes a la hora de realizar el trabajo, como son los diagramas de despliegue, al igual que ocurría con su versión Standalone. Este problema surgió con la primera versión de Papyrus que fue descargada, en febrero de 2011, que no soportaba este tipo de diagramas, por lo que se descargaron las "*Nightly Builds*", versiones que los desarrolladores cuelgan en la página de descargas del Proyecto Papyrus en la web oficial de Eclipse casi a diario. Esto supuso un nuevo problema, ya que las nuevas versiones necesitaban como motor el todavía no lanzado Eclipse Indigo, por lo que fue necesario ponerse en contacto con el equipo de desarrollo del proyecto, que atendió nuestras consultas y nos sugirió un enlace para la versión beta. Gracias a las "*Nightly Builds*" y, posteriormente, a la versión estable publicada en junio hemos podido concluir los trabajos de diseño del perfil y aplicación sobre una aplicación en particular, no sin sortear otros problemas como la falta de algunos elementos de modelado gráfico, como los Communication Paths para enlazar nodos, entre otros…

El modo en el que Papyrus y UML2 se relacionan es que Papyrus implementa de forma gráfica los elementos de UML2, de tal forma que cuando se crea un nuevo modelo Papyrus, se crea automáticamente su correspondiente modelo de soporte UML2. De igual modo, cualquier elemento que se añade al modelo creado por Papyrus se añade al modelo UML creado por UML2. De esta forma, si algún elemento no tiene soporte de forma gráfica en Papyrus, se puede completar el modelo UML2 directamente.

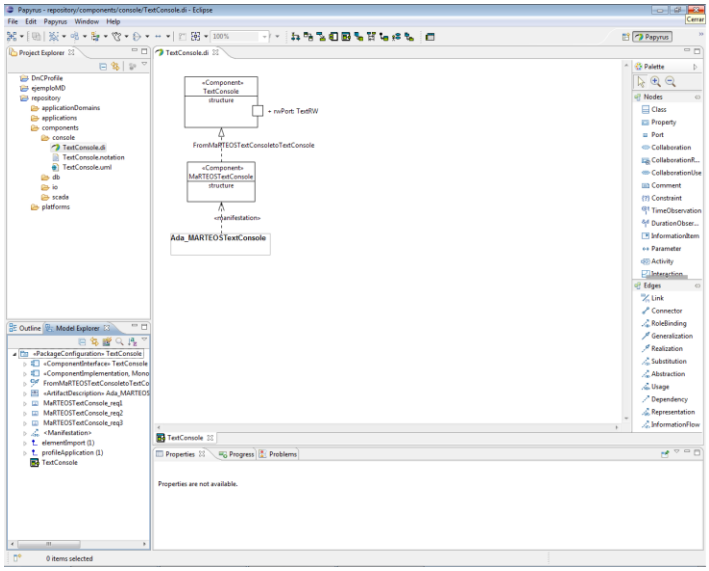

Figura 3.5: Entorno Eclipse con Papyrus

4. Perfil DnCProfile

El perfil diseñado en este proyecto corresponde al metamodelo expuesto en la especificación "*Deployment and Configuration of Component-based Distributed Applications Specification*", presentada en el apartado 2.2 de esta memoria, intentando implementarla lo más fielmente posible a la realidad.

En primer lugar, se definen unos atributos estándar que aparecen en todos, o casi todos, los elementos de nuestro perfil y que se usan por cuestiones de identificación y anotación con respecto a otros elementos de similares características:

- *label*: String que da una pequeña explicación de lo que es y lo que hace el elemento al que etiqueta. Aunque se trata de un atributo opcional, es muy útil, pues le da al usuario información básica para conocer los elementos con los que trata en cada momento.
- *UUID*: String cuyo fin es el identificar unívocamente el elemento al que pertenece.

Como se observa en la figura 4.1 el perfil se estructura en tres paquetes, donde se agrupan los estereotipos definidos para cada uno de los tres tipos de modelos de datos que conforman la especificación D&C (y que fueron presentados anteriormente). Cada estereotipo extiende a la metaclase que se ha estimado más oportuna en cada caso.

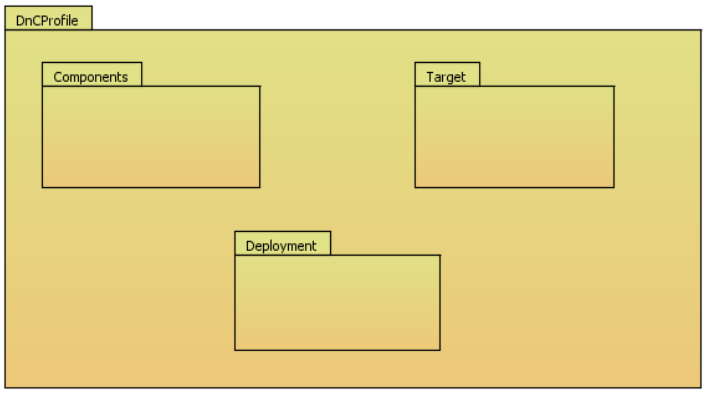

Figura 4.1: Estructura del perfil DnCProfile

4.1. Paquete Components

En el paquete Components se agrupan los estereotipos que se van utilizar durante las fases del proceso de desarrollo de componentes software reutilizables. A continuación describiremos cuáles son estos estereotipos.

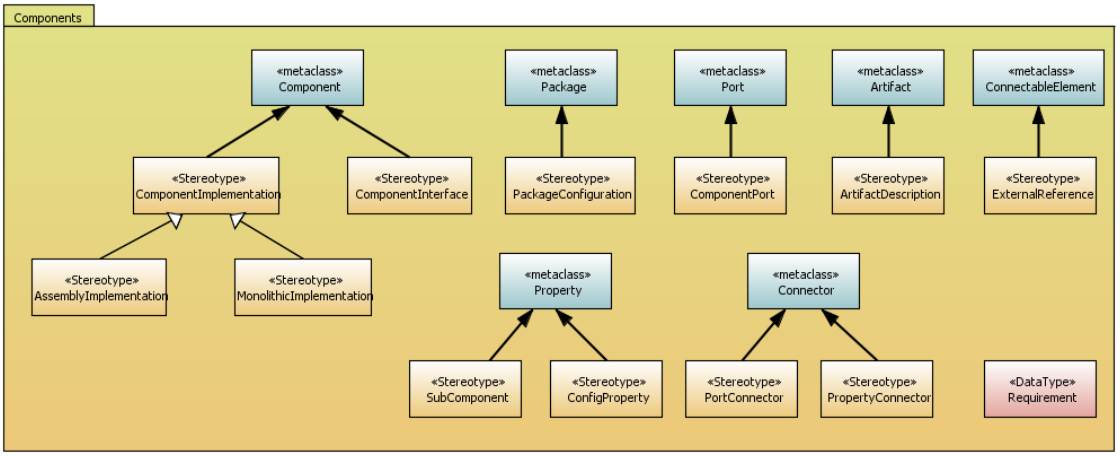

Figura 4.2: Estereotipos definidos en el paquete Components

ComponentInterface:

Este estereotipo identifica una especificación (o interfaz) de componente, esto es, un tipo de componente. Como un tipo de componente se define en base a sus propiedades y puertos, este estereotipo extiende a la metaclase UML Component, que da soporte a la definición de ambos tipos de elementos. Como se observa en la figura 4.3, cada estereotipo <<ComponentInterface>> define, además de los atributos *label* y *UUID*, las siguientes asociaciones:

- *configProperty* : ConfigProperty [0..*] => Conjunto de propiedades configurables del componente
- *ownedPort* : ComponentPort [0..*] => Conjunto de puertos que pertenecen a la interfaz de componente.

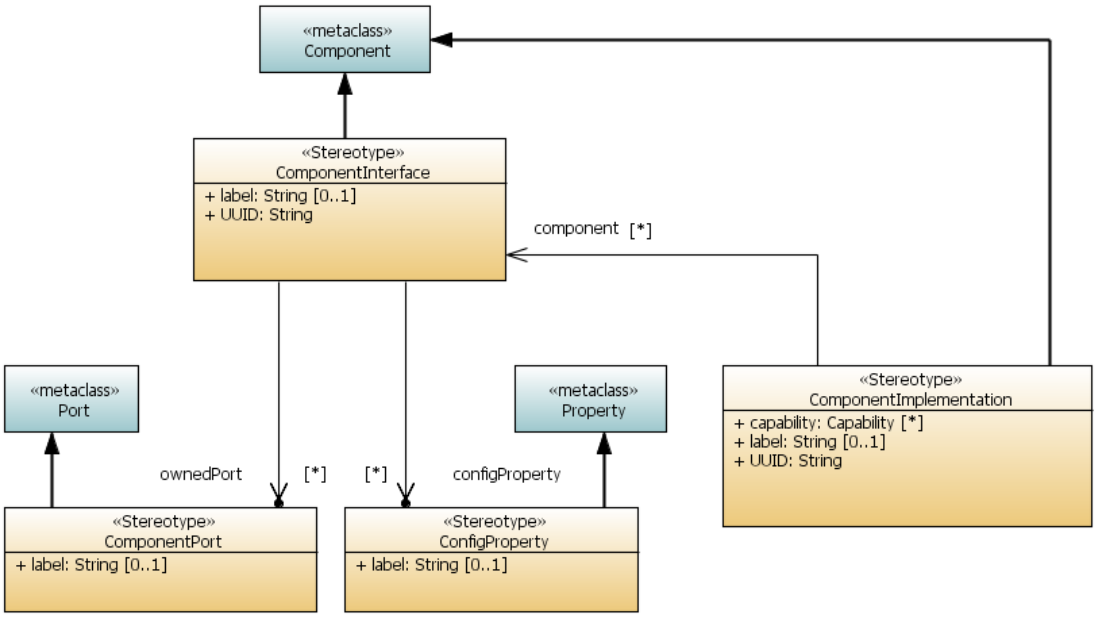

Figura 4.3: Estereotipos relacionados con <<ComponentInterface>>

ComponentPort:

De la metaclase UML Port se extiende un estereotipo <<ComponentPort>> que será utilizado para especificar el estereotipo de todos los puertos que se quieran hacer

públicos, de entre los que estén contenidos en un elemento estereotipado como <<ComponentInterface>>. No tiene asociaciones adicionales. Existe una restricción sobre este tipo de elementos, y es que los puertos que se estereotipen como <<ComponentPort>> pueden tener naturaleza de puerto ofertado o requerido, pero no ambas a la vez.

ConfigProperty:
El estereotipo <<ConfigProperty>> se extiende de la metaclase UML Property y hace referencia a todas aquellas propiedades de configuración que caracterizan y que están contenidas en una <<ComponentInterface>>. Para instanciar un componente es necesario asignar un valor a todas las propiedades de configuración definidas en su correspondiente interfaz.

ComponentImplementation:
Este estereotipo identifica una implementación específica de una interfaz de componente. Corresponde con el elemento *ComponentImplementationDescription*. Considerando que una implementación de componente se puede representar en el caso más general a través de otro componente (pues puede corresponder a un ensamblado), el estereotipo extiende de nuevo de la metaclase Component de UML (aunque visto a un nivel de abstracción inferior al del caso anterior). Este estereotipo tiene como asociaciones:

- *capability* : Capability [0..*] => Utilizado para diferenciar y discriminar entre las diferentes implementaciones de una misma interfaz de componente de forma que se elija la más correcta o adecuada al sistema desarrollado.

Todo elemento estereotipado como <<ComponentImplementation>> tiene que tener una asociación de tipo UML Realization con un elemento estereotipado como <<ComponentInterface>>.

Datatype Capability:
Los objetos de tipo Capability sirven para describir las capacidades de una implementación, que serán contrastadas con ciertos requisitos de selección para escoger las implementaciones de componente más adecuadas. Sus atributos son, además de una etiqueta:

- *resourceType* : String => Da a conocer el tipo de recurso al que se refiere la capacidad.
- *name* : String => Identifica una de las propiedades del tipo de recurso anterior.
- *value* : String => Valor que se le otorga a la propiedad anterior en la implementación del componente (este es el valor que deberá contrastarse con el correspondiente requisito de selección).

En este caso, se podría cambiar los strings por listas enumeradas que, dependiendo de la metodología en la que se esté desarrollando la aplicación, alberguen una "biblioteca" de recursos en los que el desarrollador del componente pueda basarse para completar las capacidades para contrastar con los requisitos.

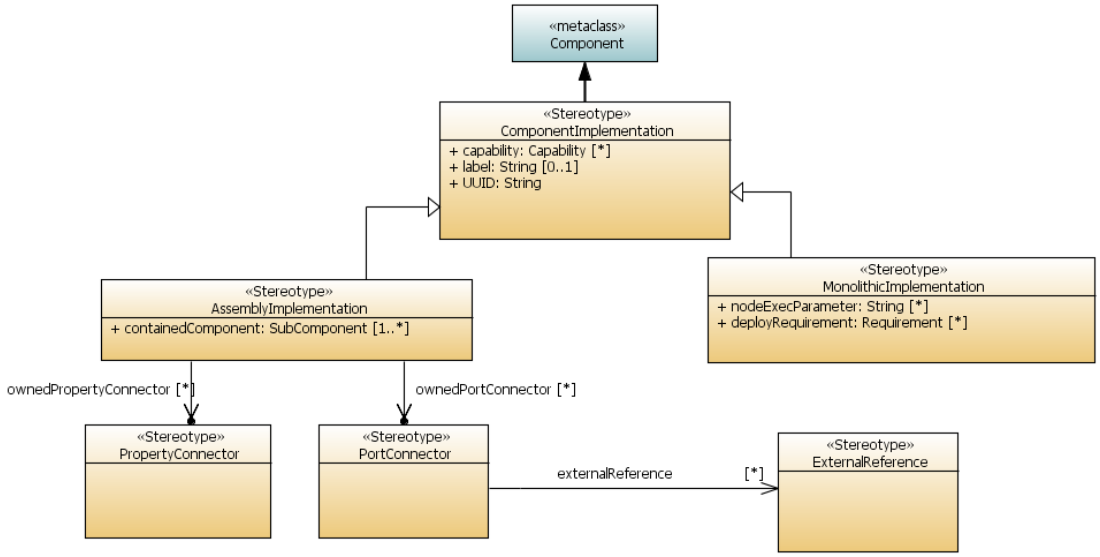

Figura 4.4: Estereotipos relacionados con <<ComponentImplementation>>

MonolithicImplementation:
Este estereotipo es una generalización de <<ComponentImplementation>> y, por tanto, extiende a la metaclase Component. Sirve para identificar una implementación monolítica de una interfaz de componente. Añade los siguientes atributos a los que hereda de <<ComponentImplementation>>:

- *deployRequirement* : Requirement [1..*] => Los *Requirement* son unas propiedades especiales utilizadas a la hora de realizar el despliegue de una aplicación. Se comparan con los recursos de la plataforma para verificar si el componente es compatible con ella y, por tanto, puede ser instalado en ella.
- *nodeExecParameter* : String => Parámetros de ejecución para el entorno de plataforma, utilizados por las herramientas de despliegue para instanciar el componente.

Datatype Requirement:
El tipo de dato Requirement, un tipo de dato no definido en UML, sirve para cualificar los requisitos que una implementación impone sobre una plataforma para poder ser ejecutada en ella. Tiene tres atributos:

- *resourceType* : String => Indica el tipo de recurso sobre el cual se aplica que requisito.
- *name* : String => Identifica una de las propiedades del recurso anterior, sobre la que se aplica el requisito.
- *value* : String => Indica el valor que se le exige a la propiedad anterior.

AssemblyImplementation:
Este estereotipo es una generalización de <<ComponentImplementation>> y por tanto, extiende a la metaclase UML Component. Sirve para identificar una implementación de una interfaz de componente construida por composición de instancias de otros componentes. Añade a los atributos que hereda de <<ComponentImplementation>>, los siguientes:

- *containedComponent* : SubComponent [1..*] => Describe los subcomponentes de los que se compone el ensamblado. Hay que recordar aquí que a este nivel, los subcomponentes se identifican sólo por la interfaz de componente que realizan.

- *ownedPortConnector* : PortConnector [0..*] => Describe el conjunto de PortConnectors que posee el ensamblado.
- *ownedPropertyConnector* : PropertyConnector [0..*] => Describe el conjunto de PropertyConnectors que posee el ensamblado.

SubComponent:
El estereotipo <<SubComponent>> es una extensión de la metaclase UML Property. Se encargará de identificar todos aquellas instancias que formen parte de un ensamblado estereotipado como <<AssemblyImplementation>>.

PortConnector:
De la metaclase Connector se extiende el estereotipo <<PortConnector>>, encargado de identificar las interconexiones entre puertos de subcomponentes dentro de un ensamblado. Además de su label, tiene dos relaciones:
- *connectedPort* : ComponentPort [1..*] => Son el conjunto de puertos que interconecta.

PropertyConnector:
Extiende también a la metaclase UML Connector. Identifica a los conectores que se utilizan para mapear las propiedades externas del componente implementado por el ensamblado respecto a las propiedades de las instancias de sus subcomponentes. Además de su label presenta una relación:
- *connectedProperty* : ConfigProperty [1..*] => Conecta la propiedad de un subcomponente con una propiedad del ensamblado.

PackageConfiguration:
Corresponde al metadato homónimo de la especificación, extiende a la metaclase UML Package y servirá, gracias a sus atributos *label* y *UUID* para identificar el paquete que contiene toda la información relativa a un tipo concreto de componente. A través del *UUID* del paquete se podrían realizar búsquedas de componentes en el repositorio. Tiene una única relación que sirve como restricción:
- *componentInterface* : ComponentInterface => Indica la única interfaz de componente que puede ser empaquetado por una <<PackageConfiguration>>, independientemente de las implementaciones que pueda tener.

ArtifactDescription:
Otro elemento importante a la hora de realizar el modelado de componentes son las <<ArtifactDescription>>, que no aparecen en la especificación del OMG. Por ello era de vital importancia crear este estereotipo, que extiende la metaclase Artifact. En un principio, aparte de la *label* y el *UUID*, se añadió a los atributos una *source*, pero que rápidamente se eliminó al comprobar que el propio artifact de UML podía albergar esa opción. La *source* es la dirección física donde se localiza el archivo de código del artifact.

4.2. Paquete Target

En el paquete Target se agrupan todos los estereotipos que se utilizan en la creación de un modelo de datos de plataforma. A continuación describiremos estos estereotipos.

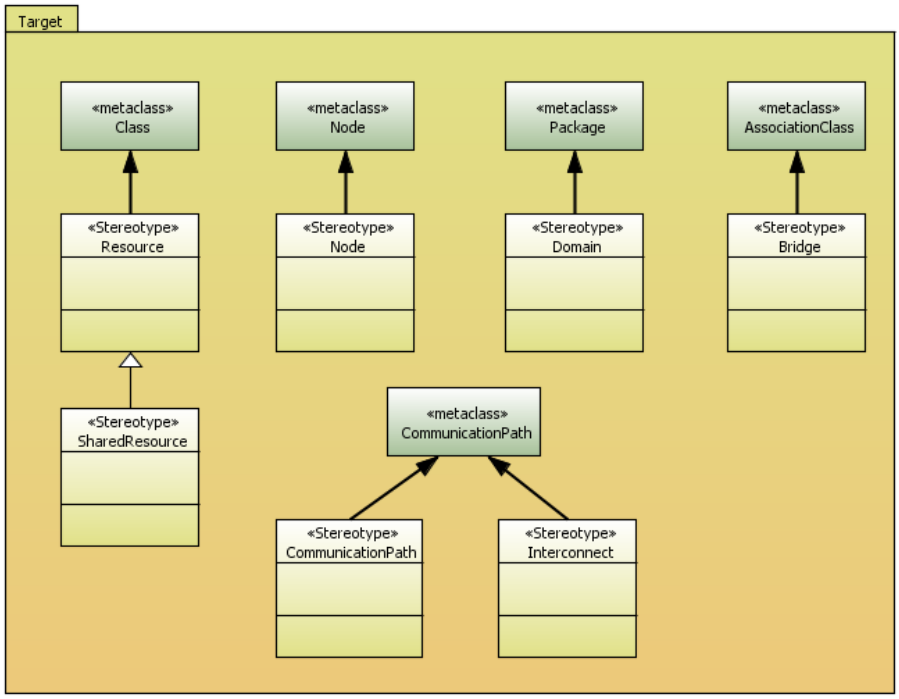

Figura 4.5: Estereotipos definidos en el paquete Target

Domain:

Existe un equivalente al <<PackageConfiguration>> del modelo de componentes. El estereotipo <<Domain>>, que también extiende a la metaclase UML Package, representa o describe todo el entorno de plataforma, y está definido por un conjunto de nodos, las comunicaciones entre éstos y los recursos que comparten. No tiene atributos extra aparte de su *label* y su *UUID*. Y sus asociaciones son con los elementos que contiene:

- *containedNode* : Node [1..*] => Nodos que pertenecen a ese dominio.
- *containedCommunicationPath* : CommunicationPath [0..*] => CommunicationPaths que proporcionan conexión entre los diferentes nodos.
- *domainResource* : SharedResource [0..*] => Recursos compartidos que pertenecen al dominio y que por tanto pueden ser compartidos por los diferentes nodos y redes de comunicación.
- *containedInterconnect* : Interconnect [0..*] => Conjunto de Interconnects que conectan nodos.

Node:

El elemento más importante del modelo de datos de plataforma es el estereotipo <<Node>>, que extiende a la metaclase UML del mismo nombre. Los nodos permiten que los componentes sean instanciados y se comuniquen entre ellos mediante los *CommunicationPaths*. Sus asociaciones son con los *Resource* y *SharedResource* que lo caracterizan y con los *CommunicationPath* e *Interconnect* que utiliza para comunicarse con otros nodos:

- *nodeConnector* : Interconnect [0..*] => Conjunto de Interconnects a los que el nodo está conectado.
- *communicationPath* : CommunicationPath [0..*] => Conjunto de CommunicationPaths a los que el nodo está conectado.
- *ownedResource* : Resource [0..*] => Conjunto de recursos propios para ese nodo.

- *availableSharedResource* : SharedResource [0..*] => Conjunto de recursos compartidos a los que ese nodo tiene acceso.

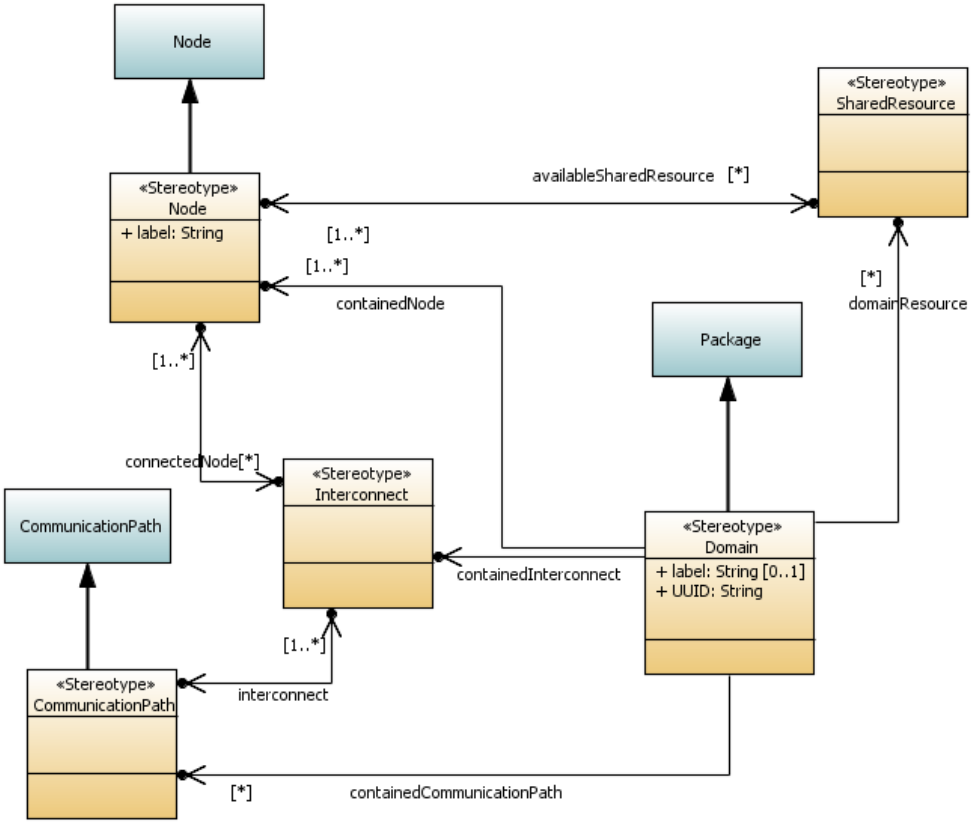

Figura 4.6: Estereotipos relacionados con <<Domain>>

Interconnect:
De la misma metaclase se extiende es estereotipo <<Interconnect>>, un mecanismo de comunicación directa entre nodos. Tampoco tiene atributos, pero sí asociaciones, con Node, Bridge, CommunicationPath y con Resource, ya que puede tener recursos, aunque nunca son compartidos:

- *connectedNode* : Node [1..*] => Es el conjunto de nodos a los que el Interconnect está conectado.
- *bridge* : Bridge [0..*] => Contiene los Bridges que proporcionan conectividad a otros Interconnects.
- *ownedResource* : Resource [0..*] => Contiene el conjunto de recursos propios del Interconnect.
- *communicationPath* : CommunicationPath => Referencia al CommunicationPath al que pertenece este Interconnect.

CommunicationPath:
Los nodos se comunican entre sí mediante, generalmente, <<CommunicationPaths>>. Este estereotipo extiende la metaclase UML de su mismo nombre. Puede estar compuesto por al menos un interconnect y por cero o más bridges. No tiene atributos adicionales pero sí asociaciones:

- *interconnect* : Interconnect [1..*] => Conjunto de *Interconnects* contenidos en ese *CommunicationPath*.

- *bridge* : Bridge [0..*] => Es el conjunto de *Bridges* contenidos en ese *CommunicationPath*.
- /*connectedNode* : Node [0..*] => Contiene el conjunto de nodos que están conectados mediante este CommunicationPath.

Bridge:
El estereotipo <<Bridge>> se utiliza para aquellas conexiones entre interconnects que proporcionan una comunicación indirecta entre nodos. No tiene atributos extra, pero sí tres relaciones:

- *interconnect* : Interconnect [1..*] => Son los *Interconnects* a los que el Bridge proporciona conectividad.
- *ownedResource* : Resource [0..*] => Es el conjunto de recursos propios del Bridge.
- *communicationPath* : CommunicationPath [1..1] => Referencia al *CommunicationPath* al que pertenece el *Interconnect*.

Resource y SharedResource:
Existen dos estereotipos que extienden a la metaclase UML Class: <<Resource>> y <<SharedResource>> (que a su vez es una especialización del anterior). <<Resource>> tiene un atributo *resourceType*, una secuencia de strings que indican el tipo de recurso, *name*, que indica el nombre de dicho recurso y *value*. Estos elementos se utilizan para caracterizar la plataforma y se comparan con los requisitos de las implementaciones de los componentes en el despliegue. <<SharedResource>>, además del atributo que hereda de Resource, tiene una asociación extra con uno o más nodos:

- *resourceUser* : Node [1..*] => Recoge todos los nodos que tienen acceso a ese recurso compartido.

4.3. Paquete Deployment

Por último, tratamos el paquete Deployment, en el que se agrupan todos los estereotipos involucrados en el despliegue de una aplicación distribuida basada en componentes. Estos estereotipos son:

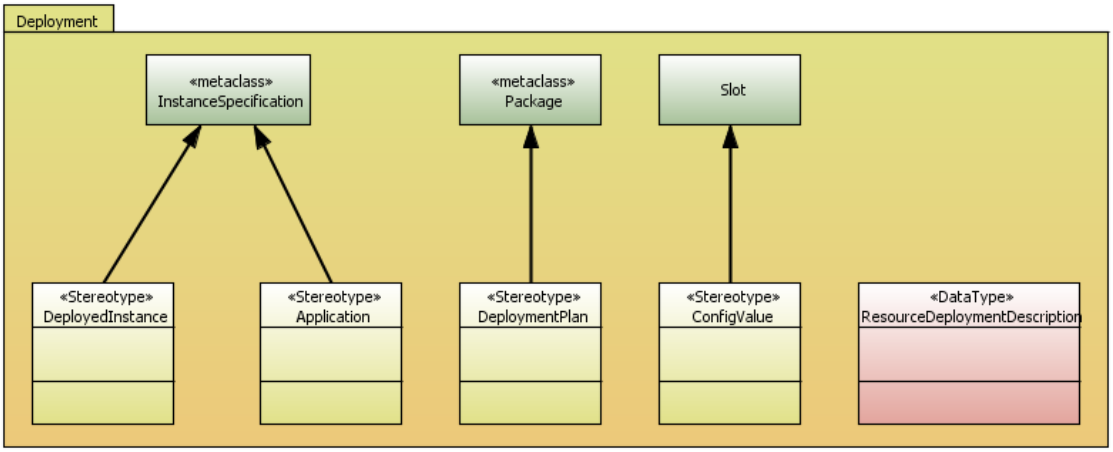

Figura 4.7: Estereotipos definidos en el paquete Deployment

DeploymentPlan:
Al igual que <<PackageConfiguration>> y <<Domain>>, el despliegue tiene una extensión de la metaclase UML Package para definir el paquete en el que se define el modelo de despliegue de la aplicación. Es el estereotipo <<DeploymentPlan>> y, al igual que los anteriormente nombrados, tiene como atributos una *label* y un *UUID* que lo identifican. Contiene dos asociaciones:

- *deployedApplication* : Application => Aplicación final que va a ser desplegada.
- *domain* : Domain => Plataforma sobre la que se despliega la aplicación.

Application:
El estereotipo <<Application>> extiende a la metaclase UML InstanceSpecification e identifica la aplicación que se va a desplegar. Aplicando el principio de recursividad del que se habló en la introducción por el cual una aplicación es un ensamblado de componentes, este estereotipo se le aplicará a instancias de elementos que en el modelo de componentes fueron identificados como <<AssemblyImplementation>>. Posee dos asociaciones:

- *configValues* : ConfigValue [0..*] => Conjunto de valores que se le asignan a las propiedades de configuración globales de la aplicación.
- *deployedInstances* : DeployedInstance [1..*] => Conjunto de instancias de implementaciones concretas de componentes que forman la aplicación concreta desplegada.

DeployedInstance:
<<DeployedInstance>> extiende también a InstanceSpecification y especificará el estereotipo de aquellas instancias de <<MonolithicImplementation>> que van a ser incluidas en el plan de despliegue, esto es, identifica las instancias de implementaciones concretas que forman la aplicación. Contiene un atributo múltiple, *deployedResource*, de elementos de tipo *ResourceDeploymentDescription* que sirve como guía a las herramientas que se encarguen de contrastar los requisitos de la implementación con los recursos de los correspondientes nodos.

ConfigValue:
El estereotipo <<ConfigValue>> extiende a la metaclase Slot. Se asignará este estereotipo a aquellas ranuras o slots de la instancia que se utilicen para asignar valor a las propiedades de configuración de una <<Application>> o de un <<DeployedInstance>>.

Datatype ResourceDeploymentDescription:
Como se ha comentado antes, este tipo de datos contiene la información necesaria para que se comparen los requisitos y los recursos que se van a desplegar, mediante sus dos atributos: *requirementName* y *resourceName*, respectivamente. Mapean los requisitos de los componentes, referenciados a través de *requirementName*, con el recurso que debe satisfacerlos, identificado a través de *resourceName*.

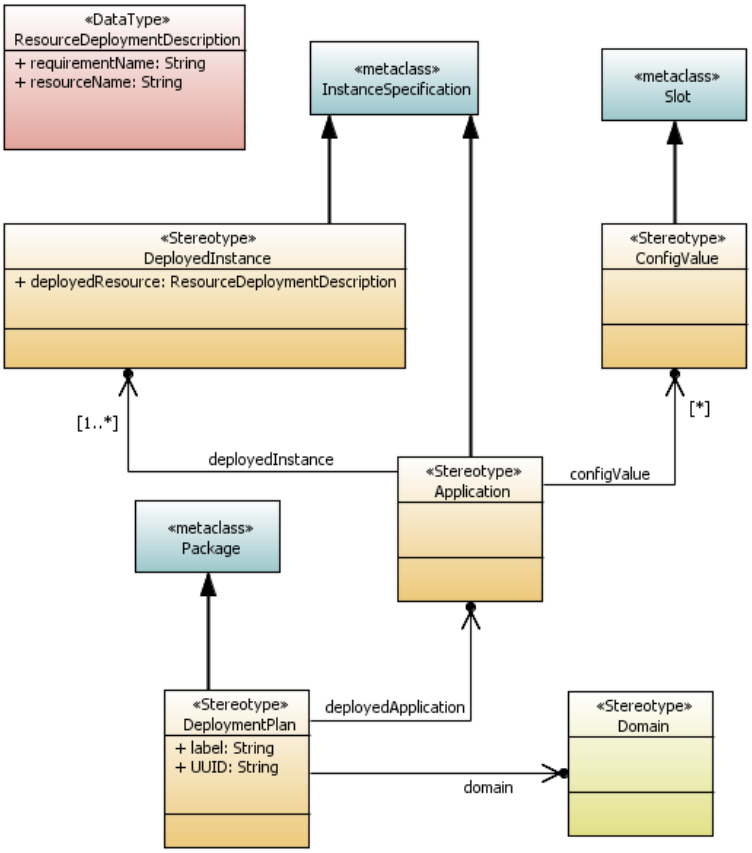

Figura 4.8: Estereotipos relacionados con <<DeploymentPlan>>

5. Metodología de desarrollo de aplicaciones basadas en componentes en UML2

5.1. Organización del entorno de desarrollo

El objetivo final en el que se enmarca este proyecto es el de generar un entorno integrado de desarrollo de aplicaciones basadas en componentes basado en UML. Con el propósito de facilitar la integración con herramientas MDA, el entorno estará soportado por una plataforma Eclipse. Los procesos de diseño y desarrollo tanto de componentes como de aplicaciones se realizarán en el entorno de desarrollo asistidos por herramientas que garanticen la validez "por construcción" de los artefactos que se generan.

Dicho entorno está constituido por dos elementos básicos: el repositorio en el que se organizan los productos generados, tanto intermedios como finales, y las herramientas que realizan las transformaciones entre ellos.

Toda la información asociada a los componentes disponibles, provengan estos de nuevos desarrollos o de adquisiciones de componentes desarrollados por terceros, se registran en el repositorio. Asimismo, en el repositorio existe almacenada información sobre las plataformas de ejecución especificadas, y sobre aplicaciones disponibles ya desarrolladas y dispuestas para su ejecución. Para facilitar el desarrollo de las herramientas de transformación que manejan dichos componentes, así como la importación y exportación de información, se ha definido una estructura fija para el repositorio, que se muestra en la figura 5.1.

Esta estructura divide el proyecto de la aplicación en cuatro apartados muy diferenciados (cada uno de ellos correspondiente a una carpeta en el repositorio):

- *applicationDomains*: en el que se almacenan los modelos con todas las interfaces que pueden utilizarse en la definición de componentes, así como los tipos de datos utilizados en ellas, estructuradas por dominios de aplicación.
- *applications*: en el que se almacenan los modelos de despliegue de las aplicaciones, así como sus propias definiciones como ensamblados de componentes.
- *components*: donde se almacenan todos los modelos de componentes reutilizables disponibles, estructurados también por dominios de aplicación.
- *platforms*: en el que se almacenan los modelos concernientes a las plataformas disponibles para desplegar aplicaciones.

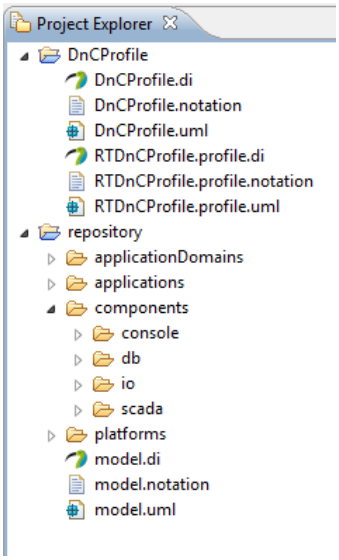

Figura 5.1: Estructura del repositorio en el workspace

5.2. Utilización del perfil

5.2.1. Importación del perfil

Para utilizar el perfil DnCProfile en los modelos UML a desarrollar, es necesario importarlo primero a nuestro *workspace*, o directorio de trabajo en Eclipse (el mismo en el que se encuentra instalado el repositorio). Para ello hay dos vías igual de sencillas.
La primera es mediante la barra de menús desplegables, pinchamos en Archivo / Import…, o con el botón derecho en la vista Project Explorer del propio entorno Eclipse.

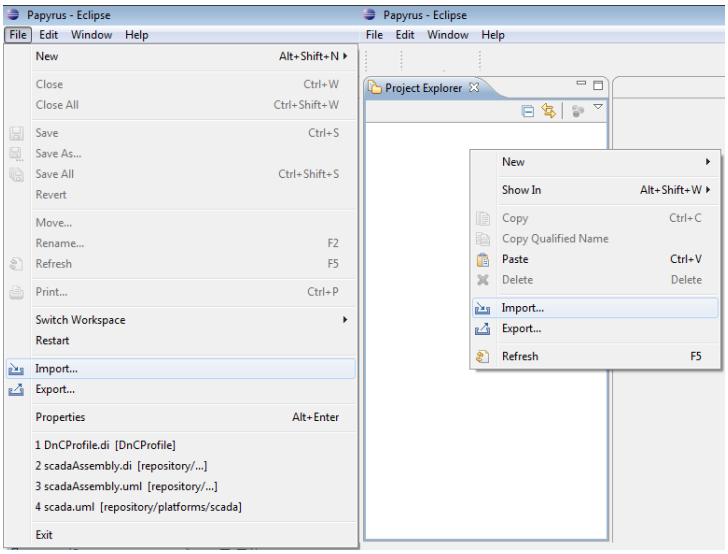

Figura 5.2: Métodos de importación

Tras pulsar en la opción Import… aparecerá una nueva ventana que indica el tipo de importación que queremos hacer:

- *Archive File*: un archivo donde está comprimido todo aquello que queremos importar. Su extensión ha de ser: *.jar, *.zip, *.tar, *.tar.gz o *.tgz, nunca *.rar.
- *Existing Projects into Workspace*: proyectos del propio *workspace* que no están en el Project Explorer, bien porque se hayan borrado de este o porque todavía no se han cargado.
- *File System*: un sistema de archivos o directorio localizado en cualquier parte del disco duro o discos extraíbles que se importa tal cual al workspace en el que se trabaja.

Elegimos la opción más acorde a nuestras necesidades, generalmente el perfil vendrá dado en un archivo comprimido, por lo que escogeremos *Archive File*, o *File System* si ya está descomprimido y, a continuación, presionamos Next >. Aparecerá una nueva ventana, bastante genérica para cualquiera de las cuatro opciones que se han presentado. En ella se escoge el archivo o directorio a importar, presionando en el primer Browse… (From…). Inmediatamente aparecerán los directorios que contiene y los archivos dentro de un directorio al pinchar sobre uno de ellos. Seleccionamos el directorio o archivos que necesitamos y pinchamos en el segundo Browse… que nos indicará la futura ubicación de los nuevos archivos en nuestro *workspace*. Terminamos el import pinchando en Finish.

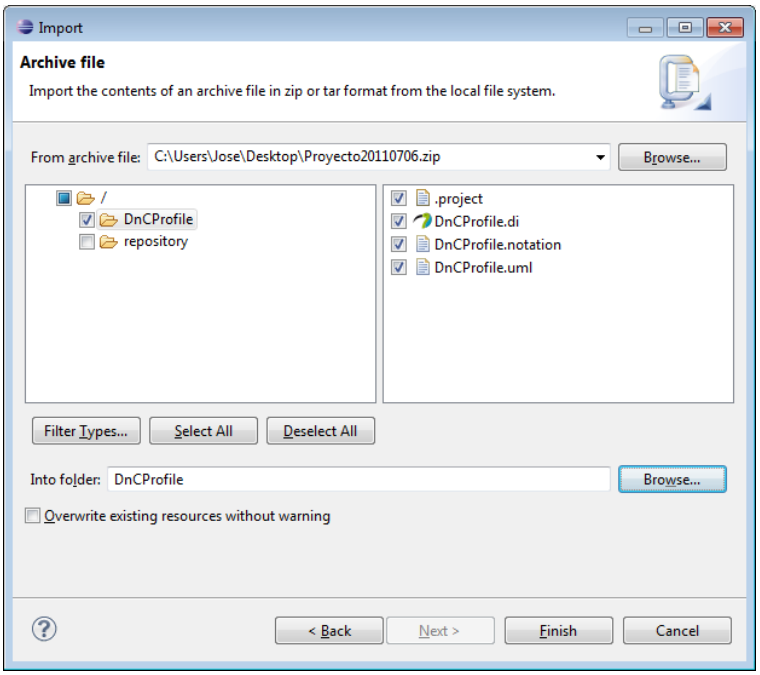

Figura 5.3: Archive file del Import

Un nuevo directorio que contiene tres archivos llamado DnCProfile aparece en el Project Explorer. Estos tres archivos son:

- DnCProfile.di: Archivo Papyrus con la representación gráfica del perfil.
- DnCProfile.notation: Archivo de anotaciones para el *.di.
- DnCProfile.uml: Archivo UML2, que constituye el fichero más importante al contener los elementos definidos en el perfil, esto es, representa el perfil en sí mismo.

5.2.2. Aplicación del perfil a un modelo

Cuando se trabaja directamente con UML2, para cargar el perfil dentro de un modelo propio, creado por nosotros o importado a nuestro *workspace*, desplegamos el menú UML Editor y pinchamos en Load resource. En la ventana emergente pinchamos en Browse Workspace (puesto que ya tenemos cargado el perfil en nuestro directorio de trabajo) y seleccionamos el archivo *.uml dentro del directorio que hemos importado.

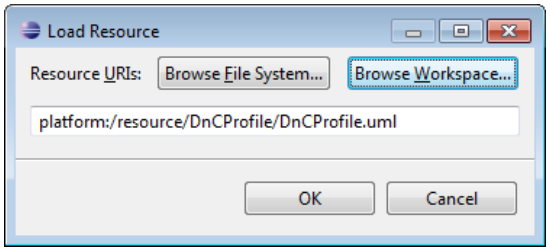

Figura 5.4: Importar perfil en UML2

Cargar el perfil DnCProfile desde Papyrus es más sencillo. Se abre el archivo *.di correspondiente al modelo con el que queremos trabajar.
Seleccionado el paquete raíz del modelo en el Model Explorer, pinchamos en la pestaña Properties y, seguidamente, en Profile. Aparecerán nuevas opciones para añadir un perfil. Hacemos click en la cruz verde (Apply profile…), creándose un nuevo diálogo en el que seleccionaremos el archivo *.uml de nuestro perfil y, en una nueva ventana, el perfil en cuestión.

5.2.3. Asignación de un estereotipo a un elemento

Con el perfil ya cargado en nuestro modelo de UML2 (debería de aparecer como una de las, al menos, dos líneas desplegables en la vista de trabajo), el siguiente paso es aplicar un estereotipo a un elemento. Para ello, en la barra desplegable de tareas elegimos el menú UML Editor / Element / Apply Stereotype. A continuación aparecerá una nueva ventana con la lista de los estereotipos que podemos aplicar a ese elemento, que se corresponden a la metaclase a la que pertenece.

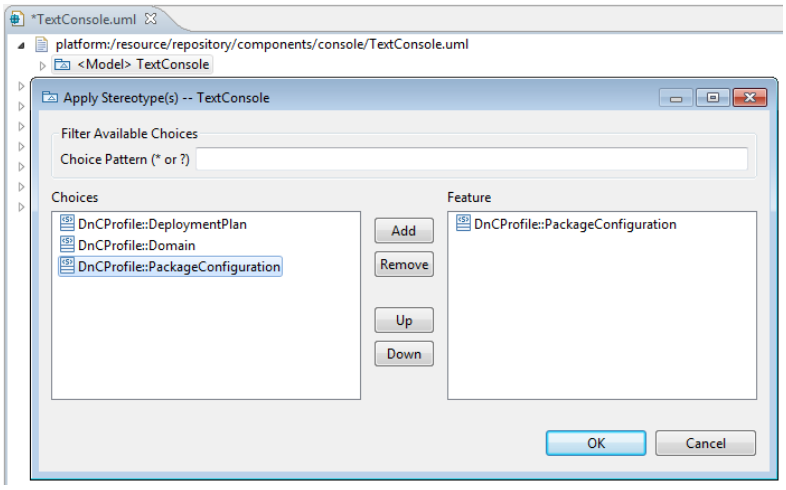

Figura 5.5: Aplicar estereotipo en UML2

Aplicado el estereotipo, cuyo nombre aparecerá entre << >> delante del de nuestro elemento, solo quedan por definir las propiedades que tiene ese estereotipo. Para ello, en la pestaña Properties aparecerá el nombre del estereotipo que hemos aplicado a ese elemento y debajo las propiedades definidas para él, que hay que completar. Para ello solo hay que pinchar sobre la propiedad y completar con lo que se desee.

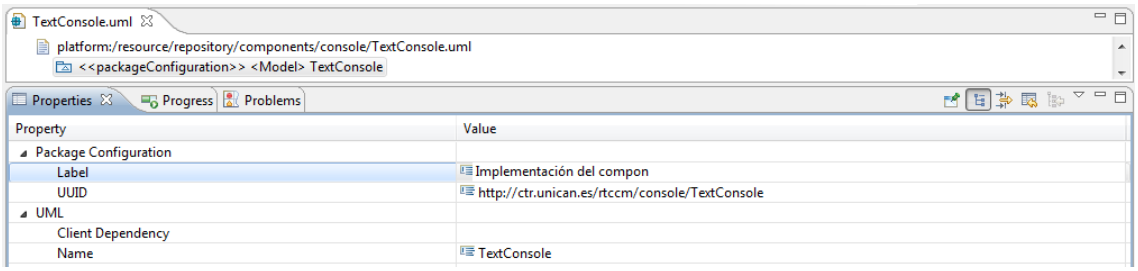

Figura 5.6: Definir propiedades en UML2

Asignar un estereotipo a un elemento en Papyrus es igual de sencillo. De nuevo en la pestaña Profile, dentro de Properties, pulsamos en la cruz verde de Applied Stereotypes. Como en el caso del UML2, aparece una nueva ventana con los estereotipos asociados a la metaclase a la que pertenece el elemento a asignar dicho estereotipo. Por último, para definir las propiedades asociadas a cada estereotipo, dentro de Applied Stereotypes, en la pestaña Profile en la que ya estábamos, desplegamos la línea del estereotipo apareciendo las propiedades a definir. Para definirlas, pinchamos en cualquiera de ellas y a continuación en la cruz verde de Property values.

5.3. Ejemplo de aplicación del perfil

En este apartado se utiliza un ejemplo para mostrar la aplicación del perfil propuesto a lo largo de todo el proceso de desarrollo de una aplicación. La aplicación ScadaDemo tiene por objetivo supervisar un conjunto de magnitudes físicas analógicas, adquiridas a través de tarjetas de adquisición de datos. Supervisar una magnitud supone leer su valor a una determinada frecuencia configurable y mantener una serie de registros estadísticos sobre los valores leídos. A una frecuencia más baja y también configurable se almacenan de forma persistente las estimaciones estadísticas evaluadas en ese periodo, junto con el instante de tiempo al que corresponden (la del momento de ser almacenadas). Asimismo, el último conjunto de datos almacenados para una determinada magnitud se muestra por pantalla de forma periódica, con periodo también configurable. La aplicación se controla a través del teclado, ofreciendo las siguientes posibilidades al operador:

- Elegir la magnitud de la que se muestran los datos almacenados.
- Ordenar o cancelar la supervisión de nuevas magnitudes

Por tanto, la funcionalidad de la aplicación responde a un conjunto de eventos concurrentes que se ejecutan de forma periódica o esporádica (en caso del evento de teclado):

- Recogida de los datos externos (evento temporizado). En cada ciclo de recogida de datos, se leen los valores de todas las magnitudes supervisadas de forma consecutiva. Esto es, la frecuencia de muestreo es única para todas las magnitudes.
- Almacenamiento de esos datos en un dispositivo de almacenamiento (evento temporizado con menor frecuencia que el anterior, de ser mayor no tendría sentido

puesto que se repetirían datos). En cada ciclo de almacenamiento se envían los datos de todas las magnitudes que están siendo supervisadas.
- Muestra por pantalla de los datos enviados al dispositivo de almacenamiento para una de las magnitudes muestreadas (evento temporizado).
- Modificación de las magnitudes mostradas por pantalla en respuesta a una pulsación del teclado (evento externo).

Resaltar de nuevo que los eventos temporizados son generados por el reloj del sistema y programables en frecuencia por el usuario.

Muchas de las aplicaciones desarrolladas mediante la metodología basada en componentes implementan arquitecturas cliente/servidor de 3 capas [18], en las que la estructura de una aplicación se divide en tres niveles:
- Una capa de presentación, que representa el nivel más alto de la aplicación. Está formada por los componentes más ligeros que, generalmente, implementan la interfaz de interacción entre usuario y aplicación.
- Una capa de negocio o de aplicación donde se implementa la funcionalidad más específica de la aplicación dentro de un dominio en concreto. Estos componentes suelen ser más complejos, pues son los que soportan la mayor parte de la lógica de la aplicación.
- Una capa de datos que engloba los componentes que dan acceso a los recursos del sistema o, como en nuestro caso, a los datos que son utilizados por los componentes de la capa de negocio.

Aplicando una arquitectura de tres capas y siguiendo una metodología de desarrollo de aplicaciones basada en componentes, la arquitectura genérica de la aplicación ScadaDemo se diseña como un ensamblado de cinco tipos de componente, cuatro reutilizables y uno que es implementado para esta aplicación en concreto.

Los cuatro componentes reutilizables son:
- ScadaEngine: Es el componente central de la aplicación, que por lo tanto soporta la mayor parte de lógica de negocio. Se trata de un componente que implementa una funcionalidad SCADA (*Supervisory Control and Data Acquisition*) de modo genérico y reutilizable, independiente del sistema SCADA concreto en el que vaya a ser integrado. Para ello, se independiza su comportamiento de los dispositivos a través de los que se capturan los datos, para lo cual se define el puerto requerido *adqPort* (interfaz *AnalogIO*), y también, del método de almacenamiento de los valores, que se realiza a través del puerto *logPort* (interfaz *Logging*). A través de su único puerto ofertado, *controlPort* (interfaz *ScadaControl*) se ofrece la funcionalidad SCADA, con métodos para ordenar la supervisión de nuevas magnitudes, cancelarlas, obtener los últimos datos enviados al logger, etc.
- IOCard: Es un componente perteneciente al dominio *io*, y está destinado a la gestión de una tarjeta de adquisición de señales analógicas o digitales, definido de tal forma que pueda proporcionar acceso a cualquier tarjeta de interfaz estandarizada. A través del puerto *analogPort* (interfaz *AnalogIO*) ofrece capacidad para leer y escribir líneas analógicas, mientras que a través del puerto *digitalPort* (interfaz *DigitalIO*) gestiona líneas digitales.
- Logger: Pertenece al dominio *db* y es un tipo de componente que se utiliza para almacenar información de texto, a la que se le asocia la marca del instante en el que

la información es registrada. Toda esta funcionalidad se ofrece a través de su único puerto ofertado, *regPort* (interfaz *Logging*).

- TextConsole: Este componente pertenece al dominio *console*. A través de su único puerto, *rwPort* (interfaz *TextRW*), ofrece la posibilidad de leer y escribir texto a través del teclado y la pantalla, respectivamente.

El componente específico de esta aplicación es:

- ScadaManager: Es el componente que implementa la capa de presentación, o cliente, de la aplicación. Es en él en el que se implementa la interfaz de usuario, desde la que se manejan las tareas específicas de supervisión.

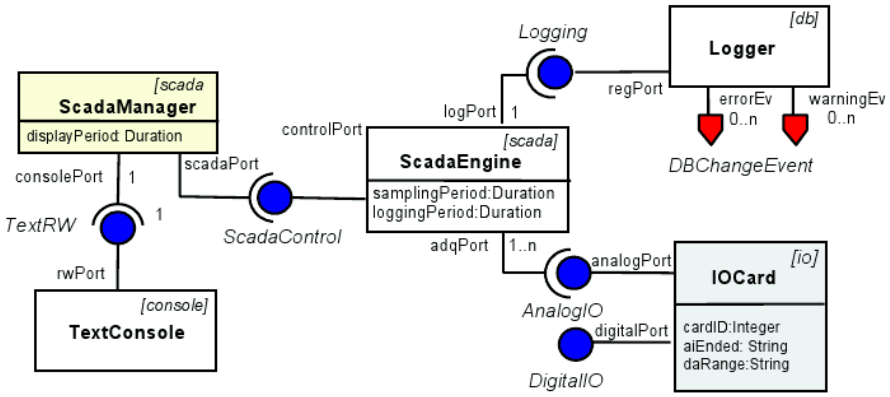

Figura 5.7: Arquitectura genérica de una aplicación Scada

Para ScadaDemo, la aplicación de prueba que vamos a poner como ejemplo, y puesto que la conexión entre IOCard y ScadaEngine está definida como una relación 1..n en la anterior arquitectura, se van a asociar dos tarjetas de adquisición de datos diferentes de forma simultánea, esto es, 2 instancias de componente IOCard.
Además, ya en esta estructura de instancias podemos ver como estarán configurados todos los valores de las propiedades de cada uno de los componentes.

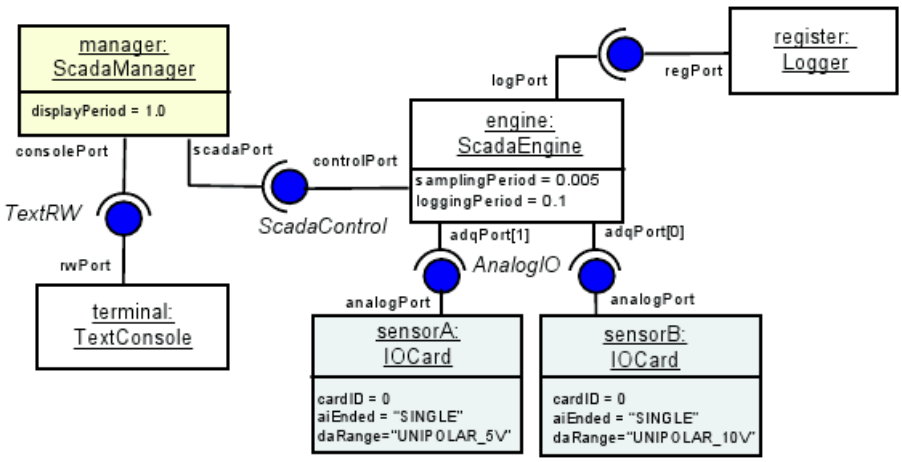

Figura 5.8: Estructura de instancias de la aplicación ScadaDemo

5.4. Desarrollo de componentes reutilizables

Para explicar el desarrollo de los componentes reutilizables se utiliza como ejemplo el desarrollo del componente ScadaManager.

Se abre un nuevo modelo Papyrus, dentro de la carpeta *components*. La definición de un componente dentro de un modelo es siempre similar, ha de tener una interfaz de componente, una o más implementaciones y cada una de ellas un Artifact asociado.

Para el modelado de la interfaz de un componente se utiliza un diagrama de estructura compuesta o Composite Structure, que se identifica con el mismo nombre que la propia interfaz, ScadaManager, en este caso.

La interfaz de componente se implementa a través de un elemento UML Component, de nombre ScadaManager, estereotipado como <<ComponentInterface>>. A continuación se definen los puertos del componente, que en este caso son dos:

- *scadaPort*: Se trata de un puerto requerido, que requiere la interfaz ScadaControl. Es el puerto a través del que se van a ordenar las tareas de supervisión y se van a requerir datos acerca de las mismas.
- *consolePort*: Se trata de un puerto requerido, que requiere la interfaz TextRW. Es el puerto a través del que se enviarán datos a pantalla y se leerán datos de teclado.

Ambos puertos se definen a través de elementos UML de tipo Port, que se añaden como puertos del propio componente (atributo port), y se estereotipan como <<ComponentPort>>. Para asignar la naturaleza de puerto requerido es necesario definir una clase auxiliar que se asigna como tipo al puerto, en la cual se define la correspondiente interfaz como requerida. Esto es necesario puesto que Papyrus solo permite que los puertos sean ofertados u ofertados y requeridos, pero no exclusivamente requeridos.

Aquellos elementos que se necesiten en el modelo pero que sean externos a él, como es el caso de las interfaces requeridas a través de los puertos, necesitan ser importados para hacer uso de ellos. El proceso de importación se puede realizar dos formas:

- A través de “Element Import”, que crea el enlace necesario para importar un único elemento de un modelo, pero que no está definido del todo en Papyrus y hay que completarlo a través de la interfaz de texto, añadiendo la ID del elemento y su href correspondiente.
- Importando todo el modelo de donde queremos obtener el elemento en cuestión, a través de “Load Resource”, en la pestaña UML Editor de la barra de tareas, y escogiendo después los elementos necesarios.

A continuación se definen las propiedades de configuración de la interfaz de componente. Cada propiedad se añade a través de un elemento de tipo UML Property, que se estereotipa como <<ConfigProperty>>. En este caso, se define la propiedad que corresponde a la frecuencia con la que se mostrarán los datos por pantalla: *displayPeriod*, de tipo Float.

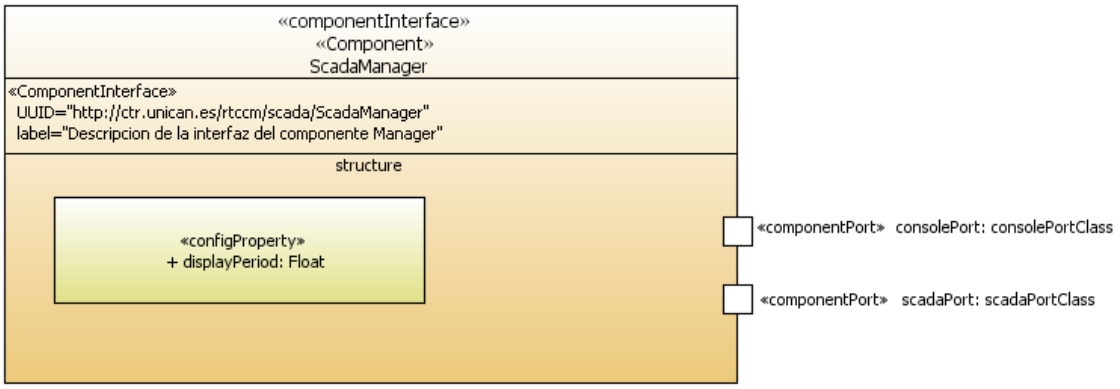

Figura 5.9: Descripción de la interfaz de componente ScadaManager

Para esta fase, el Specifier puede contar con la ayuda de la herramienta *InterfaceCatalog*, que muestra, organizada por dominios, información acerca de todas las interfaces disponibles en el *workspace*, gracias a su etiqueta (*label*) y la lista de las operaciones que contiene. La herramienta ofrece también la posibilidad de realizar el import automático de una interfaz o de un dominio completo al modelo que se esté desarrollando.

El siguiente paso en el desarrollo de un componente consiste en la elaboración y descripción de una implementación concreta de la interfaz anteriormente diseñada.
En este caso se trata de una implementación del componente para la tecnología Ada-CCM [9], denominada AdaScadaManager, escrita en lenguaje Ada 2005. La metodología que se propone en este trabajo se centra en la descripción de los componentes de cara a su posterior utilización en aplicaciones futuras, por ello no se dan detalles del proceso de elaboración del propio código del componente, el cual es más dependiente de la tecnología de componentes utilizada. Sí es importante resaltar que el *Developer*, podría disponer también de una herramienta de ayuda para la generación de código (o al menos del esqueleto de éste, acorde a la tecnología que esté utilizando) basado únicamente en los metadatos descritos hasta el momento. Este es el caso de la tecnología Ada-CCM, donde dicha herramienta genera el esqueleto del código de negocio del componente, únicamente a partir de la descripción de la interfaz de componente (por el momento dicha herramienta se basa en los descriptores XML de la interfaz, en el futuro debería ser lanzada desde el propio modelo UML).

En el ejemplo utilizado en este caso, el código, una vez elaborado, se distribuye como un único fichero, correspondiente a una librería Ada y denominado AdaScadaManager.o.

Lo importante es que se deben modelar, haciendo uso del perfil, los metadatos que sirven para hacer uso de esta librería de forma opaca. Para ello, se abre un nuevo diagrama de clases (se decide ese tipo de diagrama porque es imposible meter instancias en un diagrama de estructura compuesta), y a través de él se define un nuevo Component UML, que se estereotipa como <<MonolithicImplementation>>, puesto que se trata de una implementación monolítica y no desarrollada como ensamblado de componentes.

Es en esta clase donde se definen los requisitos que necesita esta implementación para su posterior despliegue. Estos requisitos se modelan como instancias de la clase

Requirement, definida en el propio perfil DnCProfile, dentro del propio modelo y se enlazan con el estereotipo <<MonolithicImplementation>> a través de su atributo *deployRequirement*. Cada instancia de Requirement tiene tres slots o atributos: *resourceType*, *name* y *value*, que indican el tipo de recurso sobre el que se realiza el requerimiento, su nombre y el valor que toma (como por ejemplo: "SO, name, MARTE" o "SO, version, 1.2.0"). En este caso se definen tres requisitos, como se observa en la figura 5.10.

La implementación definida implementa la correspondiente interfaz ScadaManager, por lo que se relacionan a través de una asociación UML de tipo Realization. Esta relación lo que implica, por definición, es que tanto los puertos como las propiedades definidos en la interfaz de componente (especificación) también aparezcan en la implementación.

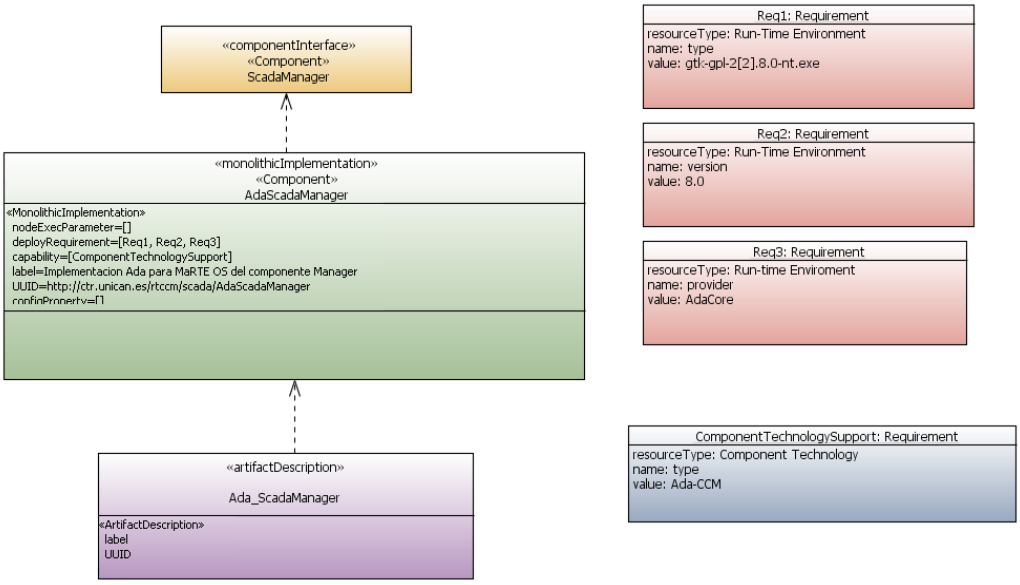

Figura 5.10: Descripción de la implementación AdaScadaManager

Por último, el diseño del componente concluye con la definición del artifact asociado a cada una de las implementaciones realizadas. Este artifact tiene asignado el estereotipo <<ArtifactDescription>>, y se asocia con su implementación mediante una relación de manifestación o Manifest, relación exclusiva para elementos de este tipo. En estos artifacts se define la localización de los archivos de código (AdaScadaManager.o en este caso). Estos archivos han de tener una dirección relativa al repositorio, para que las herramientas que lanzan la aplicación puedan encontrarlos sin problema alguno.

La última fase es la fase de empaquetado. Toda la información acerca del componente debe ser empaquetada de manera que se pueda hacer disponible para su futura utilización. Para ello se asigna al paquete raíz del modelo el estereotipo <<PackageConfiguration>>, cuyo *label* y *UUID* se rellenarán de la forma más precisa posible, ya que será lo que contenga la información más genérica posible sobre el componente que proporciona el paquete.

Una vez empaquetado el componente es necesario poder exportarlo del entorno de trabajo para depositarlo en los repositorios pertinentes. Para ello, debemos pinchar con el botón derecho sobre el directorio donde se encuentran los archivos concernientes al componente que queremos exportar (tanto el modelo UML como los ficheros

correspondientes a las implementaciones, en este caso sólo uno) y seleccionar la opción Export. Aparecerá una ventana donde elegiremos la forma en la que lo queremos exportar y finalizamos.

Si queremos realizar la acción contraria e importar un componente a nuestro *workspace* para trabajar con él, el método de importación es semejante al de importación del perfil explicado en el apartado 5.2.1.

5.5. Desarrollo de aplicaciones basadas en componentes

5.5.1. Desarrollo de la aplicación ScadaDemo

Una vez diseñados todos los modelos de componentes, podemos pasar al diseño de la aplicación mediante el ensamblaje de componentes. De nuevo, creamos un modelo Papyrus vacío y un nuevo diagrama de estructura compuesta. Al aplicar el concepto de recursividad, como ya se explicó en la introducción, una aplicación no es más que un componente compuesto, luego su definición se realiza siguiendo los mismos pasos que en el caso que acabamos de ver. Se crea una <<ComponentInterface>>, denominada ScadaDemo, que en este caso no define puertos, aunque sí propiedades (aquellas que son globales de la aplicación):

- *samplingPeriod*: para el periodo de muestreo.
- *loggingPeriod*: para el periodo de escritura de datos.
- *displayPeriod*: para el periodo en el que los datos se muestran por pantalla.

Figura 5.11: Descripción de la interfaz ScadaDemo

A continuación, creamos otro componente, ScadaDemoAssembly, que realiza el anterior, pero que en este caso se estereotipa como <<AssemblyImplementation>>. La estructura interna del componente se muestra en la figura 5.12.

Por cada subcomponente, estereotipado como <<SubComponent>>, que lo forma se define una propiedad (asignada como atributo part de la clase Component UML) cuyo

tipo corresponde a la interfaz de componente elegida. En este caso tenemos 5 propiedades: *engine*, de tipo ScadaEngine, *manager*, de tipo ScadaManager, *sensorA* y *sensorB* de tipo IOCard y finalmente, *register*, de tipo Logger. Todas ellas se asocian asimismo al atributo *containedComponent* del estereotipo <<AssemblyImplementation>>.

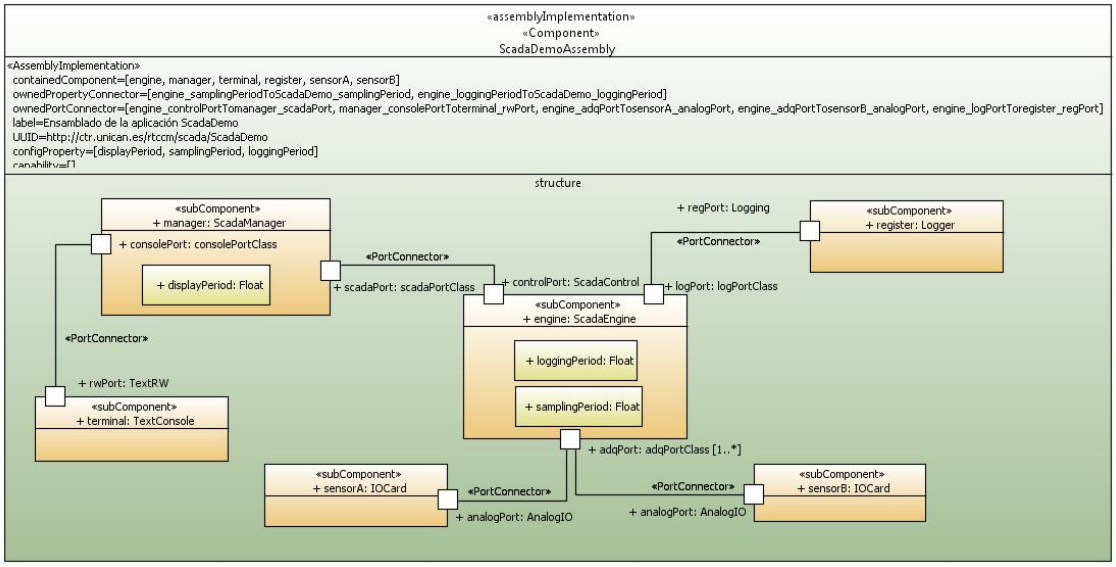

Figura 5.12: Descripción del ensamblado ScadaDemoAssembly

Para completar la arquitectura del ensamblado es necesario definir la conexión entre los subcomponentes, la cual se realiza a través de sus puertos. Para ello, debemos mostrar cada uno de los puertos de las instancias de subcomponentes en el diagrama. A través del botón derecho elegimos la opción Filters -> Show/Hide Contents y en el menú contextual que aparece, se seleccionan todos los puertos a mostrar. Una vez mostrados, tan solo queda unirlos mediante Connectors UML, siguiendo la arquitectura definida anteriormente. Cada uno de los conectores definidos se estereotipa como <<PortConnector>>.

Finalmente es necesario realizar el mapeado de las propiedades globales del componente a las propiedades de los subcomponentes. Creamos un nuevo conector en el árbol UML2 (ya que Papyrus no te ofrece la opción de crear conectores gráficamente para la unión de propiedades) y dos ConnectorEnd para unir los extremos de éste con las propiedades a mapear. Los connectorEnd han de definirse de la siguiente forma: el campo *role* se completa con la propiedad que queremos conectar, mientras que el campo *partWithPort* se completa con el elemento que contiene dicha propiedad. Para ello debemos ayudarnos de las opciones avanzadas que nos ofrece Papyrus (o, en su defecto, de las opciones del árbol UML2). Todos estos conectores tienen que definirse con el estereotipo <<PropertyConnector>>.

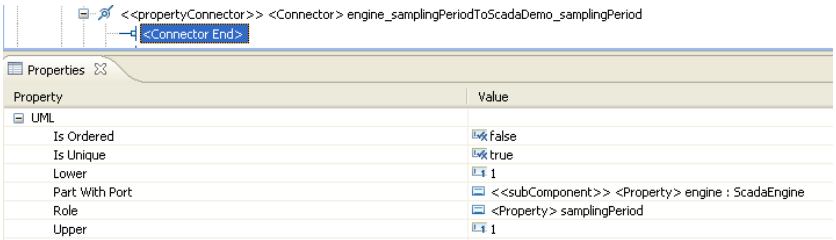

Figura 5.13: Definición de un <<PropertyConnector>> y sus ConnectorEnd

Para esta fase, el agente *Assembler* puede contar con la ayuda de una herramienta denominada *ComponentCatalog* que, de forma parecida a la herramienta *InterfaceCatalog*, daría información de los componentes disponibles en el *workspace*, con sus puertos y propiedades y con la posibilidad de importación automática.

5.5.2. Diseño de la plataforma

Una vez que tenemos la aplicación diseñada, para poder desplegarla necesitamos conocer la plataforma, y en base a ella, elegir implementaciones de componentes compatibles. Para crear un modelo de plataforma, dentro de nuestro *workspace* creamos un nuevo modelo Papyrus en la carpeta *platforms*. Le agregamos un diagrama de despliegue, que tendrá los elementos necesarios para modelar la plataforma del sistema, aunque por el momento Papyrus sólo da soporte a la edición gráfica de nodos. Lo que falta por hacer del modelo lo realizaremos a través del diagrama en árbol de UML2.

Tras definir los nodos como <<Node>>, continuaremos el diseño comunicando ambos mediante un elemento UML de tipo CommunicationPath, creándolo a partir del paquete raíz como un elemento más del diseño. Para que ambos nodos se comuniquen entre sí, tenemos que crear dos propiedades del tipo de los nodos que queremos comunicar, algo así como los ConnectorEnd de un *Assembly* y asociarlas a la propiedad Member End del CommunicationPath. Lo estereotipamos como <<Interconnect>>. Comunicados ambos nodos, hay que añadir los recursos que, en un futuro, tendrán que ser contrastados con los requisitos de los componentes. Cada recurso correspondiente al nodo se crea como un elemento de tipo Class a la que asignaremos el estereotipo <<Resource>> y que se asociarán con el estereotipo <<Node>> mediante el atributo *ownedResource*.

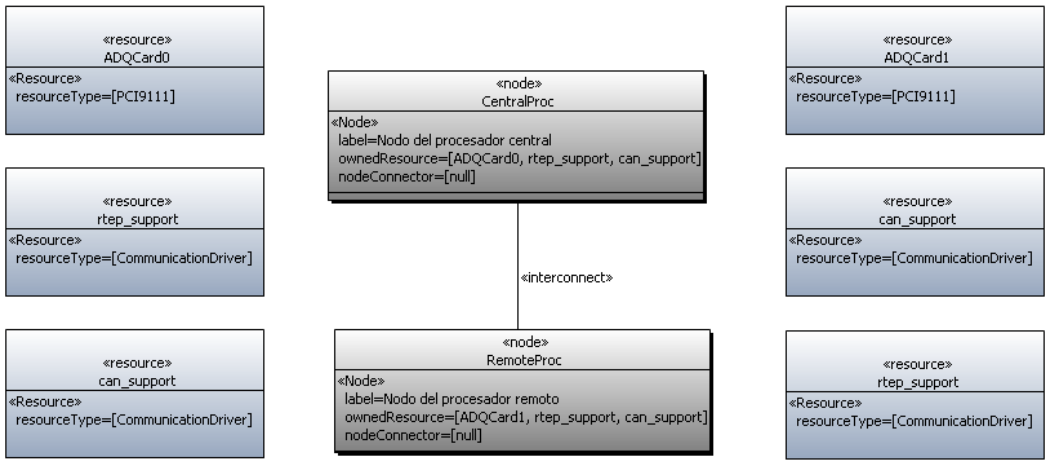

Figura 5.14: Plataforma de ejecución de ScadaDemo

5.5.3. Diseño de la aplicación desplegada

Finalizamos el diseño de aplicaciones mediante componentes creando el despliegue de nuestra aplicación ScadaDemo concreta. Para la aplicación también necesitaremos un

diagrama de despliegue en un nuevo modelo Papyrus, aunque de nuevo solo lo utilizaremos para mostrar los nodos gráficamente.

Lo primero que debemos hacer es importar el <<Domain>>, ya que necesitaremos el modelo de plataforma que ha sido creado en él. Asimismo, debemos importar el modelo en el que se define la aplicación, en este caso, el paquete ScadaDemo.

La aplicación final, estereotipada como <<Application>>, será una instancia de la <<AssemblyImplementation>> definida en el paquete anterior, esto es, de ScadaDemoAssembly. Como muestra la figura 5.15, al ser una instancia, tendrá slots a través de los que se les de valor a cada una de sus propiedades. Hay dos tipos de propiedades:

- Propiedades de configuración: Slots estereotipados como <<ConfigValue>> y correspondientes a las propiedades globales de configuración del ensamblado. Generalmente reciben valores escalares, como es el caso de *displayPeriod*, *samplingPeriod* y *loggingPeriod*, en este ejemplo. Gracias al mapeado de propiedades que se realizó en la definición del ensamblado, basta con dar valor a las propiedades globales en el elemento estereotipado como <<Application>>. Las herramientas que se encarguen del despliegue procesarán la información de ambas fuentes (el plan de despliegue y la definición del ensamblado) para configurar cada instancia concreta con los valores adecuados.
- Propiedades correspondientes a las instancias de componente que forman el ensamblado (asociación *containedComponent* en el <<AssemblyImplementation>>). Estas propiedades deben recibir como valor las referencias a las instancias de implementaciones concretas que se utilizan en esta aplicación para cada subcomponente identificado en el ensamblado. Por tanto, por cada componente del ensamblado se añade al modelo de despliegue una instancia de una implementación concreta de la correspondiente interfaz de componente, las cuales se estereotipan como <<DeployedInstance>>. En este caso, la implementación está orientada a la tecnología Ada-CCM, luego se eligen implementaciones compatibles con dicha tecnología. Así, a la propiedad *manager*, se le asigna la instancia ScadaDemoManager, que como se observa en la figura 5.15, corresponde a la implementación AdaScadaManager. A la propiedad *engine* se le asigna la instancia ScadaDemoEngine, de tipo AdaScadaEngine, etc.

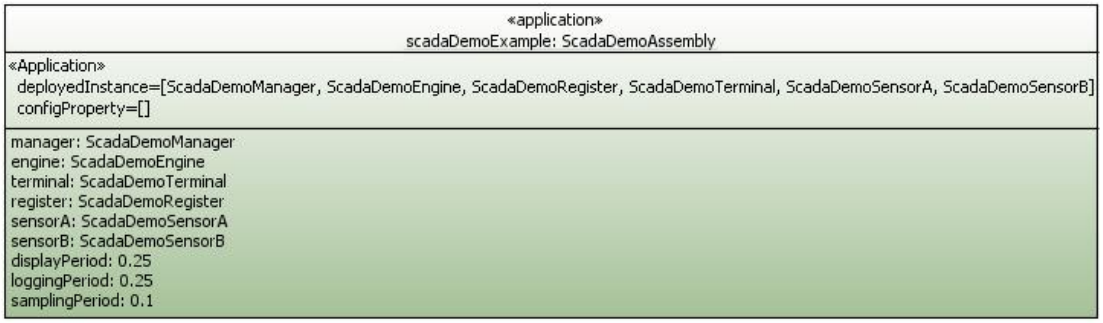

Figura 5.15: Instanciación de la aplicación ScadaDemoExample

Por último, las instancias deben ser asignadas al nodo en el que van a ser ejecutadas por lo que se unen a él a través de una relación de despliegue o Deployment. Para ello, se creará una instancia por cada nodo creado en el modelo de plataforma. En nuestro caso, crearemos dos instancias de nodo: una para CentralNode y otra para RemoteNode,

donde aplicaremos las relaciones de despliegue correspondientes, tal y como se muestra en la figura 5.16.

Todas las relaciones de despliegue se crearán en UML2 como parte del nodo (atributo client = nodo correspondiente, que aparece de forma automática). De esta forma, sólo hará falta definir el atributo supplier, por lo que seleccionaremos una de las instancias anteriormente citadas por cada relación de despliegue.

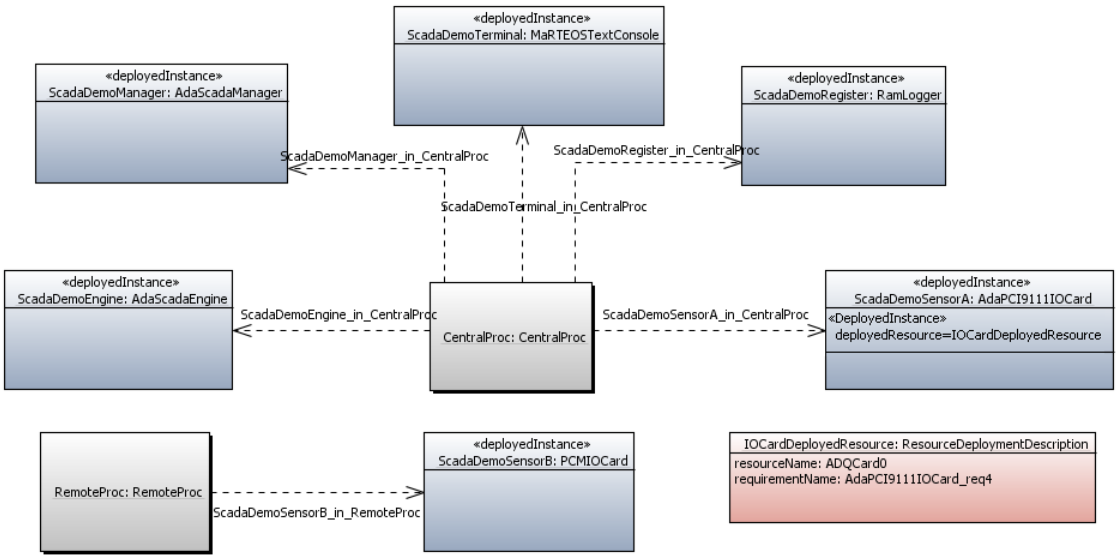

Figura 5.16: Plan de despliegue de ScadaDemoExample

Para esta fase, el *Planner* puede contar con la ayuda de una herramienta, de nuevo similar a las anteriores, que le muestre las características de las diferentes implementaciones disponibles en el repositorio, para elegir las más adecuadas a la plataforma disponible. De esta forma, debería mostrarse información acerca de las *Capabilities* de cada implementación, así como de los requisitos que imponen en la plataforma para ser ejecutadas. En base a ello, el *Planner* define los elementos de tipo *ResourceDeploymentDescription* que le sirven a las herramientas para detectar si una implementación es compatible con un determinado nodo.

Por último, el agente *Executor* lidera la fase de lanzamiento de la aplicación, haciendo uso de una serie de herramientas que, en base a la información del plan de despliegue, puedan comenzar la ejecución de la aplicación. Estas herramientas son muy dependientes de la tecnología de componentes concreta, pero en general deben llevar a cabo labores como la generación de "*glue-code*" necesario para conectar los componentes, generación del código de los contenedores de componentes en caso de utilizarse modelos de programación de estilo contenedor/componente, generación del ejecutable (o ejecutables) finales de la aplicación, etc.

6. Extensión de tiempo real: RT-DnCProfile

Una vez definido el perfil para la especificación estándar, se va a definir una extensión del mismo orientada a la descripción del despliegue y configuración de sistemas de tiempo real. Este perfil da soporte a una extensión de la propia especificación D&C, denominada RT-D&C [12], que se ha desarrollado previamente en el grupo de Computadores y Tiempo Real de la Universidad de Cantabria.

Los sistemas de tiempo real son sistemas informáticos que, por su funcionalidad o naturaleza, interaccionan con el medio externo, debiendo generar respuestas a los eventos que surjan en él. Este entorno evoluciona en el tiempo y, por ello, el correcto funcionamiento de este tipo de sistemas no sólo depende de la generación correcta de las respuestas requeridas, y los nuevos eventos que éstas produzcan, sino de que éstas se realicen en los plazos de tiempo establecidos. Frecuentemente estos sistemas forman parte de otros de mayor envergadura, es decir, son sistemas empotrados, que se encargan de la supervisión de alguna función interna del sistema completo: electrodomésticos, coches, vehículos espaciales, etc.

Los sistemas de tiempo real se pueden clasificar en sistemas de tiempo real estricto (*hard real-time*), en los que un plazo perdido supone un error fatal e irrecuperable, y sistemas de tiempo real laxo (*soft real-time*), en los que no supondría un error fatal y lo que se exige es un porcentaje mínimo de plazos cumplidos.

La principal característica de las aplicaciones de tiempo real es su predictibilidad, es decir, que el sistema ofrezca un comportamiento temporal conocido que permita certificar el cumplimiento de los plazos temporales asociados a sus respuestas [16]. Puesto que el término sistema de tiempo real se utiliza, generalmente, para referirse tanto a la aplicación software como a la plataforma, estos serán los puntos básicos en la extensión del perfil.

Respecto al modelado de aplicaciones de tiempo real, el OMG presenta dos perfiles para la estandarización de los modelos y herramientas de diseño de sistemas de tiempo real: el "*UML Profile for Schedulability, Performance and Time*" (SPT) [17], y el más actual, que sustituye y perfecciona el anterior, "*UML Profile for Modeling and Analysis of Real-Time and Embedded Systems*" (MARTE) [14]. MARTE está orientado especialmente a la descripción del comportamiento temporal interno de los sistemas y, aunque ofrece un capítulo relacionado con el modelado de componentes, no está orientado al diseño de aplicaciones basadas en éstos.

La implantación de tiempo real en las tecnologías basadas en componentes está siendo muy lenta, aunque ciertas industrias hayan demostrado su interés en adoptarlas. El principal obstáculo que presenta es el hacer compatibles las dos características de cada una de ellas:

- La capacidad de ensamblar componentes reutilizables, manejarlos de forma opaca y de integrarlos en diferentes aplicaciones sin necesidad de modificarlos.
- La capacidad de garantizar un comportamiento temporal predecible en el que se puedan certificar la ejecución de sus actividades satisfaciendo los requisitos temporales establecidos.

Por todo esto, en RT-D&C se considera que un componente de tiempo real es aquel que incluye en el paquete en el que es distribuido toda la información y metadatos que se requieran para predecir y analizar el comportamiento temporal en las aplicaciones en el que vaya a ser utilizado.

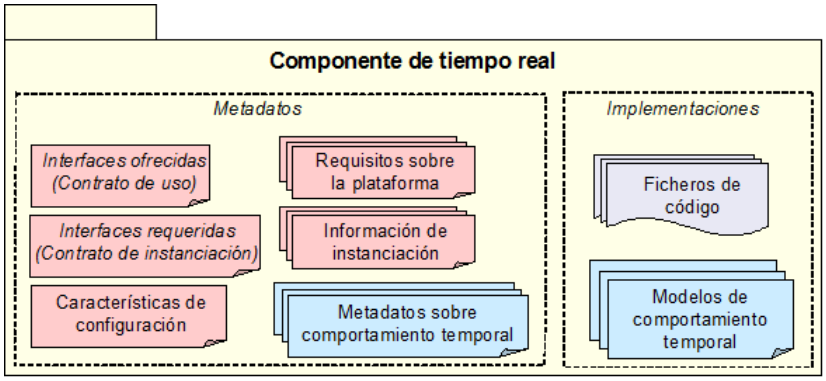

Figura 6.1: Paquete de distribución de un componente de tiempo real

Para poder llevar a cabo la obtención del modelo de una aplicación a partir de los modelos de componentes es necesario contar con alguna metodología de modelado que ofrezca características de componibilidad y reutilización. El modelo de un componente debe ser elaborado de forma reutilizable, independiente de las aplicaciones en las que vaya a ser utilizado. Para ello, será un modelo parametrizado, que dejará como parámetros sin resolver aquellas características que dependan del modelo en el que el componente es utilizado en una aplicación. Cuando en una aplicación concreta se conozcan los modelos de todos los componentes que la forman, cada uno de ellos recibirá valores adecuados para todos sus parámetros, y todos ellos podrán ser compuestos para generar el modelo global de la aplicación. Este modelo será después analizado con herramientas de análisis de tiempo real que certifiquen el cumplimiento de los requisitos temporales impuestos en la aplicación. Una metodología de este tipo ha sido desarrollada también en el grupo CTR [20].

En base a este criterio se han definido las extensiones propuestas en RT-D&C, y por tanto, los cambios o extensiones que se deben realizar en el perfil ya propuesto deben suponer que:

- Un componente incluye metadatos de tiempo real que permiten al diseñador de una aplicación evaluar si se puede elaborar un modelo de comportamiento temporal de la aplicación que sirva de base a las herramientas de diseño y análisis de tiempo real.
- Se incluyen metadatos que permitan al diseñador elegir los componentes más adecuados para una aplicación dependiendo de la reactividad (capacidad de respuesta a eventos) de éstos.

- Pueden ser definidos parámetros especiales que permitan controlar la planificación y concurrencia (propiedades de configuración de tiempo real) de las implementaciones de componentes.

6.1. RT-DnCProfile

Todos los elementos que se definen en este perfil son extensiones de aquellos con el mismo nombre del perfil DnCProfile, a excepción de los que propiamente se indiquen, que serán de nueva creación.

El perfil RTDnCProfile importa el perfil DnCProfile y se estructura en cuatro paquetes, tres de ellos (RTComponents, RTTarget y RTDeployment) con el mismo significado que en el caso anterior, más un paquete adicional, RTWorkload, cuyo significado se explica más adelante.

6.1.1. Paquete RTComponents

Como se observa en la figura 6.2, los nuevos estereotipos definidos como extensiones de los definidos en el perfil DnCProfile son <<RTComponentInterface>>, <<RTComponentPort>>, <<RTComponentImplementation>> y <<RTMonolothicImplementation>>. La función principal de estos estereotipos consiste en identificar a los elementos que estereotipan como elementos de tiempo real e incluir los metadatos que se derivan de dicha naturaleza.

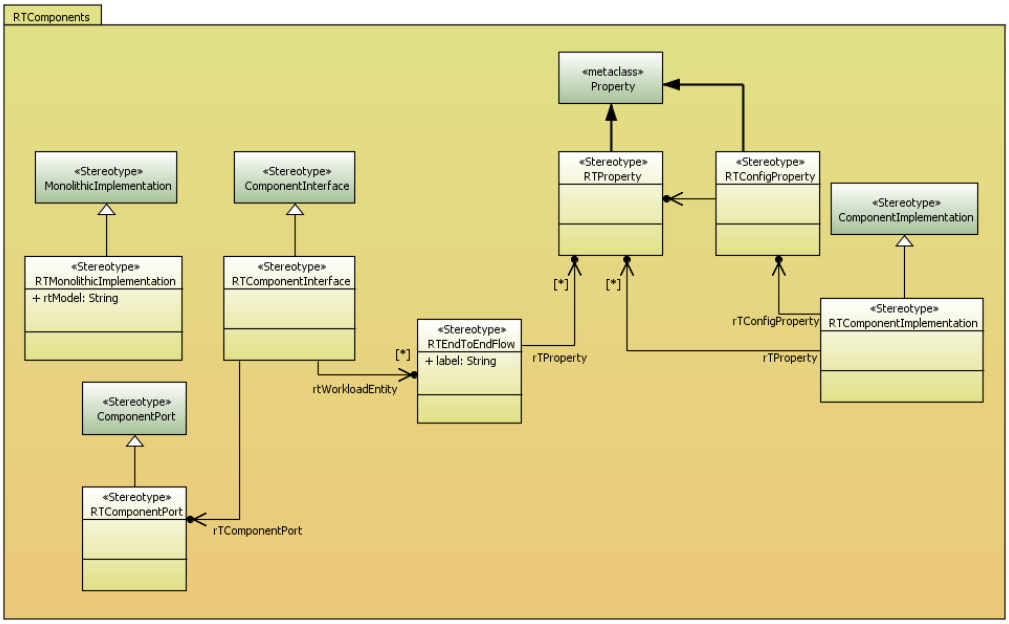

Figura 6.2: Paquete RTComponents

RTComponentInterface:
El estereotipo <<RTComponentInterface>> identifica una interfaz de componente como de tiempo real, esto es, el componente que la implemente va a ofrecer algún tipo de

modelo temporal de los servicios que ofrece y de las respuestas a eventos que puede generar en un sistema. Define las siguientes asociaciones:

- *rtWorkloadEntity* : RTEndToEndFlow [0..*] => Conjunto de tipos de respuestas a eventos que el componente puede generar en una aplicación.
- *rtComponentPort* : RTComponentPort [0..*] => Conjunto de puertos del componente que ofrecen modelos de comportamiento temporal. Se trata de un subconjunto de la asociación containedPort, heredada de la clase raíz.

RTComponentPort:
Este estereotipo <<RTComponentPort>> extiende el estereotipo <<ComponentPort>> del DnCProfile, y únicamente añade la semántica de que se trata de un puerto de tiempo real, esto es, un puerto para el que el componente va a ofrecer modelos de comportamiento temporal de todos los servicios ofertados a través de él.

RTEndToEndFlow:
<<RTEndToEndFlow>> es un estereotipo de nueva creación, que extiende a la metaclase UML Classifier. Identifica los tipos de respuesta a eventos que puede iniciar un determinado componente. Además del atributo *name* que la identifica y un atributo *label* que da información sobre su naturaleza, define la siguiente asociación:

- *rtProperty* : RTProperty [0..*] => Propiedades configurables del comportamiento temporal de la respuesta al evento. Podrán recibir diferentes valores para cada respuesta concreta de este tipo que se incluya en una aplicación.

RTProperty:
El estereotipo <<RTProperty>> también es de nueva creación y extiende a la metaclase UML Property. Señala propiedades de tiempo real correspondientes a los componentes o a sus respuestas. Una propiedad de tiempo real es aquella que influye en el comportamiento temporal del componente, y por tanto, de la aplicación en que éste sea utilizado. Corresponden a las propiedades configurables (o parámetros) de los modelos de tiempo real reutilizables en los que se basan las metodologías de modelado con características de componibilidad de las que se habló anteriormente.

RTComponentImplementation:
Extiende a <<ComponentImplementation>> e identifica una implementación de componente como de tiempo real, esto es, una implementación que además de su código, incluye como parte de su distribución, información acerca de su comportamiento temporal. Añade las siguientes asociaciones:

- *rtProperty* : RTProperty [0..*] => Propiedades configurables del comportamiento temporal del componente. Podrán recibir diferentes valores para cada instancia de componente de este tipo que se incluya en una aplicación.
- *rtConfigProperty* : RTConfigProperty [0..*] => Propiedades configurables de la implementación del componente relacionadas con el tiempo real, esto es, con aspectos como la concurrencia, la sincronización, etc.

Es importante distinguir entre estos dos conjuntos de propiedades. El primero se refiere a las propiedades configurables del modelo de tiempo real del componente, mientras que el segundo se trata de propiedades que deben ser asignadas al código del componente para configurar el comportamiento temporal del mismo durante la ejecución efectiva de la aplicación. Los valores de las propiedades del segundo conjunto

podrán extraerse de los obtenidos por las herramientas de análisis de tiempo real para algunas de las propiedades del primero.

RTConfigProperty:
Extiende a <<ConfigProperty>> e identifica aquellas propiedades de configuración de una implementación de componente cuyo valor influye en el comportamiento temporal del componente en tiempo de ejecución. Define el siguiente atributo:

- *mappedRTProperty*: RTProperty => Propiedad de tiempo real del componente de la que se toma valor. Las herramientas de análisis obtendrán valores óptimos para las propiedades de los modelos de tiempo real de los componentes, y estos valores serán mapeados a las propiedades reales que deben ser asignadas en tiempo de ejecución.

RTMonolithicImplementation:
Extiende a <<MonolithicImplementation>> y la añade únicamente un atributo, *rtModel*, que referencia la dirección física donde se encuentra el modelo de tiempo real para ese componente. Este modelo es un modelo parametrizado, cuyas propiedades son las definidas en el atributo *rtProperty* definido en el estereotipo <<RTComponentImplementation>>.

6.1.2. Paquete RTTarget

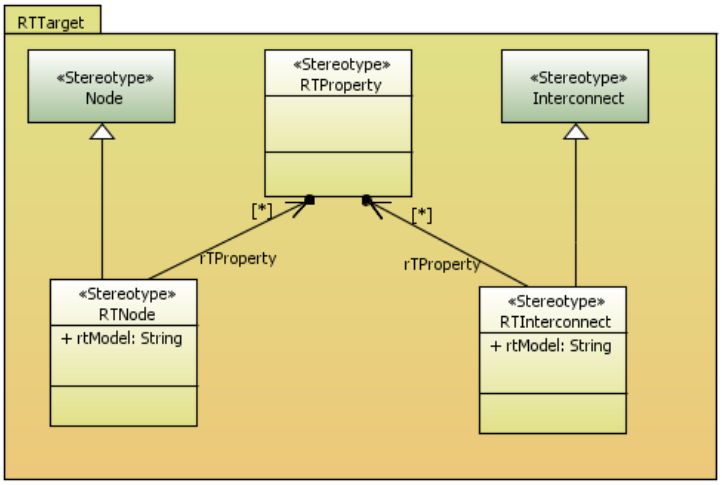

Figura 6.3: Estereotipos definidos en el paquete RTTarget

Un aspecto vital a tener en cuenta de cara al modelado de sistemas de tiempo real es que para la elaboración del modelo se necesita tanto el modelo de los componentes como el modelo de la plataforma en los que éstos van a ser ejecutados. Por ello, es necesario contar también con los modelos de comportamiento temporal de los elementos de la plataforma.

Para el modelo de plataformas se extienden dos estereotipos del perfil anterior: <<RTNode>> y <<RTInterconnect>>, que extienden respectivamente a <<Node>> e <<Interconnect>>. Ambos añaden el atributo *rtModel*, que al igual que en el caso anterior, referencia el modelo de tiempo real del nodo o red de comunicación. Además poseen ambos una asociación múltiple con elementos estereotipados como

<<RTProperty>>, también explicado anteriormente, que dota a ambos de propiedades configurables de tiempo real.

6.1.3. Paquete RTDeployment

Las aplicaciones de tiempo real basadas en componentes también necesitan un plan de despliegue que permita formalizar la aplicación y servir como base para su lanzamiento y ejecución. Este plan será básicamente igual que en el caso general, excepto que las implementaciones monolíticas del ensamblado se instanciarán como <<RTDeployedInstance>>, los valores de las <<RTProperty>> como <<RTValue>> y los valores de las <<RTConfigProperty>> como <<RTConfigValue>>.

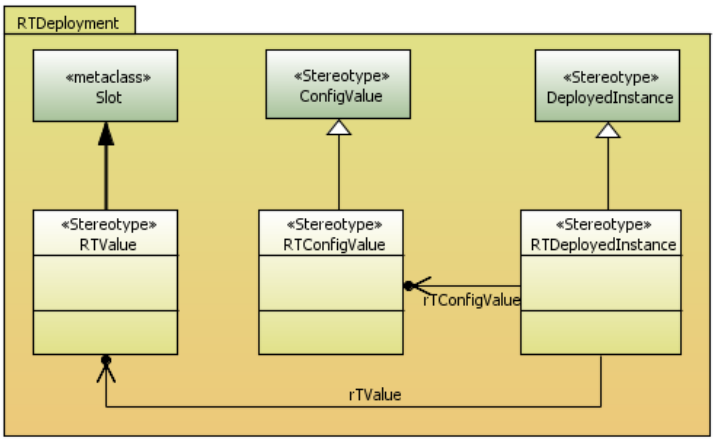

Figura 6.4: Paquete RTDeployment

RTDeployedInstance:
El estereotipo <<RTDeployedInstance>> extiende al estereotipo del DnCProfile <<DeployedInstance>> dotándolo de la misma semántica y señalando que es una instancia de un componente de tiempo real. Tiene dos asociaciones adicionales:

- *rtConfigValue* : RTConfigValue [0..*] => Valores que se le asignan a las propiedades de configuración del componente relacionadas con tiempo real.
- *rtValue* : RTValue [0..*] => Valores que se le asignan a las propiedades configurables del modelo de tiempo real del componente.

RTConfigValue:
El estereotipo <<RTConfigValue>> extiende al estereotipo <<ConfigValue>> e identifica aquellas propiedades de configuración asociadas a tiempo real. No tiene ni atributos ni asociaciones adicionales.

RTValue:
El estereotipo <<RTValue>> extiende a la metaclase Slot e identifica los valores que se le asignan a las propiedades de tiempo real del componente, esto es, de su comportamiento temporal o del modelo de tiempo real.

6.1.4. Paquete RTWorkload

Para poder generar el modelo de tiempo real de una aplicación de tiempo real se necesita contar con un modelo que describa las actividades que se ejecutan en la aplicación con respecto a las interacciones con el medio, esto es, las respuestas a eventos que se realizan en una aplicación, lo cual constituye su carga de trabajo (*workload*).

En la especificación D&C no existe ningún tipo de metadato que permita describir cargas de trabajo de aplicaciones, como es obvio al estar orientada a aplicaciones de carácter general,. Por ello, una de las principales extensiones que se han realizado en la extensión RT-D&C es definir un nuevo modelo de datos, complementario a los modelos ya existentes para componentes, plataformas y ejecución, que de soporte a la descripción de cargas de trabajo. Los estereotipos que se definen para dar soporte a este modelo son los que aparecen en la figura 6.5.

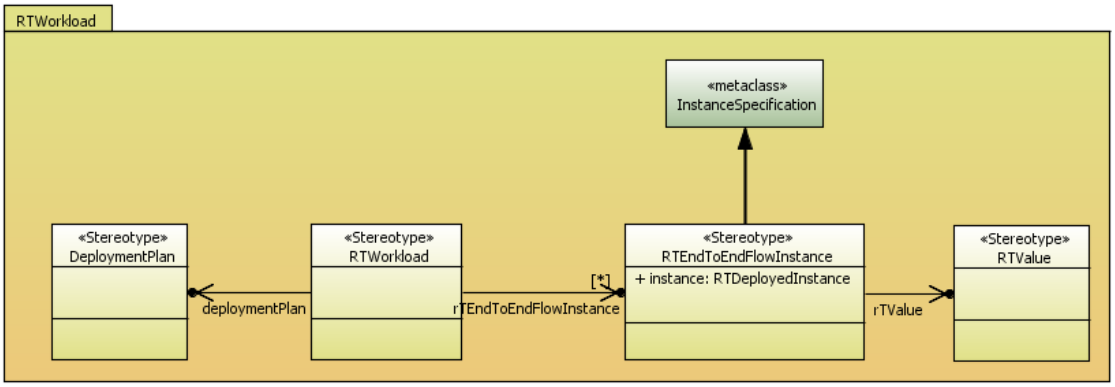

Figura 6.5: Estereotipos definidos en el paquete Workload

RTWorkload:
El estereotipo <<RTWorkload>> extiende a la metaclase Package. Identifica una carga de trabajo, esto es, el conjunto de actividades que ejecuta la aplicación de acuerdo a las interacciones que se esperan del entorno.
Tiene una única asociación:

- *rtEndToEndFlowInstance* : RTEndToEndFlowInstance [0..*] => Conjunto de todas las respuestas a eventos que se ejecutan en la aplicación.
- *deploymentPlan* : DeploymentPlan => Referencia al plan de despliegue al que se asocia esta carga de trabajo.

RTEndToEndFlowInstance:
<<RTFEndToEndFlowInstance>> extiende a la metaclase InstanceSpecification y se asignará a todas aquellas instancias de las clases <<RTEndToEndFlow>> definidas en el modelo de componentes. Cada una de ellas representa una respuesta a evento que se ejecuta en el sistema. Tiene una asociación múltiple:

- *rtValue* : RTValue [0..*] => Conjunto de valores que se asignan a las propiedades de tiempo real de cada respuesta concreta.
- *instance*: RTDeployedInstance => Referencia a la instancia de componente (de entre las definidas en el correspondiente plan de despliegue) que inicia esta respuesta en el sistema. Esta referencia es necesaria para que las herramientas de composición de modelos puedan acceder al modelo temporal concreto de la respuesta.

6.2. Ejemplo RT-ScadaDemo

De la misma forma que se aplicó el perfil DnCProfile en una aplicación ScadaDemoExample para comprobar su correcta definición, se aplicará esta extensión en la misma aplicación pero teniendo en cuenta su carácter de aplicación de tiempo real.

Las variaciones realizadas serán muy sencillas, pero cambian sustancialmente el proceso de desarrollo al tratarse de componentes y aplicación de tiempo real.

6.2.1. Desarrollo de componentes de tiempo real

Desde el punto de vista de la descripción de la interfaz de componente, además de definir el componente como <<RTComponentInterface>>, la principal diferencia radica en que se han de crear las clases que describen las respuestas a eventos asociadas a dicha interfaz, es decir, las respuestas a eventos que los componentes de dicho tipo pueden generar en una aplicación. La carga de trabajo de las futuras aplicaciones se definirá como el conjunto de instancias de dichas respuestas que se ejecutan efectivamente en cada aplicación. Para declararlas, se crea, dentro del propio modelo de componente, un nuevo diagrama de clases.

Cada respuesta se creará como una clase que se definirá como <<RTEndToEnfFlow>> y a la que se le pueden añadir propiedades configurables, estereotipadas como <<RTProperty>>. Una vez creadas, todas esas clases se añadirán al atributo *rtWorkloadEntity* de la interfaz de componente correspondiente.
En nuestro ejemplo, la interfaz de componente ScadaManager tiene dos respuestas asociadas. La respuesta DisplayTrans representa la respuesta al evento periódico que lanza la muestra de datos por pantalla, mientras que la respuesta ChangeTrans representa la respuesta al evento externo de teclado, que modifica la magnitud de la que se están mostrando los datos por pantalla o cancela tareas de supervisión. En ambos casos se define como propiedad de tiempo real el plazo de tiempo en el que la respuesta debe ser satisfecha (*deadline*), que podrá ser configurado de manera independiente en cada aplicación en la que se use el componente. Asimismo, en el caso de la respuesta ChangeTrans se define como parámetro una cota mínima para el tiempo que puede transcurrir entre dos lanzamientos consecutivos de la misma (necesario para realizar el análisis).

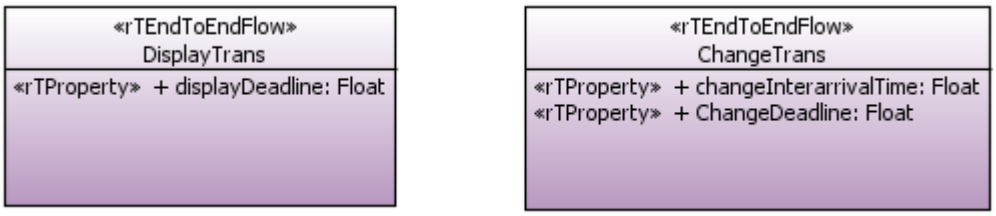

Figura 6.6: Respuestas a eventos de ScadaManager

La fase de elaboración de una implementación de componente cuando éste tiene naturaleza de tiempo real implica que junto al código del componente debe elaborarse su modelo de tiempo real, y la descripción de dicho modelo se debe añadir a la descripción de la implementación, de manera que pueda ser manejado de forma opaca.

Cabe reseñar que los modelos de tiempo real que se definan para los correspondientes componentes, de forma similar a lo que ocurre con los ficheros de código, han de ser especificados a través de la dirección física en la que se localizan, como se puede ver en la figura 6.7. De esta forma, la herramienta que genera el modelo de tiempo real de una aplicación conoce perfectamente dónde tiene que acceder para obtener lo que necesite.

El componente en este caso se estereotipa como <<RTMonolithicImplementation>> y a través de su atributo rtModel se referencia, por tanto, su modelo.

Este modelo puede ser parametrizado, por lo que otro de los trabajos adicionales que hay que realizar al implementar un componente de tiempo real es describir las propiedades del modelo de tiempo real y las propiedades de configuración de tiempo real en el caso de que exista alguna. Éstas se crearán en las implementaciones de cada componente y se definirá como <<RTProperty>> y <<RTConfigProperty>>, respectivamente.

En la implementación de ejemplo AdaScadaManager, en la figura 6.7, se definen las siguientes propiedades:

- Dos propiedades del modelo de tiempo real, *keyboardThPrty* y *displayThPrty*, que representan las prioridades a las que se realizan las respuestas *DisplayTrans* y *ChangeTrans*, respectivamente. Los valores de estas propiedades pueden ser asignadas automáticamente por herramientas de análisis de tiempo real, una vez obtenido el modelo de la aplicación completa en la que se utilice el componente.
- Dos propiedades de configuración del componente, *keyboardTh* y *displayTh*, que sirven para asignar prioridades a los dos threads internos que se encargan de la ejecución de las respuestas anteriores. Los valores a asignar a estas propiedades son los valores que las herramientas de tiempo real obtengan para las propiedades anteriores, por ello se mapean internamente a las propiedades correspondientes.

En este ejemplo, la relación entre las propiedades de tiempo real y las de configuración de tiempo real es 1 a 1, pero no siempre tiene porqué ser así.

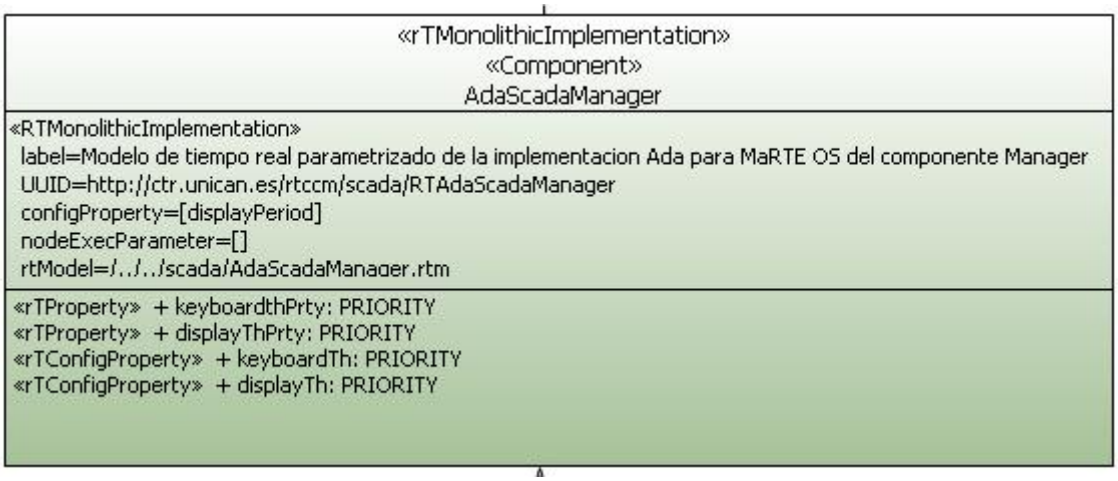

Figura 6.7: Descripción de la <<RTComponentImplementation>> AdaScadaManager

6.2.2. Diseño de plataformas de tiempo real

La única diferencia que podemos apreciar al implementar una plataforma de tiempo real, sin contar los evidentes cambios de los estereotipos, es que los nodos pueden soportar modelos de tiempo real y propiedades configurables para dichos modelos.

En nuestro ejemplo, como vemos en la Figura 6.8, el modelo de tiempo real de ambos nodos es el mismo, y ambos definen una propiedad de tiempo real denominada *speedFactor*. Esta propiedad describe la relación proporcional entre la velocidad de los procesadores de diferentes nodos tomando uno como referencia. Esto es, el modelo que se asigna al nodo modela un nodo con un procesador de una velocidad determinada (750MHz en este caso). Cada instancia de nodo que se añada al despliegue puede corresponder a un nodo con otra velocidad. Asignando el valor adecuado a la propiedad *speedFactor*, podemos reutilizar el mismo modelo para diferentes nodos.

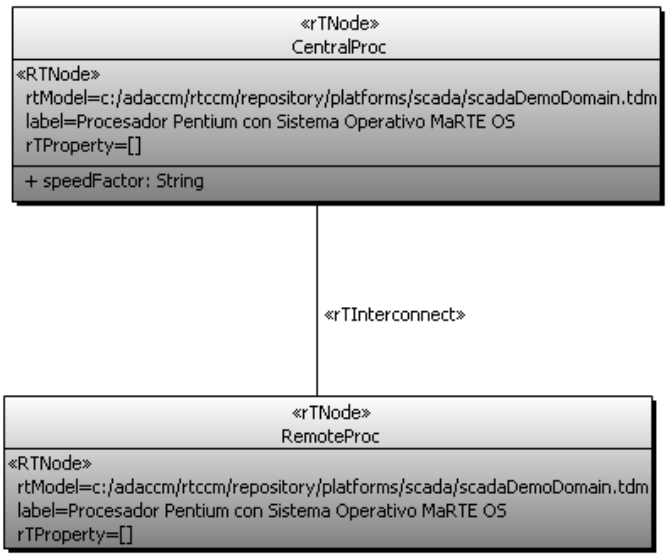

Figura 6.8: Plataforma de ejecución ScadaDemo de tiempo real

6.2.3. Diseño de una aplicación de tiempo real

Para el diseño de la aplicación basada en componentes de tiempo real, tan solo se han de asignar estereotipo a las instancias de las implementaciones de componentes como <<RTDeployedInstance>> y definir valores a sus slots. En la mayoría de los casos será suficiente con asignar valor a las propiedades de tiempo real, estereotipándose el Slot en este caso como <<RTValue>>. Las propiedades de configuración de tiempo real se pueden dejar con sus valores por defecto, pues tomarán valor automáticamente de los valores que se obtengan para sus correspondientes propiedades de tiempo real. Incluso en muchos casos, los valores que se asigna a las propiedades de tiempo real son valores iniciales, que pueden ser luego modificadas por las herramientas.

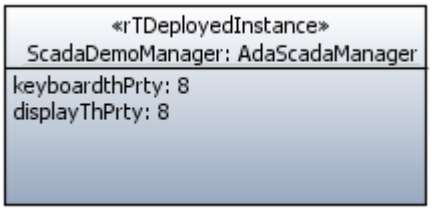

Figura 6.9: Instancia de una implementación de tiempo real

Asimismo, para cada instancia de <<Node>> o <<Interconnect>> utilizado en el despliegue, es necesario asignar valores también a sus propiedades de tiempo real. En este caso, ambos nodos corresponden a procesadores de 1.5GHz, luego reciben un valor de *speedFactor* = 2.

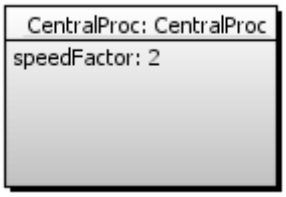

Figura 6.10: Instancia de un nodo de la aplicación de tiempo real

6.2.4. Diseño de una carga de trabajo

El cambio en la definición del plan de despliegue de una aplicación de tiempo real no es muy grande, sin embargo, en este caso no es suficiente con definir el despliegue. Para ser capaces de obtener el modelo de tiempo real de una aplicación es necesario contar con la descripción del despliegue, de la correspondiente plataforma, y también de la carga de trabajo de la aplicación.

Para el diseño de una carga de trabajo para una aplicación de tiempo real, se crea un nuevo proyecto Papyrus en nuestro *workspace*. Este proyecto contendrá un diagrama de clases que será donde se crearán todas las instancias de las respuestas a eventos implementadas en esta aplicación en concreto.

Para ello, se tendrá que importar todos los modelos de componentes utilizados en la aplicación. Cada entidad se instanciará y se le asignará el estereotipo <<RTEndtoEndFlowInstance>>. Sus slots se definirán como <<RTConfigValue>>.

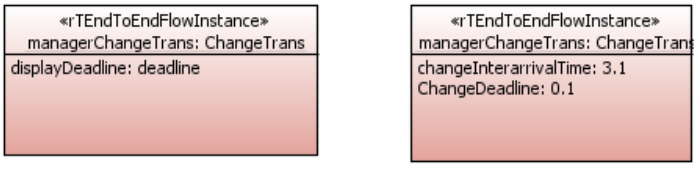

Figura 6.11: Carga de trabajo de ScadaDemo

Asimismo, es necesario importar el correspondiente plan de despliegue para asignar a cada respuesta concreta, la instancia de componente a la que está asociada (atributo *instance* de <<RTEndToEndFlowInstance>>).

Con los tres modelos (despliegue, plataforma y carga de trabajo), la herramienta de composición de modelos puede acceder a los modelos de cada componente y elemento de la plataforma, configurarlos de acuerdo a como se haya expuesto y componerlos para generar el modelo final analizable de la aplicación. Del análisis de dicho modelo se extraen los valores a asignar a las propiedades configurables de tiempo real de los componentes y así se tiene la certeza de que la aplicación cumplirá los requisitos que tiene asignados en tiempo de ejecución.

7. Conclusiones y trabajo futuro

El desarrollo de aplicaciones basado en componentes es una buena metodología a la hora de crear nuevas aplicaciones, sobre todo por el hecho de la reutilización de componentes creados por otras personas y almacenados en repositorios.

El perfil definido en este trabajo da soporte a la descripción de los metadatos asociados a componentes y aplicaciones basadas en componentes, que permiten llevar a cabo el proceso de despliegue de una aplicación satisfaciendo el principio de opacidad típicamente requerido en el desarrollo de aplicaciones basado en componentes.

Con el perfil propuesto, queremos dar una nueva posibilidad a los programadores en cuanto a eficacia y velocidad de creación, evitando la escritura de código, susceptible a errores y haciendo su trabajo en un entorno más cómodo y gráfico. Los diseñadores de una aplicación sólo deben manejar los conceptos que se exponen en el perfil, sin necesidad de acceder a las características internas de los componentes.

Además, con este perfil completamos, en lo que al apartado del despliegue se refiere, una especificación incompleta de un importante grupo como es el OMG, relacionado con la computación.

La extensión del perfil a sistemas de tiempo real permite a los diseñadores de aplicaciones el manejo opaco no sólo del código de los componentes, sino también de sus modelos de tiempo real, con lo que se puede llevar a cabo el análisis de planificabilidad de las aplicaciones de forma automática, sin más que asignar valores a una serie de atributos de los correspondientes estereotipos del perfil.

El trabajo desarrollado hasta el momento ha consistido básicamente en la definición del perfil y su aplicación a un ejemplo concreto. Queda por tanto como trabajo futuro la parte de explotación del perfil, esto es, el desarrollo de todo el conjunto de herramientas que se van a emplear en el proceso de desarrollo y que se van a basar únicamente en los modelos UML anotados con el perfil definido. Algunas de estas herramientas se han identificado ya en el apartado 6, aunque podrían definirse otras más.

Para su desarrollo será necesario utilizar técnicas de transformaciones de modelos, bien de modelo a modelo, utilizando herramientas como ATL, o bien de modelo a texto, con herramientas como Acceleo o Xpand.

Por último, el perfil desarrollado podría integrarse en Eclipse UML2/Papyrus como un perfil registrado, de manera que no haga falta importarlo al propio workspace. Es un

trabajo que se intentó realizar para enriquecer algo más el proyecto realizado, pero que por falta de información no fue posible completar.

8. Bibliografía y referencias

[1] A. W. Brown, "Large-scale, component-based development", Upper Saddle River, N. J.: Prentice Hall PTR, 2000.

[2] C. Szyperski, "Component Software: Beyond Object-Oriented Programing", Addison-Wesley and ACM Press, ISBN 0-201-17888-0, 1998.

[3] I. Crnkovic, S. Larsson y M. Chaudron, "Component-based Development Process and Components Lifecycle", Intl. Conference on Software Engineering Advances, ICSEA'06, IEEE, Tahiti, French Polynesia, Octubre 2006.

[4] Microsoft "The Component Object Model Specification" Report Vol. 99, Microsoft Standards, Redmond, WA: Microsoft, 1996

[5] Object Management Group: UML Superstructure Specification v2.3, OMG Document number: formal/2010-05-05, Mayo 2010.

[6] G. Heineman y W. Councill, editors, "Component-based Software Engineering: Putting the pieces together", Adisson-Wesley, USA, 2001.

[7] Sun Microsystems, "Enterprise JavaBeans Specification", java.sun.com/products/ejb/docs.html, Agosto 2001

[8] Microsoft, ".NET Home Page", http://microsoft.com/net/

[9] Object Management Group: "CORBA Component Model Specification" OMG document number: formal/06-04-01, Abril 2006.

[10] Object Management Group: "Deployment and Configuration of Component-based distributed Applications" OMG document number: formal/2006-04-02, Abril 2006.

[11] Object Management Group: Architecture Borad MDA Drafting Team, Model Driven Architecture – A technical Perspective, OMG Document, ormsc/2001.

[12] P. López Martínez, "Desarrollo de sistemas de tiempo real basados en components utilizando modelos de comportamiento reactivos", Junio 2010.

[13] Object Management Group: Quality of Service for CORBA Components Specification, OMG Document number: ptc/06-04-15, Abril 2006.

[14] Object Management Group, "UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) Beta 2", OMG document number: ptc/2009-05-13, Mayo 2009.

[15] ATESST, web page: http://www.atesst.org/

[16] J. Stankovic, "Misconceptions about real-time computing", Computer, Octubre 1988, pg. 10-19.

[17] Object Management Group, "UML Profile for Schedulability, Performance and Time Specification", Version 1.1, OMG document number: formal/05-01-02, Enero 2005.

[18] Eckerson, Wyane W. “Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications.” Open Information Systems 10, 1 (January 1995): 3 (20).
[19] Object Management Group, web page: http://www.omg.org
[20] Mod-MAST: mast.unican.es/modmast
[21] Altova-UModel: http://www.altova.com/umodel.html
[22] NoMagic-Magic Draw: http://www.magicdraw.com
[23] Commissariat à l'énergie atomique: http://www.cea.fr