Programa Oficial de Postgrado en Ciencias, Tecnología y Computación
Máster en Computación
Facultad de Ciencias - Universidad de Cantabria

Implementación de la tecnología de componentes CCM sobre el middleware de distribución ICE

Laura Barros Bastante
barrosl@unican.es

Directores:
Patricia López Martínez y
José María Drake Moyano.
Grupo de Computadores y Tiempo Real
Departamento de Electrónica y computadores.

Santander, Julio, 2008

Agradecimientos

En primer lugar, agradecer a mis directores de tesis de máster, Patricia López Martínez y José María Drake Moyano por su propuesta, su colaboración e interés, durante el desarrollo del presente trabajo. Nuevamente, a José María Drake, por tratarme como un igual, en la etapa de docencia que he comenzado este año, y ayudarme en las primeras dificultades de esta nueva faceta. A los alumnos, porque estamos aprendiendo juntos.

En segundo lugar, a mis compañeros del Grupo de Computadores y Tiempo Real, por lo aprendido junto y gracias a ellos y por los buenos momentos.

Por último, y como dicen los últimos serán los primeros, a mi familia y a Jacobo, por todo lo que les debo, y lo que me queda por deberles, a mis amigos y amigas, los que he conocido en esta nueva etapa de investigación en la Universidad, que comenzó hace dos años, y los que han estado ahí siempre.

Resumen:

Se aborda el desarrollo de una tecnología de componentes software que se basa en el modelo de referencia CCM (Container Component Model), con componentes especificados de acuerdo con el modelo de despliegue y configuración (D&C), ambos definidos por la organización OMG y en la que la interacción entre componentes distribuidos sea soportada por el middleware de distribución ICE de la organización ZeroC y por los recursos que proporciona este producto.

La tecnología de componentes va a estar orientada al desarrollo de aplicaciones con características de respuesta temporal (performance) especificado, por lo que deberá estar dotada de los recursos y servicios necesarios para el control de los hilos de flujo de control (threads) que ejecutan las actividades de los componentes.

Para la tecnología, se propone el diseño del contenedor que adapta los componentes a la plataforma de ejecución y las herramientas de despliegue que instalan, configuran y lanzan la ejecución de las aplicaciones definidas mediante un plan de despliegue especificado según el estándar D&C.

Financiación:

Este trabajo ha sido financiado por el Ministerio de Educación y Ciencia del Gobierno de España dentro del proyecto THREAD (TIC2005-08665-C03-02), por el Programa IST de la Comisión Europea dentro del proyecto FRESCOR (FP6/2005/IST/5-034026) y por la Red de Excelencia ARTIST2. Este trabajo refleja sólo el punto de vista del autor; la UE no se responsabiliza del uso que se pueda hacer de la información contenida.

Índice

I Ámbito y objetivos del trabajo....1
I.1-Aplicaciones distribuidas: Requisitos de las aplicaciones y aportaciones del middleware. 1
I.2-Middleware Ice de ZeroC. 2
I.2.1-Principales características. 2
I.2.2-Limitaciones de un middleware no orientado a componentes. 2
I.3-Tecnología de componentes. 3
I.4-CCM (Container Componet Model). 3
I.5-Especificación D&C de OMG. 4
I.6-Justificación de la elección de CCM y D&C para la plataforma de sistemas de control y seguridad. 5
I.7-Objetivos de la tesis. 7
II Especificación, diseño y empaquetamiento de componentes Java-CCM. 9
II.1-Proceso de desarrollo de un componente. 9
II.2-Especificación UML2.0 de un componente. 10
II.3-Especificación D&C de un componente CCM. 12
II.3.1 Clases raíces para la descripción de los componentes. 12
II.3.2-Descripción de la interfaz de un componente. 13
II.4-Implementación Java de un componente. 15
II.5-Descripción D&C de la implementación de un componente. 18
II.6-Empaquetamiento de un componente Java. Descripción D&C del paquete. 20
III Diseño del contenedor de un componente Java-CCM 22
III.1- Modelo de referencia CCM. Elementos y funcionalidad. 22
III.1.1-Adaptación del componente a la plataforma. 22
III.1.2 Estructura interna de un componente 22
III.2 Implementación Java del Contenedor 24
III.2.1-Implementación de las Interfaces de la tecnología CCM. 24
III.2.2-Implementación Java del Wrapper. 26
III.3-Implementación Java de Executor 28
III.4-Implementación Java de Contexto. 29
III.5-Implementación Java del gestor de threads. 31
III.6 Implementación Java del Home. 33
III.7-Plan de despliegue y herramienta de despliegue local. 36
III.7.1-Plan de despliegue local. 36
III.7.2-Herramienta de despliegue local. 37
IV Ejemplo de aplicación Homeland Alarm 41
IV.1-Demostrador Homeland: Especificación y arquitectura. 41
IV.2-Catálogo de componentes. 43
IV.3-Modelo de despliegue de la plataforma. Descripción D&C. 45
IV.4-Plan de despliegue. Descripción D&C. 48
Conclusiones y trabajo futuro 52
Referencias 54

Índice de figuras

Figura 1.1. Middleware en las aplicaciones de seguridad y control. 1
Figura 1.2.1.Resumen de características de Ice 2
Figura 1.5. Plantillas W3C- *schema* de la especificación D&C 5
Figura 2.1. Proceso de desarrollo de un componente 9
Figura 2.2. Elementos que definen la funcionalidad de negocio y de gestión de SoundGenerator. 11
Figura 2.3.1. Clases raíces para la descripción de los componentes... 13
Figura 2.4.1. Estructura del código de negocio de un componente 16
Figura 2.4.2. Descripción del código de negocio de la implementación JSSoundGenerator................ 18
Figura 3.1.1: El contenedor adapta el componente a la plataforma. 22
Figura 3.1.2. Estructura de la implementación de un componente 23
Figura 3.2.1 Diagrama de clases de la implementación Java-CCM de un componente. 25
Figura 3.3. Diagrama de clases del Ejecutor del Componente SoundGenerator 28
Figura 3.4. Diagrama de clases del Contexto del componente SoundGenerator 30
Figura 3.5. Elementos que definen la funcionalidad de negocio y de gestión de MediaSource. 32
Figura 3.6.1. Diagrama de clases del Home 34
Figura 3.6.2 Ficheros del componente ejecutable Java-CCM. 36
Figura 4.1.1.Arquitectura del demostrador Homeland 41
Figura 4.1.2. Arquitectura software de la aplicación Homeland Alarm 43
Figura 4.3.1. Clases raíces para la descripción de una plataforma. 45
Figura 4.3.2.Plataforma y distribución de las instancias de los componentes 46
Figura 4.4. Clases raíces que sirven para definir el plan de despliegue de una aplicación. 48

Abreviaturas y acrónimos

CCM	Container Component Model
Ice	Internet Communication Engine
Java-CCM	Modelo de Componentes basado en Contenedores para Ice desarrollado en Java
SLICE	Specification language for ICE
IDL	Interface Description Language
W3C	World Wide WEB Consortium
XML	Extensible Markup Language
OMG	Object Management Group
IST	Information Society Technologies European Programme
ITEA	Information Technology for European Advancement
RTEP	Real-Time Ethernet Protocol
CAN	Controller Area Network
JMF	Java Media Framework
PIM	Platform Independent Model
PSM	Platform Specific Model
BIM	Business Interface Management
D&C	Deployment and Configuration of Component-based Applications Specification

I Ámbito y objetivos del trabajo.

I.1-Aplicaciones distribuidas: Requisitos de las aplicaciones y aportaciones del middleware.

El presente proyecto se ha realizado en el ámbito de un proyecto de investigación [1] destinado a generar las tecnologías que se necesitan para el desarrollo de sistemas informáticos cuyo objetivo es garantizar la seguridad de espacios públicos (aeropuertos, áreas comerciales, museos, etc.) e infraestructuras estratégicas (centrales nucleares, presas, tendidos ferroviarios, etc.). Estos sistemas son ejemplos paradigmáticos de sistemas distribuidos y heterogéneos, en los que se utilizan equipos y dispositivos hardware distribuidos por extensas áreas geográficas, y que deben ser supervisadas y controladas tanto desde puntos interiores a esas zonas, como desde puntos remotos externos a ellas. Con la actual tecnología de computadores, se consiguen las mejores prestaciones, la mayor fiabilidad, y el menor costo, si se utiliza una arquitectura distribuida constituida por un número arbitrario de computadores interconectados entre sí a través de una red de comunicaciones, de forma que se pueda ubicar un computador en la proximidad de cada equipo o grupo de equipos (o incluso en el interior de ellos).

El diseño y despliegue de una aplicación informática en una plataforma distribuida de esta naturaleza es compleja, ya que requiere añadir a las dificultades propias de diseño de una aplicación compleja, el conocimiento detallado de la propia plataforma. La necesidad de conocer y ser experto del propio dominio de las aplicaciones y del sistema (diferentes sistemas operativos y servicios de comunicaciones) dificulta considerablemente el desarrollo de las aplicaciones distribuidas. La solución a este problema ha sido introducir el concepto de *middleware* (figura 1.1). Éste es una capa software que se instala por encima de los servicios del sistema (sistema operativo y servicios de comunicaciones) de cada nodo, y que ofrece a las aplicaciones un entorno de recursos y servicios uniforme que es independiente de la naturaleza de los elementos de la plataforma de ejecución. El *middleware* ofrece recursos de ejecución, monitorización y control de las aplicaciones distribuidas que son equivalentes a los que ofrece un sistema operativo en una plataforma monoprocesadora. Dado que la transmisión de señales de audio y vídeo en tiempo real tiene mucha relevancia en las aplicaciones de seguridad y control, también debe preverse que el *middleware* sea compatible con los protocolos requeridos por esta funcionalidad.

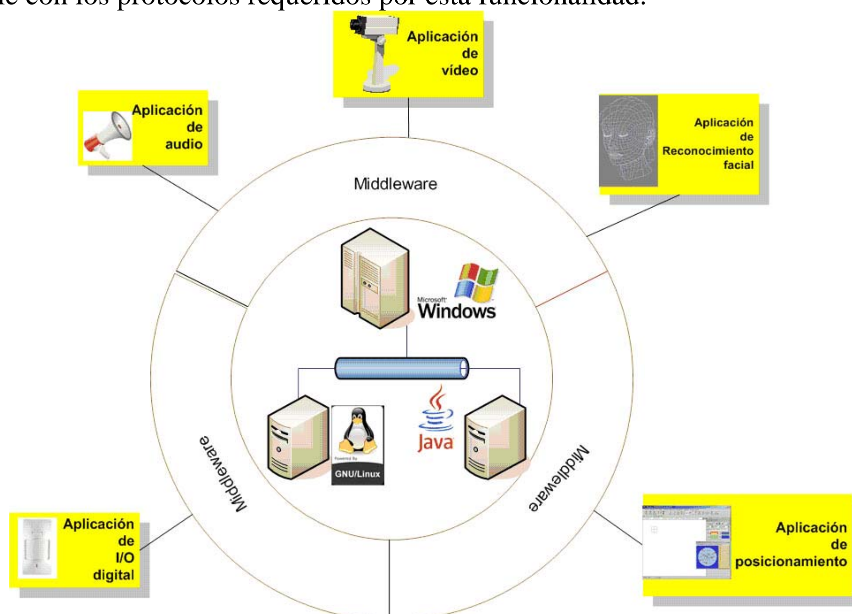

Figura 1.1. Middleware en las aplicaciones de seguridad y control.

El trabajo que se presenta en esta memoria ha consistido en evaluar el *middleware* ICE de la compañía ZeroC para su utilización en las plataformas de los sistemas de seguridad y control de infraestructuras que se abordan en el citado proyecto.

I.2-Middleware Ice de ZeroC.

I.2.1-Principales características.

ICE (Internet Communications Engine) es un *middleware* orientado a objetos que soporta varios lenguajes (figura 1.2.1). Es utilizado por aplicaciones *mission-critical* (FCS de Boeing/SAIC) líderes en el mercado.

Ice es software libre, estando disponible el código fuente completo y ofrecido bajo los términos de Licencia Pública General de GNU (GPL). Existen licencias comerciales para los clientes que deseen usar ICE para código propietario.

Nombre:	Internet Communications Engine (ICE)	
Definido por:	ZeroC Inc. (www.zeroc.com)	
Documentación:	http://www.zeroc.com/download/Ice/3.1/Ice-3.1.1.pdf	
Lenguajes:	C++, Java, C#, Visual Basic, Python, PHP	
Plataforma:	Windows, Windows CE, Unix, GNU/Linux, *BSD, OSX, Symbian OS, J2RE 1.4 o superior, J2ME	
Destacado:	APIs limpios y bien diseñados. Conjunto de servicios muy cohesionados. Amplia variedad de características avanzadas (despliegue, persistencia, multi-protocolo, cifrado, compresión de datos, ...)	
Descargas:	http://www.zeroc.inc/download.html	
	ZeroC Ice, ZeroC IceE	*GPL*

Figura 1.2.1.Resumen de características de Ice

I.2.2-Limitaciones de un middleware no orientado a componentes.

Desde el punto de vista técnico Ice hereda la mayoría de los conceptos de CORBA (el equipo técnico está compuesto por ocho especialistas en CORBA encabezados por Michi Henning, ex-miembro de OMG), pero se libera de las ataduras que impone el diseño por comité y la compatibilidad hacia atrás con protocolos obsoletos y no escalables. En este sentido logra mayor rendimiento, mayor simplicidad de uso, mayor número de características avanzadas y un diseño muy cohesionado [2,3].

Puede parecer sencillo, dadas las similitudes semánticas de ICE y CORBA, implementar aplicaciones que puedan compilarse indistintamente con soporte de CORBA o de ICE. Esto no es cierto, puesto que la mayoría del código que utiliza el middleware es dependiente de los servicios empleados.

En particular, una diferencia importante, entre ambos middleware, es el hecho de que Ice no sea un middleware orientado a componentes [4]. Un componente es una implementación software opaca con una funcionalidad bien definida, que ha sido diseñada para su reutilización por composición, en futuras aplicaciones, y que opera, y ha sido empaquetado de acuerdo con un modelo.
Las limitaciones más relevantes que se encuentran en un *middleware* orientado a objetos son:

- Define la funcionalidad de los objetos a través de interfaces declaradas como contratos entre clientes y servidores, siendo un requerimiento que los componentes que dependan de otros objetos deban conectarse explícitamente a ellos. Esta conexión explícita conduce a implementaciones frágiles y no reusables.
- No se especifica un modelo de referencia genérico de servidor, de manera que en cada servidor debe especificarse de forma particular las tareas de inicialización, las políticas de calidad de servicio, la gestión del entorno, etc. Estas tareas hacen que las aplicaciones sean más complejas, menos reutilizables y menos flexibles.
- El administrador de la plataforma es el encargado de pasar el software a cada procesador, siendo también el responsable de configurarlo adecuadamente en función del servidor que instanciará, y siendo también el responsable de ejecutar las aplicaciones de cada procesador. Esto conduce a sistemas difíciles de mantener.

I.3-Tecnología de componentes.

Actualmente la tecnología de componentes software se ve como la mas prometedora (si no la única) solución para proveer el incremento de productividad, la fiabilidad de los sistemas y la calidad de servicio que requiere el avance de la tecnología de la información. El hecho de no disponer de una tecnología de componentes para el *middleware* ICE, puede ser un handicap frente a otros *middleware*, tales como CORBA, que dispone del éstandar CCM (CORBA Component Model) [5,6].

Desarrollar una tecnología de componentes para el *middleware* ICE, y por tanto, eliminar las limitaciones enumeradas anteriormente, es el reto del presente trabajo.

El objetivo esencial de una arquitectura basada en componentes, es poder construir aplicaciones ensamblando componentes reutilizables que se encuentran disponibles y que han sido diseñados con independencia de las aplicaciones en las que se utilizan. Un componente es un módulo software que ofrece de forma auto-contenida y opaca:

- La especificación de la funcionalidad y servicios ofertados (contrato de uso).
- Los servicios externos a él que necesita para implementar su funcionalidad y los recursos que requiere para ser ejecutado (contrato de instanciación).
- La información introspectiva (*metadata*) que se necesita para su manejo y ensamblado, en particular, por parte de entornos de diseño y desarrollo basados en herramientas.
- Los elementos de código ejecutable que lo implementan, y uno o múltiples entornos de ejecución.

Además, empaquetado todo como un producto integrado y estandarizado, que permite su ensamblado en una aplicación sin necesidad de conocer los detalles de su naturaleza y diseño.

Los componentes deben haber sido diseñados para poder ser ejecutados en diferentes plataformas sin necesidad de modificar, y ni siquiera conocer, su código interno.

I.4-CCM (Container Componet Model).

Varios proyectos europeos, MERCED[7], COMPARE[8] y FRESCOR[9] abordan, desde diferentes puntos de vista, el desarrollo de una tecnología de componentes compatible con los requisitos que presentan los sistemas embebidos, distribuidos y de tiempo real. Todos ellos toman como punto de partida el modelo CCM (Container/Component Model) que sigue

la especificación LwCCM [10], pero eliminan la dependencia de CORBA como sistema de comunicación. Siguiendo este patrón, el código de negocio del componente se puede desarrollar de forma totalmente independiente de la tecnología subyacente, siendo el contenedor del componente el que ofrece todos los recursos necesarios para adaptar dicho código de negocio a la plataforma y tecnología correspondiente. Dicho contenedor es generado de forma totalmente automática a partir de la información introspectiva (metadata) que proporcionan los componentes.

Ejemplos de tecnologías desarrolladas siguiendo el modelo CCM son:

- MicroCCM[11], destinada a sistemas embebidos y basada en un *middleware* CORBA propio y reducido para sistemas con recursos limitados, desarrollada por la empresa Thales.
- Ada-CCM[12] destinada a sistemas embebidos de tiempo real y que se comunican a través de *sockets* con protocolo RTEP y bus CAN, desarrollada dentro del grupo de Computadores y Tiempo Real (CTR) de la Universidad de Cantabria.
- Java-CCM[13], objeto del presente trabajo, basada en la adaptación de Ice, un middleware no orientado a componentes, a la tecnología CCM .

I.5-Especificación D&C de OMG.

El propósito de la especificación D&C (Deployment and Configuration of Component-based Applications Specification) [14] es formalizar los procesos y la información que se gestiona en el despliegue de las aplicaciones distribuidas basadas en componentes. El despliegue es entendido en esta especificación como el conjunto de tareas a desarrollar entre la adquisición del software elaborado y la ejecución del mismo sobre una plataforma. Esto requiere especificación para las tareas en lo referente a:

- Descripción de los requerimientos de despliegue del software.
- Mecanismos de empaquetamiento del software y de su información introspectiva (*metadata*) para su distribución desde el diseñador hasta el instalador (*planner*).
- Almacenamiento del software en el entorno de diseño de las aplicaciones antes de que se tomen las decisiones de despliegue sobre una plataforma determinada.
- Describir la topología, los recursos y las capacidades de las plataformas en las que se despliegan las aplicaciones.
- Planificar el despliegue de las aplicaciones, esto es, tomar las decisiones de cómo se distribuyen las instancias de los componentes en los nudos y cómo se hace uso de la infraestructura de ejecución.
- Transferir y preparar los nudos y recursos para que puedan alojar el software que deben ejecutar.
- Lanzar la ejecución, monitorizar y, en su caso, terminar la ejecución de la aplicación.

El objetivo del estándar es unificar las diferentes especificaciones propuestas por los promotores de tecnologías y plataformas a fin de hacer inter-operables las herramientas de las diversas empresas y que deben integrarse para construir los entornos de desarrollo de las aplicaciones basadas en componentes.
Hay dos razones por las que se ha preferido esta especificación a la establecida en la especificación Corba Component Model de OMG:

- Los servicios y las herramientas de la especificación Corba Component Model están muy influidas y condicionadas por la plataforma CORBA, que en la tecnología CCM no se utiliza.
- Al estar la especificación D&C formulada de forma abstracta e independiente de cualquier tecnología, ha sido dotada de los medios necesarios para que sea adaptada a cualquier otra y en particular a la Java-CCM que se desarrolla. Una de las

principales ventajas es que está formulada mediante un modelo UML que permite su extensión para incorporar nuevos aspectos de diseño, como la gestión de recursos o el comportamiento temporal.

La adaptación del estándar D&C a la tecnología Java-CCM se ha realizado a través de la elaboración de un conjunto de plantillas W3C-*schema* que definen el contenido y el formato de los documentos XML que describen las interfaces, las implementaciones, los paquetes de los componentes, la descripción de las plataformas distribuidas y la formulación del plan de despliegue que definen las aplicaciones basadas en componentes. En la figura 1.5, se muestran las cinco plantillas que cubren el estándar y las dependencias entre ellas.

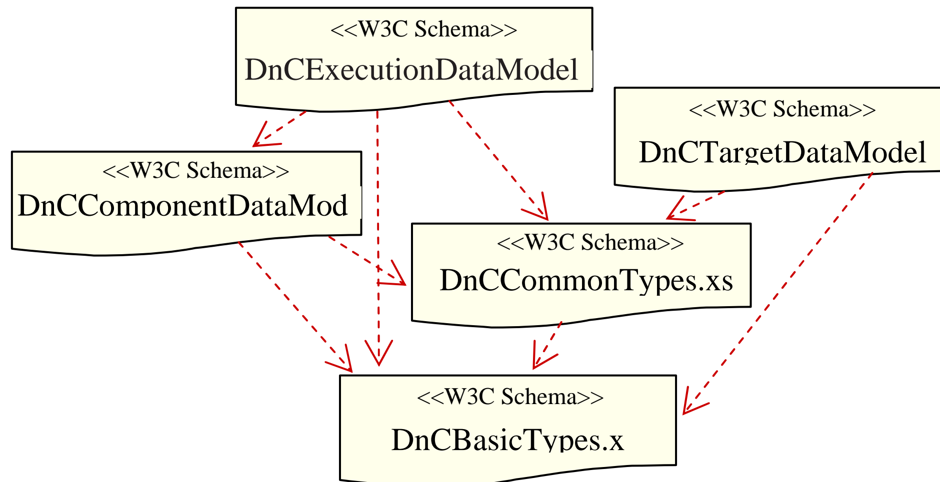

Figura 1.5. Plantillas W3C- *schema* de la especificación D&C

I.6-Justificación de la elección de CCM y D&C para la plataforma de sistemas de control y seguridad.

La elección de la arquitectura CCM se justifica por satisfacer los requisitos de las plataformas propias de los sistemas de control y seguridad como por ejemplo:

- **Facilitar una arquitectura orientada a servicios:** La arquitectura cliente/servidor proporciona un ámbito idóneo para el desarrollo de la tecnología de seguridad, video interactivo y control de infraestructuras.
- **Orientada a una arquitectura cliente/servidor multi capa:** En una estrategia cliente/servidor de múltiples capas (*n-tier client/server*) los clientes son muy ligeros, y delegan en servidores intermedios todos aquellos aspectos de su funcionalidad que se puedan considerar estables y por tanto van a ser reutilizadas en múltiples aplicaciones.
- **Plataforma distribuida y heterogénea:** La naturaleza de muchos de los procesadores va a estar condicionada por los equipos y dispositivos hardware que deben controlar.
- **Compatible con múltiples lenguajes de programación:** Las plataformas tienen capacidad de operar sobre varios sistemas operativos (UNIX/LINUX, WINDOWS NT y JAVA), hecho condicionado por los diferentes equipos y dispositivos hardware presentes en este tipo de sistemas. Asimismo, los componentes software que se integran en las aplicaciones son legados o han sido desarrollados para otras plataformas y requieren un determinado tipo de procesador o de sistema operativo.
- **Proporcionar un *middleware* para las aplicaciones distribuidas:** Proporciona a los diseñadores un entorno uniforme sobre el que desarrollar sus aplicaciones.

- **Facilitar la escalabilidad**: El dominio de tecnologías de seguridad requiere tratar plataformas y aplicaciones cuya complejidad puede necesitar ser escalada en varios órdenes de magnitud. Desde aplicaciones sencillas sobre una plataforma mono-procesadora, hasta aplicaciones complejas que requieren cientos de procesadores.

Dichos requerimientos, son satisfechos per se por el uso del middleware. Los siguientes requerimientos justifican el uso de una tecnología de componentes:

- **Fiable y robusta frente a fallos :** Esto requiere que la plataforma ofrezca recursos y servicios para:
    - Garantizar la operatividad de cada aplicación frente a fallos de otras aplicaciones de la plataforma.
    - Detectar las aplicaciones o los componentes de las mismas con fallos.
    - Reconfigurar las aplicaciones o sustituir los componentes con fallos para mantener su operatividad.
- **Fiable y robusta frente a sobrecargas:** Se requiere disponer de herramientas capaces de detectar y gestionar las situaciones de sobrecarga.
- **Capacidad de monitorizar y controlar el estado y el uso de recursos de las aplicaciones**: Para satisfacer todos estos tipos de requisitos no funcionales relacionados con el acceso a los recursos de la plataforma, se necesita que los servicios de la plataforma que gestionan los recursos tengan capacidad de conocer el uso que de ellos hace la aplicación, y así mismo, controlar el acceso de la aplicación a ellos. Con la utilización de la metodología de componentes basadas en contenedor, se ofrece la oportunidad de liberar al diseñador de tener que tratar estos aspectos, dejándolos bajo la responsabilidad del contenedor que está especializado en la adaptación a la plataforma
- **Gestión de configuración, despliegue y lanzamiento de las aplicaciones distribuidas**: En una aplicación basada en componentes la gestión de todas estas operaciones es uniforme y está guiada por el plan de despliegue de la aplicación, y no por el código del componente, ni por la programación que se haga de la plataforma. Las aplicaciones basadas en componentes se describen a través del plan de despliegue que ofrece todos los datos que la herramienta de despliegue y configuración requiere para llevar a cabo estas operaciones de forma automática, y sin necesidad de conocer el código de los componentes. El estándar D&C de OMG define la información del plan de despliegue de las aplicaciones distribuidas basadas en componentes, así como la información introspectiva que debe incluir cada componente para su gestión.
- **Capacidad de integrar subsistemas especializados de tiempo real o embarcados**: En los servicios de seguridad no se consideran aplicaciones distribuidas con requisitos de tiempo real estricto. Sin embargo, sí que debe preverse que existan otros sistemas auxiliares, que operen dentro de ellos, con estos requisitos u otros semejantes, y cuya configuración, acceso a la información que generan y control deben poder hacerse desde aplicaciones que operan en este tipo de plataformas.
- **Responder en sus especificaciones a estándares internacionales.**: Utilizar un *middleware* propietario como es ICE, que sólo tiene un suministrador, es una estrategia razonable para el desarrollo de un sistema concreto siempre que ICE satisfaga sus requisitos. Sin embargo utilizar un *middleware* propietario para el desarrollo una línea de productos que van ser empleados en futuras aplicaciones, tiene los riesgos de que la vigencia y disponibilidad de ICE no está asegurada y no dispone de sustituto. La introducción de la metodología de componentes busca reducir este riesgo. El código de negocio de los productos (que es lo que realmente tiene valor) se desarrolla absolutamente independiente de la tecnología de la

plataforma y solo está condicionado por el modelo de referencia de la tecnología que corresponde a un estándar, y por tanto, tiene una vigencia mayor.

I.7-Objetivos de la tesis.

El objetivo básico del presente trabajo es adaptar la tecnología CCM de diseño de aplicaciones basada en componentes para que pueda ser utilizada en plataformas distribuidas y heterogéneas, que estén dotadas del *middleware* ICE, no orientado a componentes. En nuestro caso, nos hemos restringido a componentes codificados en lenguaje Java y que se instancian en JVM (Java Virtual Machine). En otro trabajo desarrollado dentro del mismo proyecto, se ha desarrollado una versión similar para el lenguaje C++.

Requisitos importantes de la tecnología que se establecieron al inicio del trabajo, y que se han conseguido, son:

- *Reutilización del código de negocio*. El código de negocio que proporciona la funcionalidad de los componentes debe estar completamente desacoplado del middleware ICE que constituye su soporte básico en la plataforma. Esto es, se ha desarrollado una configuración software que hace posible formular el código de negocio sin que exista en él referencias al middleware ICE. Toda las referencias a ICE se encuentran en el contenedor. Esto es muy importante, porque con ello el código de negocio de los componentes puede ser reutilizado (sin modificación) en diferentes tecnologías ya sea el *middleware* ICE, CORBA u otro equivalente. Obviamente para poder reutilizar un componente en una determinada tecnología hay que disponer de los correspondientes generadores de código para la nueva plataforma.
- *Se utiliza el modelo de referencia (framework) estandarizado CCM de OMG*. Este modelo se utiliza para la especificación de los componentes, de la plataforma y de las aplicaciones. Una consecuencia muy favorable de utilizar el estándar es que la tecnología es compatible con las herramientas de despliegue y control desarrolladas para el mismo estándar por otras empresas.
- *Entorno de desarrollo de aplicaciones basado en herramientas automáticas*. El criterio principal que ha guiado la especificación de la arquitectura ha sido facilitar la generación automática del código. A tal fin, los códigos de los elementos del contenedor se han formulado mediante plantillas sencillas parametrizadas.

Los aspectos que se han desarrollado en este trabajo para adaptar la tecnología CCM al *middleware* ICE son:

- Se ha adoptado el lenguaje SLICE (Specification Language for ICE) [2] para formular las interfaces que definen la funcionalidad de los componentes. Con ello, se consigue la compatibilidad de los componentes desarrollados en Java con los codificados en otros lenguajes.
- Se ha adaptado a la tecnología Java-CCM las plantillas W3G-Schema que definen los formatos y los contenidos de los modelos de los componentes, de las aplicaciones y de las plataformas, de acuerdo con el estándar D&C de OMG.
- Se ha diseñado la arquitectura de los contenedores que adaptan el código de negocio a la plataforma y hacen posible su ejecución en ella. En ella, se integran los recursos y servicios del middleware ICE que implementan la interacción entre los componentes.
- Se ha formulado las reglas para la generación automática del código Java de los contenedores de los componentes.
- Se ha formulado las estrategias de configuración de los componentes haciendo uso de las propiedades ICE.

- Se ha desarrollado un prototipo muy sencillo de gestor de componentes, que permite la gestión del ciclo de vida de los componentes que se instalan en un nudo de la plataforma en el contexto de la ejecución de una aplicación.

Finalmente, se ha utilizado como demostrador de la tecnología un ejemplo denominado Homeland Alarm. El ejemplo es completo y permite describir muchas de las características que se implementan en la tecnología, como por ejemplo, la especificación D&C de la plataforma y del plan de despliegue de la aplicación.

II Especificación, diseño y empaquetamiento de componentes Java-CCM.

II.1-Proceso de desarrollo de un componente.

Durante el proceso de desarrollo de una aplicación, se distinguen dos partes diferenciadas, el proceso de desarrollo de los componentes, como unidades independientes distribuibles y comercializables que pueden ser reutilizadas en aplicaciones futuras y el proceso de desarrollo de las aplicaciones en una plataforma concreta, construyéndolas como conjuntos de componentes ensamblados ofreciendo las funcionalidades requeridas por sus especificaciones.

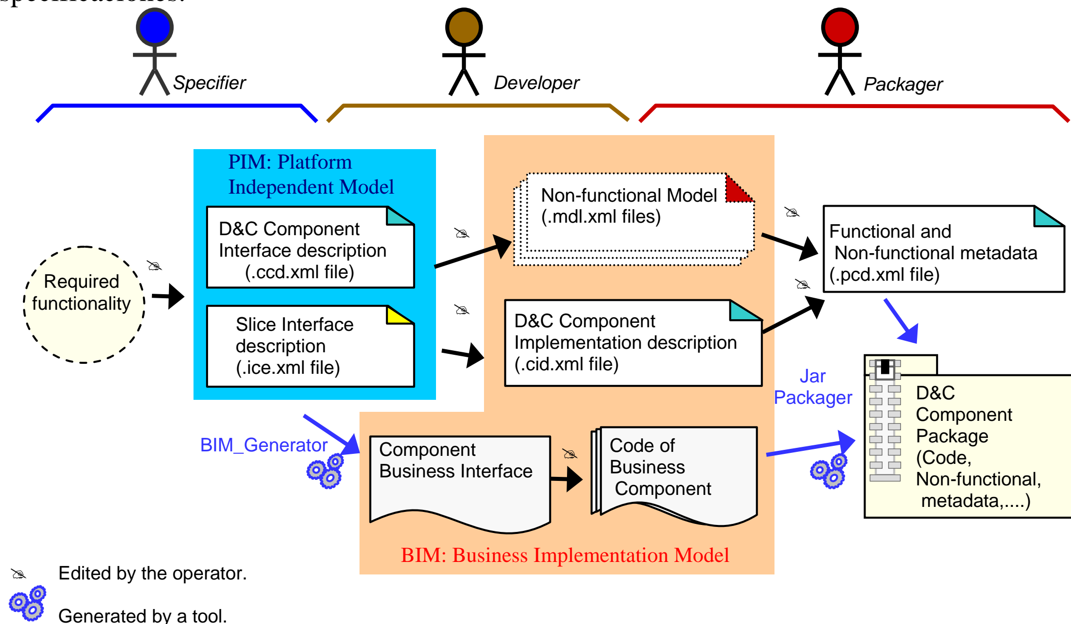

Figura 2.1. Proceso de desarrollo de un componente

En la figura 2.1, describimos el proceso de desarrollo de un componente y que es el objeto de este capítulo.

El especificador (specifier) es el experto en el dominio de la aplicación. Realiza el modelo PIM (*Platform Independent Model*) que describe el comportamiento del componente con independencia de su implementación o del mecanismo middleware con el que se accede a él. Es el modelo que utiliza el desarrollador de las aplicaciones al integrarlo en sus diseños, para decidir la idoneidad de su funcionalidad, sus posibilidades de configuración y cómo debe ser utilizado. En concreto, cuando el especificador detecta una funcionalidad no cubierta en el dominio:

- Realiza la especificación del componente que satisface dicha funcionalidad, de acuerdo a la especificación D&C, a través del fichero de la descripción de la interfaz de un componente.
- Realiza la descripción de cada una de las interfaces implicadas en la funcionalidad del componente, a través del lenguaje SLICE.

El desarrollador (developer) realiza el modelo BIM (*Business Implementación Model*). Describe una implementación concreta del componente en la que se han definido los recursos en que basa su funcionalidad. Es el modelo que se introduce para que el diseñador que desarrolla el código de negocio que implementa el componente, pueda trabajar de forma libre y sin necesidad de conocer la tecnología de distribución subyacente. Un mismo componente puede tener múltiples implementaciones con diferentes requerimientos en cuanto a recursos de la plataforma. Todas ellas pueden ser adaptadas a la tecnología de una plataforma específica utilizando herramientas automáticas. En concreto:

- Escribe el código de negocio del componente en el lenguaje como si lo fuera a ejecutar en un monoprocesador. El código de negocio deberá implementar la interfaz de negocio del componente, que permite, entre otras cosas, gestionar el control de vida del componente.
- Realiza el modelo no funcional que incluye parámetros de configuración (no funcionales) y referencias a otros modelos no funcionales de los otros componentes con los que interactúa [15].
- Realiza la descripción y modelo de las implementaciones (Implementation descriptions): indicando como se debe usar e instalar el componente.

El empaquetador (packager) construye el paquete que se puede distribuir y comercializar. Contiene:

- Código del componente.
- Modelo de comportamiento no funcional.
- Modelo de uso del componente, que permite conocer y gestionar la funcionalidad del mismo sin conocer el código interno.

A continuación, describimos el proceso de desarrollo de un componente concreto, ***SoundGenerator*****,** sencillo generador de sonidos implementado utilizando los recursos que ofrece la propia plataforma Java 2 SE 5.0 a través de las clases incluidas en el paquete javax.Sound.Sampled. Sirve para interpretar segmentos de sonidos sobre la tarjeta de sonidos del propio PC. Las facetas (interfaces que ofrece) , receptáculos (interfaces que requiere), atributos y constructores con los que se gestiona el componente se describe en el fichero .ccd.xml (*SoundGenerator.ccd.xml*). Esta descripción está formulada de acuerdo con el estándar D&C de OMG.

II.2-Especificación UML2.0 de un componente.

Los elementos que definen la funcionalidad de negocio y de gestión de un componente, se describen en el diagrama de clases de la figura 2.1, a través del ejemplo del componente SoundGenerator[1].

La funcionalidad externa de cualquier implementación de un componente se define mediante su interfaz externa. Esta incluye:

- ***Tipos del domini*****o**: Por ejemplo, el componente SoundGenerator utiliza el tipo enumerado AlarmSoundType que define los identificadores de los sonidos que están predefinidos en el componente.
- ***Interfaz de gestión*** (*SoundGeneratorMng*): Constituye la interfaz a través del que el contenedor gestiona el ciclo de vida del componente. Representa la funcionalidad de gestión que se requiere en cualquier componente Java-CCM.
    - *Funciones de acceso a las facetas del componente:*
        - *getPlayerPortFacet()*: función de navegación con la que se tiene acceso a la faceta PlayerPortFacet.
    - *Operaciones de conexión con los receptáculos:*
        - *setTheLogger():* procedimiento de conexión de los receptáculos con las facetas de los componentes externos con los que se conectan.
    - *Operaciones de configuración:*

[1] A fin de hacer más legible la memoria hemos optado por utilizar el ejemplo concreto del componente SoundGenerator para la explicación de los conceptos, en vez de utilizar un ejemplo genérico con nombres irrelevante.

setDingSound(), setChordSound(), setDefSound(), setChimesSound(), setAlarmSound() y setFailSound() : con ellos se asigna el valor a los parámetros de configuración del componente (*dingSound, chordSound, defSound, chimesSound, alarmSound* y *failSound*).

- *Operaciones de gestión del ciclo de vida del componente:* *activate()*, *passivate()* y *remove()*: se utilizan para controlar el ciclo de vida del componente.

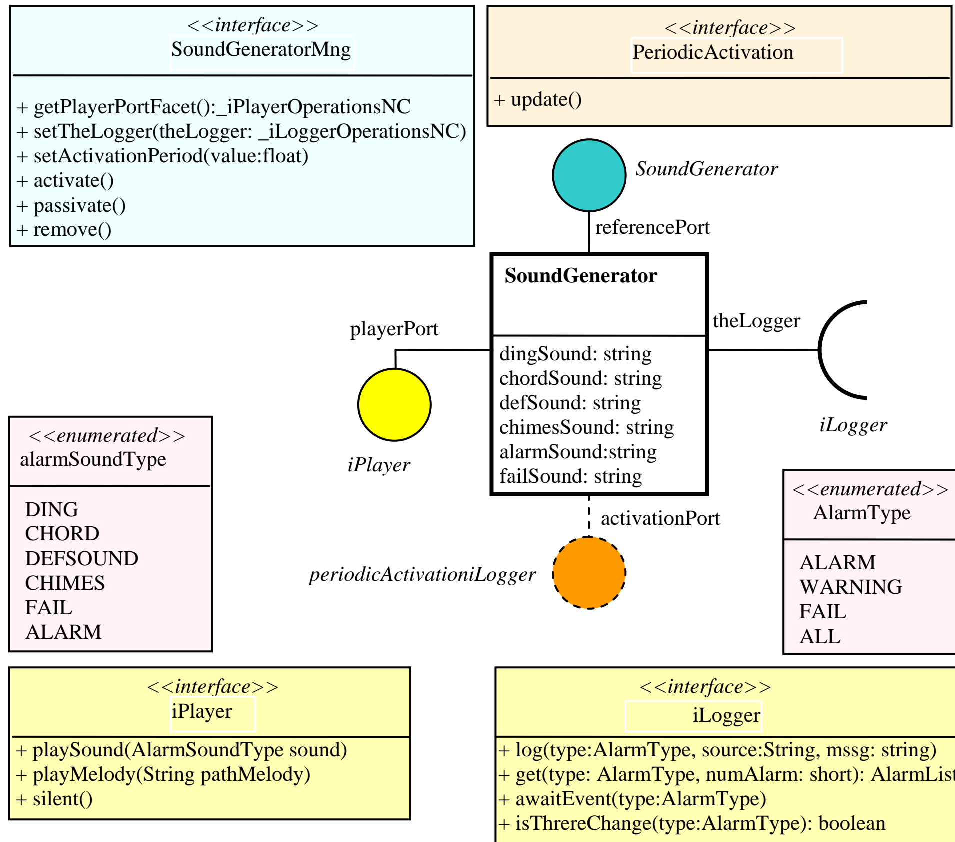

Figura 2.2. Elementos que definen la funcionalidad de negocio y de gestión de SoundGenerator.

- **Puertos de negocio:** El comportamiento funcional (de negocio) externo de un componente se describe a través de sus puertos.
  - **Facetas** (*playerPort*): ofrece los servicios de negocio del componente que puede invocar un cliente (*playMelody(), playSound()* o *silent()*). Implementa la interfaz que define su funcionalidad (*iPlayer*). La descripción formal de la interfaz se describe en el módulo Slice con el nombre del dominio (*multimedia*) del fichero que contiene la interfaz (*iPlayer.ice*).
  - **Receptáculos** (*theLogger*): a través de ellos se requieren los servicios (*log()* para el reporte de errores en la carga o reproducción del sonido) del componente accedido. Cada receptáculo espera ser conectado a una faceta que implemente la interfaz que corresponda (*iLogger*). La descripción formal de la interfaz se realiza en el módulo con el nombre del dominio (*database)* del fichero que alberga la interfaz (*iLogger.ice*).
- **Puertos de activación** (no ofrecidos por el componente SoundGenerator): implementan una de las dos interfaces *PeriodicActivation* o *OneShotActivation* definidas en la tecnología Java-CCM. Cuando el componente es instanciado en una aplicación, el contenedor reconoce estos puertos, y por cada uno de ellos, crea un thread y haciendo uso de él ejecuta el procedimiento definido en la interfaz. Si

el puerto implementa la interfaz PeriodicActivation, el thread invocará periódicamente el método *update()*, y si la interfaz implementada por el puerto es OneShotActivation, invocará una sola vez el procedimiento *run()*. Por cada puerto PeriodicActivation que implemente el componente, este debe definir una propiedad *<nombre del puerto>Period* con la que se define en configuración el periodo de invocación del método update().

II.3-Especificación D&C de un componente CCM.

La adaptación del estándar D&C a la tecnología CCM se ha llevado a cabo a través de la adaptación de un conjunto de plantillas W3C-S*chema* que definen el contenido y el formato de los documentos XML que describen las interfaces, las implementaciones, los paquetes de los componentes, la descripción de las plataformas distribuidas y la formulación del plan de despliegue que definen las aplicaciones basadas en componentes. Todas las implementaciones de un componente corresponden a una especificación del componente.

Todos los elementos mostrados en el modelo UML que definen la funcionalidad del componente (puertos), y que sirven para describirlo, además de las propiedades de configuración, se recogen en un documento XML que sigue la especificación D&C de la interfaz de un componente. Dicho documento, sigue el *schema* de la plantilla *DnCComponentDataModel.xsd.*

II.3.1 Clases raíces para la descripción de los componentes.

En la figura 2.3.1 se muestran las clases raíces definidas en la especificación D&C para describir un componente. Las estructuras de datos de estas clases se definen en la plantilla *DnCComponentDataModel.xsd*, y su contenido puede encontrarse en el Anexo C.

Un paquete de componente (*PackageConfiguration*) describe un componente como elemento distribuible. Cada paquete tiene una única descripción de la interfaz de un componente y una o múltiples implementaciones del mismo.

La citada interfaz del componente describe su funcionalidad desde un punto de vista externo y a su vez viene descrita por medio de la clase *ComponenteInterfaceDescription*. Dos componentes que ofrezcan una misma interfaz, son mutuamente sustituíbles entre sí en cualquier aplicación. La interfaz describe las operaciones, atributos, puertos y parámetros de configuración, que, en conjunto, proporcionan al diseñador la información necesaria para determinar su funcionalidad y la forma de uso.

La implementación de un componente puede se independiente de otros componentes (monolítica, constituida por un conjunto de módulo de código o *artifact*) o puede estar recursivamente implementada como una agrupación plana de instancias de subcomponentes interconectados con una determinada topología, en cuyo caso, la descripción del componente se formula con referencia a la descripción de sus subcomponentes. Las implementaciones monolíticas se describen mediante una clase *MonolithicImplementationDescription* y las implementaciones compuestas se describen mediante una clase *ComponentAssemblyDescription.*

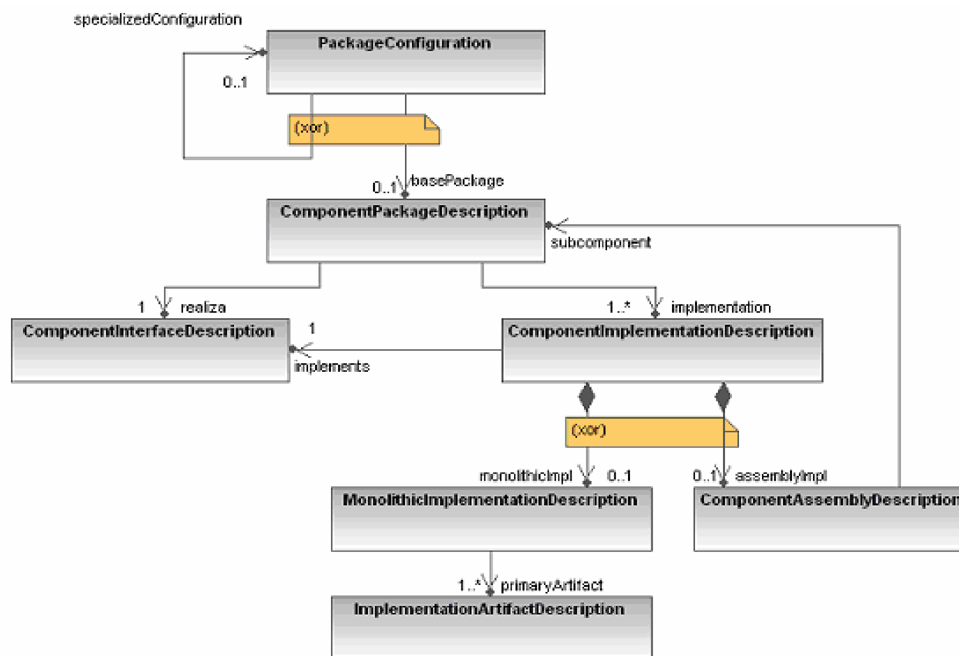

Figura 2.3.1. Clases raíces para la descripción de los componentes

II.3.2-Descripción de la interfaz de un componente.

La descripción del modelo PIM ejecutada por el especificador, se realiza mediante el fichero que describe la interfaz del componente (*SoundGenerator.ccd.xml*) y su contenido y formato son conformes a la especificación D&C de OMG (como hemos comentado, sigue la plantilla *DnCComponentDataModel.xsd)*. A partir de él se tiene acceso a los ficheros que definen la funcionalidad de las interfaces y que también son parte de este modelo (*iPlayer.ice* y *iLogger.ice*).

File: "*SoundGenerator.ccd.xml*"

```
?xml version="1.0" encoding="UTF-8" standalone="no"?>
<DnCcdm:componentInterfaceDescription xmlns:xmi="http://www.omg.org/XMI"
                    xmlns:DnCcdm="http://ctr.unican.es/cbsdnc/DnCComponentDataModel"
                    xmlns:DnCct="http://ctr.unican.es/cbsdnc/DnCCommonTypes"
                    xmlns:rtDnC="http://ctr.unican.es/cbsdnc/rtDnC"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:schemaLocation="http://ctr.unican.es/cbsdnc/DnCComponentDataModel
C:\XML\ICE\DnCComponentDataModel.xsd">
          <description label="Componente que proporciona los recursos para generar sonidos por invocacion
local o remota. Genera cualcuier tipo de sonido, pero tiene preprogramados sonidos con semantica de actividad,
fallo y alarma"
          UUID="http://ctr.unican.es/hola/C_SoundGenerator"
          specificType="components/multimedia/soundGenerator.ccd.xml"
          supportedType="components/multimedia/soundGenerator.ccd.xml">
          <!-- Bussines port descrition -->
                    <port name="playerPort"
                              specificType="interfaces/multimedia/iPlayer.ice::multimedia::iPlayer"
                              supportedType="interfaces/multimedia/iPlayer.ice::multimedia::iPlayer"
                              provider="true"
                              kind="FACET"/>
                    <port name="theLogger"
                              specificType="interfaces/database/iLogger.ice::database::iLogger"
                              supportedType="interfaces/database/iLogger.ice::database::iLogger"
```

```
                                        provider="false"
                                        exclusiveUser="true"
                                        optional="true"
                                        kind="SIMPLEXRECEPTACLE"/>
                        <port name="activationPort"
                                        specificType="PERIODIC_ACTIVATION"
                                        supportedType="PERIODIC_ACTIVATION"
                                        provider="true"
                                        kind="PERIODICACTIVATION"/>
                        <!-- Configuration properties -->
                        <property name="dingSound"   type="STRING" label="sound of an action"/>
                        <property name="chordSound" type="STRING" label="sound of  an action without exit"/>
                        …
                        <!-- General data of the component -->
                        <infoProperty name="author">
                                <value>Laura Barros</value>
                        </infoProperty>
                        <infoProperty name="version">
                                <value>13/02/08</value>
                        </infoProperty>
                        <infoProperty name="repository">
                                <value>http://ctr.unican.es/Java-CCM/homeland</value>
                        </infoProperty>
                </description>
</DnCcdm:componentInterfaceDescription></DnCcdm:componentInterfaceDescription>
```

En el fichero se describe la interfaz del componente *SoundGenerator*. Entre otras cosas expresa que:

- Su identificador (*URI*) es *http://ctr.unican.es/hola/SoundGenerator.*
- Ofrece el puerto (faceta) *playerPort* que implementa la interfaz *iPlayer*, la cual se describe en el módulo *multimedia* del fichero *interfaces/multimedia/iPlayer.ice*
- Requiere el puerto (receptáculo) *theLogger* que corresponde a la interfaz *iLogger*, la cual se describe en el módulo *database* del fichero *interfaces/database/ iLogger.ice*
- Define las seis propiedades de configuración *dingSound*, *chordSound…*
- Asocia a muchos elementos los comentarios (*label)* que definen los elementos.
- Formula un conjunto de propiedades de información como el nombre del autor (*author*), etc.

File: *"iPlayer.ice"*

```
/*********************************************************************
 *                     PROYECTO JAVA-CCM
 *               Grupo Computadores y Tiempo Real (CTR)
 *                    UNIVERSIDAD DE CANTABRIA
 *
 * Description: Interface de un recurso que genera sonidos.
 *
 * @Autor  Laura Barros Bastante
 * @Version 18/04/2007
 *********************************************************************/
module multimedia{
      // Enumeración de los sonidos predefinidos
      enum AlarmSoundType {
            DING,          // Sonido de confirmación de que se ha realizado una acción con éxito.
            CHORD,         // Sonido que representa que no ha tenido éxito la  acción realizada.
            DEFSOUND, // Representa que se ha realizado una acción de forma mas reforzada que DING
            CHIMES,        // Melodía que indica la conclusión con éxito de una actividad relevante.
            FAIL,          // Melodía que representa el fallo de una actividad emprendida.
            ALARM};             // Sonido prolongado y continua que describe una situación de alarma.
      exception FileIsNotFound{string filePath;};       // Se ha fallado en el intento de abrir un fichero
      exception OperationIsNotImplemented{string operationName;};
                              // La operación invocada  no está implementada
       interface iPlayer{
            void playSound(AlarmSoundType sound);
```

```
                              // Genera un sonido que anuncia un cierto estado del sistema.  El parámetro soundType
                              // es de tipo enumerado y representa el sonido.
            void playMelody(string pathMelody) throws FileIsNotFound,OperationIsNotImplemented;
                              // Genera el sonido que se encuentra almacenado en un fichero tipo .wav.
            void silent();  // Silencia el sonido si lo hubiera
      };
};
```

File: "iLogger.ice"

```
/*************************************************************************
 *                    PROYECTO JAVA-CCM
 *             Grupo Computadores y Tiempo Real (CTR)
 *                   UNIVERSIDAD DE CANTABRIA
 *
 * Description: registra persistentemente alarmas, avisos y fallos.
 *
 * @Autor  Laura Barros Bastante
 * @Version 19/04/2007
 *************************************************************************/
module database{
      enum AlarmType{
            ALL,          //Se requieren registros de cualquier tipo ALARM, WARNING o FAIL
            ALARM,        //El registro corresponde a una alarma.
            WARNING, //El registro corresponde a un aviso (Warning)
            FAIL};        //El registro es un fallo o excepción (FAIL).
      struct element{                 //Campos de los elementos de las listas de errores.
            string fecha;             //La fecha/hora en que se ha producido
            AlarmType valor;          // El tipo de mensaje
            string source;            //Origen de la alarma
            string mssg;              //Especifica su naturaleza
            };
      sequence<element> AlarmList;  // Se ha invocado una operación que no está implementada
      exception OperationIsNotImplemented{string operationName;};
       interface iLogger{
            void log( AlarmType type,string source,string mssg) throws OperationIsNotImplemented;
                        // Registra una alarma
            AlarmList get(AlarmType type,short numAlarm) throws OperationIsNotImplemented;
                        //Requiere una lista de alarmas registradas:
            string getEventService(AlarmType type);
                        //Retorna el IOR del canal de transferencia de eventos.
            bool isThereChange(AlarmType type);
                       //Retorna true si se han registrado eventos desde la última vez que se ha leído el Logger.
      };
};
```

II.4-Implementación Java de un componente.

Para cada una de las implementaciones del componente que se ofrezcan, el diseñador debe formular el código de negocio que implementa la funcionalidad que tiene asignada el componente. Éste puede ser legado o desarrollado de forma específica para él.

A fin de que pueda ser procesado por la herramienta de integración, se requiere que el código de negocio satisfaga el siguiente conjunto de características:

1. Debe ofrecer un constructor que al ser invocado genere una instancia local de todos los objetos que constituyen el código de negocio. El modo de invocación del constructor, así como los parámetros que le deben ser pasados, se detallan en la descripción de la implementación (en la descripción *JSSSoundGenerator.ccd.xml*).
2. Debe disponer de una clase (*JSSSoundGenerator.java*) que implemente la interfaz de gestión del componente (*SoundGeneratorMng.java*). El constructor retorna el acceso a la clase principal referenciada como un objeto que implementa esta interfaz.

3. La interfaz de gestión del componente se genera automáticamente a partir de su especificación D&C (*SoundGenerator.ccd.xml*).
4. La implementación de negocio debe ofrecer las facetas que están descritas en su especificación. A cada la faceta que posea el componente se accede por la función (*getIPlayer()*) que a tal fin tiene la interfaz de gestión. Las facetas deben implementar las interfaces que se especifican en el modelo, cuya descripción exacta (*iPlayerOperationsNC.java*) se genera de forma automática a través de la descripción Slice de la interfaz (*iPlayer.ice*). Además, si tiene varias facetas implementando la misma interfaz, deberá crear un objeto por cada una de ellas. Véase la Figura 2.4.1.
5. Lo mismo ocurre con los puertos de activación del componente, por cada uno de ellos, el código de negocio deberá instanciar un objeto que implemente la interfaz correspondiente, y en consecuencia, la funcionalidad asociada a dicho puerto.

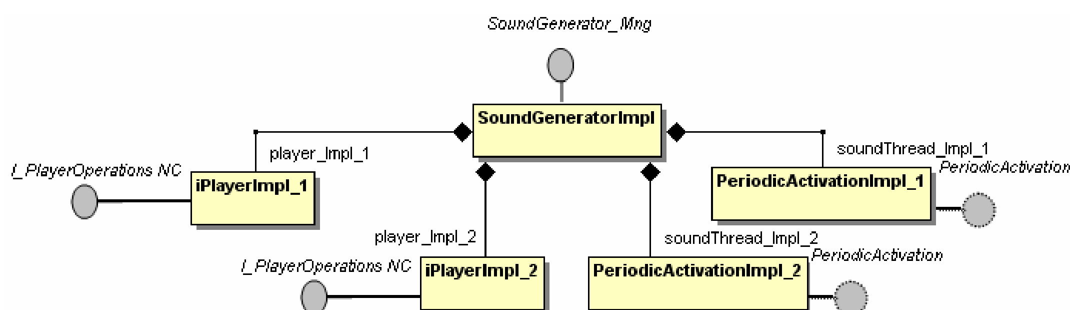

Figura 2.4.1. Estructura del código de negocio de un componente

La generación de una implementación de negocio de un componente se efectúa siguiendo los siguientes pasos:

1) Se procesa la especificación del componente *SoundGenerator.ccd.xml* con la herramienta *BIM_Generator* (*Bussines Interface Management Generator*) para obtener la interfaz de gestión (*SoundGeneratorMng.java*).

File: *SoundGeneratorMng.java*

```
import multimedia.*;
import database.*;
import ccm.*;
public interface SoundGeneratorMngr extends CCMBussinesLifeControl{

    // Navigation operations
    _iPlayerOperationsNC getplayerPortFacet();

    // Connect operations
    void settheLogger(_iLoggerOperationsNC theLogger);

    // Configuration attributes mutator
    void setdingSound(String value);
    void setchordSound(String value);
    void setdefSound(String value);
    void setchimesSound(String value);
    void setalarmSound(String value);
    void setfailSound(String value);

}
```

2) Se procesan las interfaces que se encuentran descritas en lenguaje Slice (*iPlayer.ice*) utilizando el precompilador (*Slice2java*) basado en el proporcionado por ZeroC. El resultado es la interfaz funcional que debe implementar el código de negocio (*_iPlayerOperationsNC.java)*. También se generan otros ficheros auxiliares relativos a los tipos referenciados en la interfaz.

File: *_iPlayerOperationsNC.java*

```
// **********************************************************************
//
// Copyright (c) 2003-2007 ZeroC, Inc. All rights reserved.
//
// This copy of Ice is licensed to you under the terms described in the
// ICE_LICENSE file included in this distribution.
//
// **********************************************************************
// Ice version 3.2.0
package multimedia;

public interface _iPlayerOperationsNC
{
   void playSound(AlarmSoundType sound);
   void playMelody(String pathMelody)
      throws FileIsNotFound,
            OperationIsNotImplemented;
   void silent();
}
```

3) El diseñador debe formular el código de negocio del componente (o adaptarlo si es legado). La única limitación es que ofrezca el constructor, la interfaz de gestión y las interfaces de negocio previamente generadas. Éstas pueden ser generadas en una o en varias clases (esto es último es necesario cuando, por ejemplo, un componente tiene varias facetas que implementan la misma interfaz). En la figura 2.3.2 se muestra el diagrama UML de la implementación y anteriormente, un segmento del código de negocio de la implementación *JSSSoundGenerator.java* del componente *SoundGenerator*.

File: *JSSSoundGenerator.java*

```
/**********************************************************************
 *                    PROYECTO JAVA-CCM
 *              Grupo Computadores y Tiempo Real (CTR)
 *                   UNIVERSIDAD DE CANTABRIA
 *
 * Description: this class contains the private variables and methods
 *
 * @Autor  J.M Drake, Patricia López,Laura Barros
 * @Version 22/12/2007
 **********************************************************************/
import database.*;
import multimedia.*;
import java.io.File;
import javax.sound.sampled.*;
public class JSSSoundGenerator implements SoundGeneratorMng{
        private iPlayerOperationsNC player_impl;
        private _iLoggerOperationsNC theLogger;
        // Configuration attributes
        private String dingSound;
….
        private class PlayerPortFacetNew implements _iPlayerOperationsNC {
        // Class reference
            JSSSoundGenerator owner;
            private PlayerPortFacetNew(JSSSoundGenerator owner) {this.owner = owner;}
            public void playSound(AlarmSoundType sound) { …. }
        }
             public void playMelody(String pathMelody)
                                 throws FileIsNotFound, multimedia.OperationIsNotImplemented {... }
              public void silent() {….. }
```

```
            }
            // For each facet
            public _iPlayerOperationsNC getplayerPortFacet() { return playerPortFacet; }
            //For each receptacle
            public void settheLogger(_iLoggerOperationsNC theLogger){ this.theLogger=theLogger;}
            //For each attribute
            public void setdingSound(String value){dingSound=value;}
….
            // Local variables
….
           // Constructor
            public JSSSoundGenerator(){…..}
            // From CCMBussinesLifeControl interface
           public void activate(){…..}
           public void passivate(){…..}
           public void remove(){…..}
          // private methods …
}
```

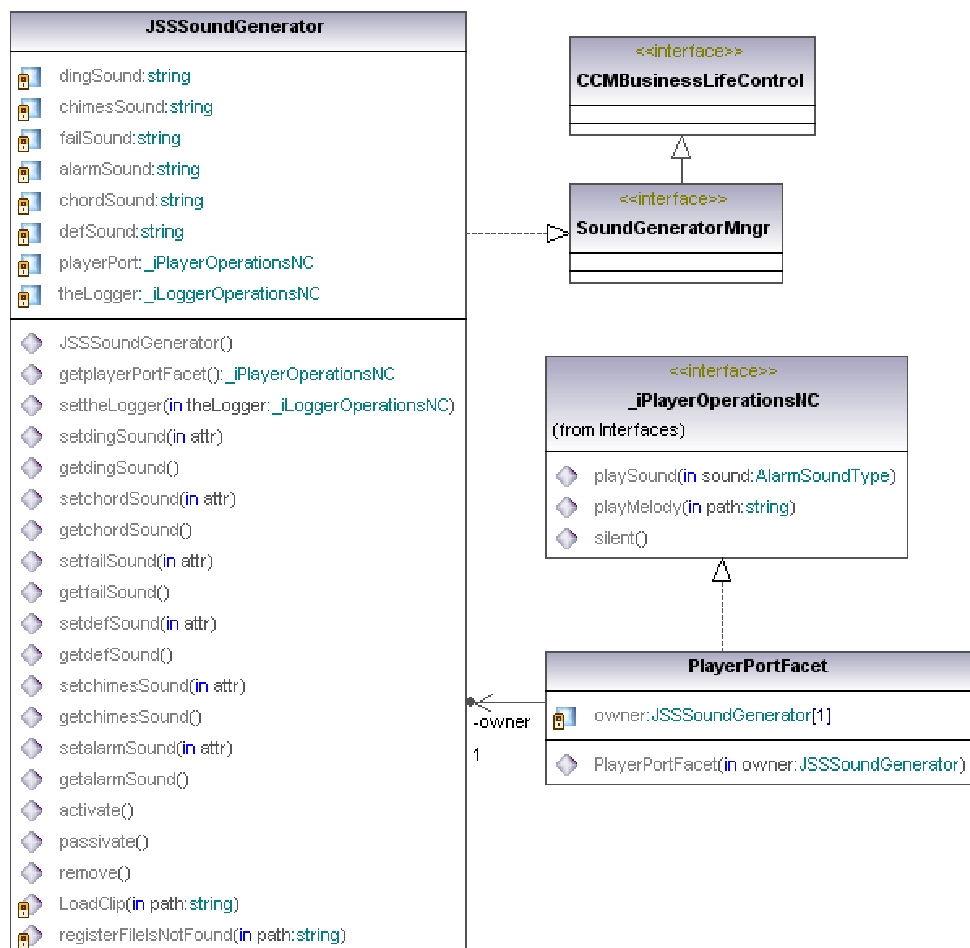

Figura 2.4.2. Descripción del código de negocio de la implementación JSSoundGenerator.

II.5-Descripción D&C de la implementación de un componente.

En esta descripción se describe la implementación del componente *SoundGenerator* dentro del modelo BIM ejecutada por el desarrollador, que se realiza mediante un fichero (*SoundGenerator.cid.xml*) cuyo contenido y formato son conformes a la especificación D&C de OMG (siguiendo la plantilla *DnCComponentDataModel.xsd)*.

Algunas de las características de este componente son:

- Su identificador (*URI*) es http://ctr.unican.es/hola/Iplm_jssSoundGenerator.
- Su interfaz se describe en el fichero externo *components/multimedia/SoundGenerator.ccd.xml*
- La implementación se denomina: *jssSoundGenerator.* Una implementación es una versión del componente que ha sido desarrollada utilizando una estrategia de diseño, un lenguaje y un entorno de programación específicos. Incluye todas aquellas características de comportamiento o de instalación que no quedan incluidas en el código de las implementaciones y que deben ser introducidas por el implementador directamente:
    - Informa que es un componente de negocio reutilizable con diferentes *middlewares*.
    - Es una implementación monolítica cuya especificación no depende de otras implementaciones.
    - Contiene múltiples ficheros con códigos y datos (recursos y capacidades que se requieren del entorno para que pueda ser instanciado el componente):
        - El elemento *jssSoundGenerator* contiene el programa principal. Es un fichero, y su dirección una vez instalado será *components/multimedia/.SoundGenerator/JSSSoundGenerator.java*. Para ser ejecutado requiere que el recurso "run-time_Enviroment" sea del tipo *Java_2_SE*, con versión superior a la "*5.0*"…
        - El elemento *alarm* es un fichero de datos y su dirección una vez instalado en el repositorio será *components/multimedia/.SoundGenerator/alarm.wav*.
        - En otras versiones de la tecnología (como por ejemplo RT-CCM [15], incluye el modelo conducta temporal específico que describe el comportamiento de la implementación del componente.

En la siguiente tabla, se muestra la descripción de la implementación *JSSSoundGenerator* que se describe de forma completa en el Anexo A.

File:SoundGenerator.cid.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<DnCcdm:componentImplementationDescription
xmlns:DnCcdm="http://ctr.unican.es/cbsdnc/DnCComponentDataModel"
xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ctr.unican.es/cbsdnc/DnCComponentDataModel
C:\XML\ICE\DnCComponentDataModel.xsd">
   <!-- Component Package Description -->
   <description label="Ejemplo de implementacion para el componente SoundGenerator"">
      <implements>
         <ref>http://ctr.unican.es/components/multimedia/.SoundGenerator</ref>
      </implements>
      <!--MonolithicImplementationDescription-->
      <monolithicImpl>
         <primaryArtifact name="jssSoundGenerator">
            <referencedArtifact>
               <description location="components/multimedia/.SoundGenerator/JSSSoundGenerator.java">
                  <infoProperty name="ArtifactType">
                     <value>MAINJAVACODE</value>
                  </infoProperty>
                  <deployRequirement resourceType="run-time_Enviroment"
                              name="run-time_Enviroment_Requirement">
                     <property name="type"> <value>Java_2_SE</value> </property>
                     <property name="version"> <value>5.0</value> </property>
                           …
                  </deployRequirement>
```

```
                </description>
            </referencedArtifact>
        </primaryArtifact>
        <primaryArtifact name="alarm">
            <referencedArtifact>
                <description location="components/multimedia/.SoundGenerator/rsrc/alarm.wav">
                    <infoProperty name="ArtifactType"> <value>DATAFILE</value> </infoProperty>
                </description>
            </referencedArtifact>
        </primaryArtifact>
          …
      </monolithicImpl>
   </description>
</DnCcdm:componentImplementationDescription>
```

II.6-Empaquetamiento de un componente Java. Descripción D&C del paquete.

El fichero de descripción del paquete del componente (*JSSSoundGenerator.pcd.xml*) describe toda la información que requiere la herramienta de configuración y despliegue para gestionar la instanciación, configuración, despliegue, enlazado y ejecución del componente en el entorno Java-CCM. En él se describen los ficheros (*artifacts*) que constituyen el componente, los requerimientos que se realizan sobre la plataforma para su despliegue y el procedimiento que se debe seguir para que opere. El contenido y el formato de este fichero sigue el estándar D&C de OMG (plantilla *DnCComponentDataModel.xsd)*. En el fichero que mostramos a continuación, hemos añadido el fichero *ComponentImplementationDescription* dentro de la descripción del paquete del componente (se puede incluir o referenciar).

En esta descripción se describe el paquete en que se distribuye el componente *SoundGenerator*. Algunas de las características de este componente son:

- Su identificador (*URI*) es *http://ctr.unican.es/hola/Iplm_jssSoundGenerator*.
- Su interfaz se describe en el fichero externo *components/soundGenerator/SoundGenerator.ccd.xml*
- Incluye la implementación denominada: *jssSoundGenerator*.

File: SoundGenerator.pcd

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<DnCcdm:packageConfiguration
        xmlns:DnCcdm="http://ctr.unican.es/cbsdnc/DnCComponentDataModel"
        xmlns:xmi="http://www.omg.org/XMI"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://ctr.unican.es/cbsdnc/DnCComponentDataModel
C:\XML\ICE\DnCComponentDataModel.xsd"
        label=""
        UUID="http://ctr.unican.es/hola/Iplm_jssSoundGenerator">
         <!-- Component Package Description -->
        <basePackage>
        <!-- ComponentInterfaceDescription-->
                <realizes>
                        <ref>components/multimedia/SoundGenerator.ccd.xml</ref>
                </realizes>

        <!-- ComponentImplementationDescription-->
        <!-- Implementation -->
        <implementation name="jssSoundGenerator">
                <referencedImplementation>
……. (véase Anexo A)
```

Finalmente, el empaquetador construye el paquete (mediante las *Packaging tools* que integran los ficheros de código en paquetes *.jar*). El código, una descripción formalizada de su funcionalidad, su configurabilidad, los elementos de código de que se compone, los recursos que necesita para su instanciación, la forma en que se ejecuta, etc, se empaqueta y se distribuye como un elemento opaco que va a ser procesado por herramientas automáticas.

III Diseño del contenedor de un componente Java-CCM

III.1- Modelo de referencia CCM. Elementos y funcionalidad.

III.1.1-Adaptación del componente a la plataforma.

La plataforma es el conjunto de procesadores interconectados por una o varias redes de comunicación, en la que se ejecutan las aplicaciones basadas en componentes en concurrencia con otras muchas aplicaciones, a veces todas conocidas (sistema cerrado) y otras veces desconocidas en parte (sistema abierto).

En general las plataformas son distribuidas y heterogéneas. Cada procesador está dotado de su sistema operativo o entorno de ejecución y servicios de comunicaciones adecuados a la naturaleza de la red (o redes) por medio de la que se conecta a otros procesadores.

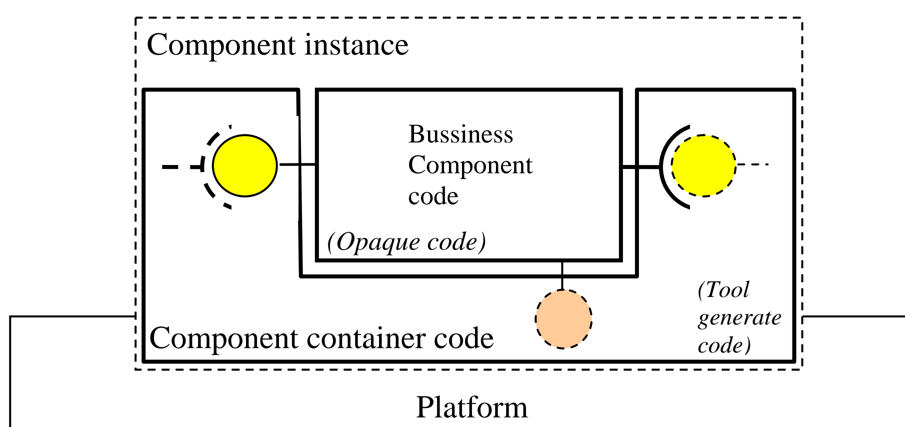

Figura 3.1.1: El contenedor adapta el componente a la plataforma.

La plataforma de ejecución se encuentra descrita mediante un modelo que describe cada procesador y cada red que forma parte de ella. Se describen los recursos que cada elemento dispone, de forma que, cuando en el proceso de despliegue se asigna una instancia de un componente a un nudo, se puede elegir la implementación del componente adecuado al mismo y comprobar si el nudo ofrece los recursos que la implementación del componente requiere.

La adaptación de un componente a las características de los procesadores se realiza utilizando el concepto de contenedor (*container*). El código de negocio de un componente se mantiene inalterado, siendo el código del contenedor el que permite instanciarlo en el procesador y lo dota de los mecanismos de comunicación para comunicarse con otros componentes. Dicho código puede ser elaborado, además, como si el entorno de ejecución fuese siempre monolenguaje y monoprocesador.

Dado que en la descripción de un componente se describen su funcionalidad y los recursos que utiliza de la plataforma, el código del contenedor de un componente se puede generar automáticamente a partir de la descripción, utilizando un procedimiento sistemático susceptible de ser implementado automáticamente por una herramienta.

III.1.2 Estructura interna de un componente

En la figura 3.1.2 se muestran los elementos que forman parte de la estructura interna de un componente.

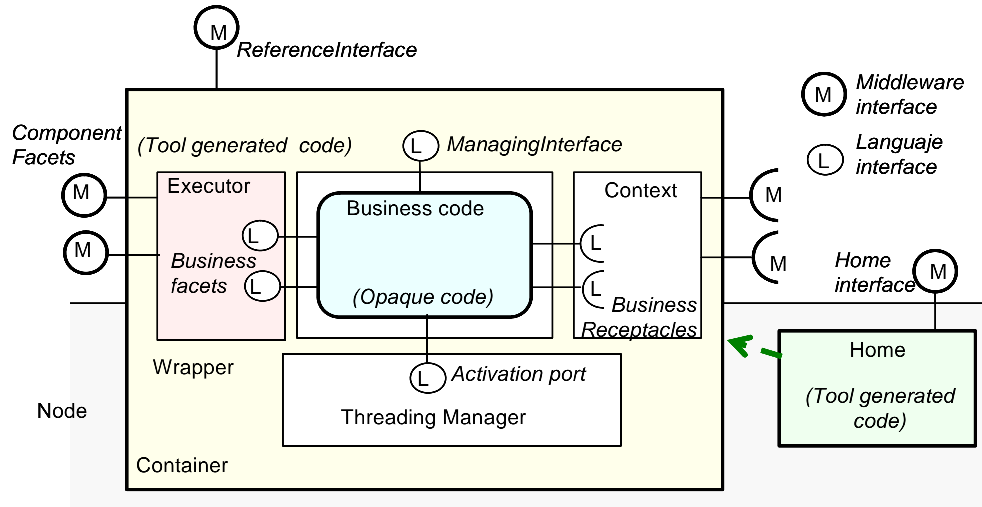

Figura 3.1.2. Estructura de la implementación de un componente

Como hemos comentado anteriormente, la tecnología requiere que el código de negocio del componente ofreca un puerto para gestionar de forma opaca y uniforme su configuración, conectividad y ciclo de vida. Este puerto implementa la interfaz <*nombre componente*>Mng, que es generada de forma automática a partir de la especificación del componente y que no tiene dependencias con la tecnología subyacente. Para la gestión del componente esta interfaz define los siguientes métodos:

- Métodos de navegación, *get<FacetName>*(), que permiten obtener cada una de las implementaciones de las facetas que ofrece el componente.
- Métodos de conexión, *set<ReceptacleName>*(), que permite conectar cada uno de los componentes servidores.
- Métodos de asignación, *set<AttributeName>*(), de los valores de los atributos de configuración del componente.
- Métodos de navegación, *get<ActivationPortName>*(), que permiten obtener las implementaciones de los puertos de activación definidos para el componente.
- Métodos para la gestión del ciclo de vida del componente.

El contenedor engloba todos los recursos que adaptan el código de negocio a la plataforma, siguiendo las reglas de interacción propuestas en LwCCM. Como se muestra en la 3.1.2, está compuesto por varios elementos:

- El contenedor en sí (Wrapper), que ofrece la interfaz equivalente del componente. Dicha interfaz es la que LwCCM establece como la única interfaz a través de la que los clientes o la herramienta de despliegue acceden al componente. Esta interfaz está estandarizada, y extiende a la interfaz CCMObject, que entre otras, ofrece métodos para acceder a las facetas del componente, conectar componentes a sus receptáculos, etc.
- El ejecutor (executor), que implementa los servants o los adaptadores, a través de los que se invocan las operaciones de las facetas del código de negocio. Este elemento es quien aporta todos los mecanismos ICE que se requieren para llevar a cabo la ejecución de las invocaciones.
- El contexto (context) es el mecanismo a través del que el código de negocio puede invocar operaciones en los componentes conectados a través de los receptáculos. Engloba los proxies que se requieren para traducir la invocación local del código de negocio, en una invocación (local o remota) basada en el middleware de base.
- El gestor de threads (threading manager) incluye todos aquellos elementos necesarios para gestionar los puertos de activación que se hayan declarado en el componente. Por cada puerto de activación, deberá crear un thread con sus

correspondientes parámetros (el procedimiento a ejecutar, el parámetro de planificación correspondiente, el periodo en el caso de una activación periódica), y gestionarlo a lo largo de todo su ciclo de vida.

A continuación, describimos los distintos elementos que conforman el contenedor en la tecnología Java-CCM.

III.2 Implementación Java del Contenedor.

III.2.1-Implementación de las Interfaces de la tecnología CCM.

Todos los componentes que siguen la especificación CCM deben ofrecer la interfaz CCMObject (que a su vez hereda de otras interfaces propias de la tecnología). Para tener acceso a una interfaz desde el middleware ICE se necesita disponer de su especificación en lenguaje SLICE. A tal fin, se han *traducido* las interfaces definidas por OMG en el modelo LwCCM (expresadas en lenguaje IDL) al lenguaje SLICE. Dichas interfaces se han recogido en el fichero *ccmcomponens.ice* cuyo código completo puede verse en el Anexo C.

File: ccmcomponent.ice

```
/**********************************************************************
 *                    PROYECTO JAVA-CCM
 *             Grupo Computadores y Tiempo Real (CTR)
 *                 UNIVERSIDAD DE CANTABRIA
 *
 * Description: CCM model
 *
 * @Autor  J.M Drake, Patricia López,Laura Barros
 * @Version 18/12/2007
 **********************************************************************/
module ccm{
/* Navigation Interface */
        …
        //CCM: interface Navigation {
        //              Object provide_facet (in FeatureName name)raises (InvalidName);
        //              FacetDescriptions get_all_facets();
        //              FacetDescriptions get_named_facets (in NameList names)raises (InvalidName);
        //              boolean same_component (in Object object_ref);
        //};
        interface Navigation {
                        Object* provideFacet (string name) throws InvalidName;
                        FacetDescriptions getAllFacets();
                        FacetDescriptions getNamedFacets (NameList names) throws InvalidName;
                        bool sameComponent (Object* objectRef);
        };
        ...
/* Events Interface  */
        …
/*Receptacles Interface */
        …
/*CCM object Interface */
        …
        //CCM: interface CCMObject : Navigation, Receptacles, Events {
        //              CCMHome get_ccm_home( );
        //              …
        //};
        interface CCMObject extends Navigation, Receptacles, Events {
                    Proxy getComponentDef();
                                  …
        };
};
```

Del mismo modo, se definen en SLICE las interfaces para la gestión de los threads desde el contenedor: *OneShotActivation* para una única activación y *PeriodicActivation* para una activación periódica, cuyo código se puede consultar en el Anexo C.

La versión de negocio de la implementación del componente se transforma en una implementación instanciable e invocable en una plataforma con la tecnología Java-CCM a través de tres objetos de las clases ejecutor, contenedor y contexto (*JSSSoundGeneratorExec, JSSSoundGeneratorWrapper y JSSSoundContext*), cuyo código se obtiene mediante un generador automático de código a partir de la descripción del código de negocio. En la figura 3.2.1 se muestra el diagrama de clases en el que se representan las relaciones entre estas clases:

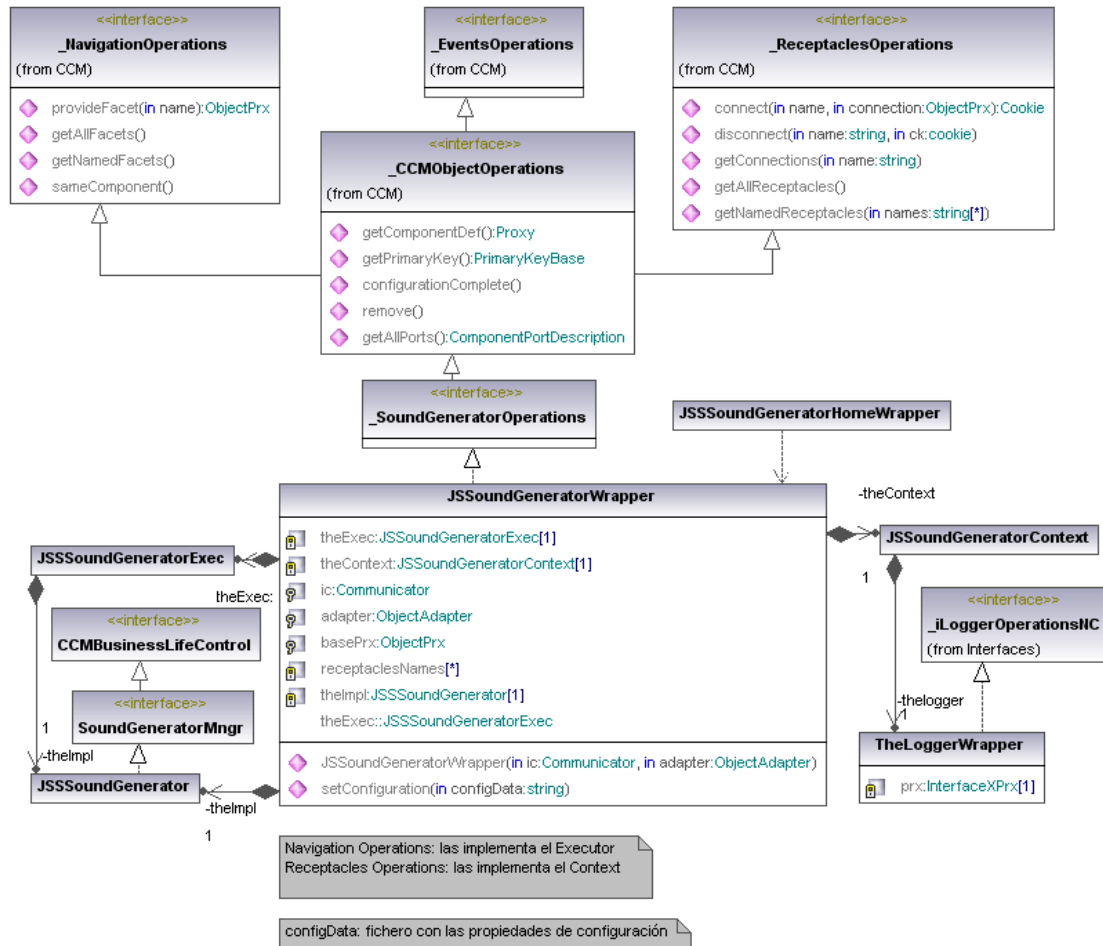

Figura 3.2.1 Diagrama de clases de la implementación Java-CCM de un componente.

Código de negocio del componente (*JSSSoundGenerator.java*): Representa la clase principal del código de negocio, que ha sido desarrollado de acuerdo con las reglas definidas en el apartado previo. El código debe importar las interfaces que implementan sus facetas y que requieren sus receptáculos (los paquetes *multimedia* y *database*). Estos paquetes se obtienen del precompilado de las interfaces en lenguaje *Slice*.

Interfaz de referencia (*__SoundGeneratorOperations***):** interfaz especificada por CCM, a través de la que los clientes o la herramienta de despliegue acceden al componente ya que extiende a la interfaz *CCMObjectOperations*, que hereda de otras interfaces definidas en la tecnología: una que ofrece métodos para acceder a las facetas del componente

(*_NavigationOperations*),otra que ofrece métodos para la gestión de eventos y por último (*_ReceptacleOperations*) que sirve conectar componentes a sus receptáculos.

III.2.2-Implementación Java del Wrapper.

Contenedor (JSSSoundGeneratorWrapper.java): Representa el contenedor que es creado sobre el nudo de ejecución, a fin de ejecutar el código de negocio y los mecanismos de comunicación proporcionados por el *middleware*. Ofrece también la interfaz (*_SoundGeneratorOperations*) a través de las que las herramientas de configuración y despliegue, u otros componentes, pueden gestionar el ciclo de vida del componente, su configuración y conexión/desconexión con otros componentes. Esta interfaz está estandarizada y corresponde a una extensión directa de la interfaz *CCM_Object* definida en el paquete de la tecnología *ccm*, y por tanto es independiente de la funcionalidad del componente.

El diagrama de clases de la figura 3.2, muestra los elementos que componen la clase Wrapper:

- El ejecutor contiene la instancia del Contexto, el Ejecutor y la Implementación (*theContext, theExec,theImpl*).
    - Delega los métodos de navegación de facetas (*provideFacet(), getAllFacets()*,etc) en los del mismo nombre, ofrecidos por la clase Ejecutor.
    - Delega los métodos de conexión y navegación de receptáculos (*connect(), getAllReceptacles()*,etc) en los del mismo nombre, ofrecidos por la clase Context.
    - Ofrece un método para activar la implementación y establecer la conexión del componente a través de sus receptáculos: *configurationComplete()*.
    - Ofrece un método para obtener todos los puertos funcionales (facetas, receptáculos) del componente: *getAllPorts()*.
    - Dispone de un método para borrar el acceso a la instancia del componente: *remove()*.
    - Dispone de un método para establecer los valores de los atributos de configuración de la instancia del componente: *setConfiguration()*.

File: ***JSSSoundGeneratorWrapper.java***

```
/**************************************************************************
 *                    PROYECTO JAVA-CCM
 *             Grupo Computadores y Tiempo Real (CTR)
 *                  UNIVERSIDAD DE CANTABRIA
 *
 * Description: wrapper class of JJSoundGenerator
 *
 * @Autor  J.M Drake, Patricia López,Laura Barros
 * @Version 5/03/2008
 **************************************************************************/
import multimedia.*;
import ccm.*;
public class JSSSoundGeneratorWrapper implements _SoundGeneratorOperations{
        /*Variables*/
        JSSSoundGeneratorExec theExec;
        JSSSoundGeneratorContext theContext;
        JSSSoundGenerator theImpl;
        Ice.Communicator ic = null;
        Ice.ObjectAdapter adapter;
```

```
		Ice.ObjectPrx basePrx;
		String objectName;
		/*Constructor*/
		public JSSSoundGeneratorWrapper() {
			theContext = new JSSSoundGeneratorContext();
			theImpl = new JSSSoundGenerator();
			theExec = new JSSSoundGeneratorExec(theImpl);
			theExec.setContext(theContext);}
	//This method are executed by the executor
		public Ice.ObjectPrx provideFacet(String name, Ice.Current __current)
				throws InvalidName {return theExec.provideFacet(name);		}
		//This method are executed by the executor
		public FacetDescription[] getAllFacets(Ice.Current __current) {
			return theExec.getAllFacets();}
		//This method are executed by the executor
		public FacetDescription[] getNamedFacets(String[] names,
				Ice.Current __current) throws InvalidName {
			return theExec.getNamedFacets(names);}
		//This method are executed by the executor
		public boolean sameComponent(Ice.ObjectPrx objectRef, Ice.Current __current) {
			return (basePrx.ice_id()==objectRef.ice_id());} //A este nivel
		//This method are executed by the context
		public Cookie connect(String name, Ice.ObjectPrx connection,
				Ice.Current __current) throws AlreadyConnected,
				ExceededConnectionLimit, InvalidConnection, InvalidName {
			return theContext.connect(name, connection);}
		//This method are executed by the context
		public void disconnect(String name, Cookie ck,
				Ice.Current __current) throws CookieRequired, InvalidConnection,
				InvalidName, NoConnection {…}
		//This method are executed by the context
		public ConnectionDescription[] getConnections(String name,
				Ice.Current __current) throws InvalidName {
			return theContext.getConnections(name);}
//This method are executed by the context
		public ReceptacleDescription[] getAllReceptacles(Ice.Current __current) {
			return theContext.getAllReceptacles();
		}
		//This method are executed by the context
		public ReceptacleDescription[] getNamedReceptacles(String[] names,
				Ice.Current __current) throws InvalidName {
			return theContext.getNamedReceptacles(names);
		}
		/*Method for completing the configuration*/
		public void configurationComplete(Ice.Current __current)
				throws InvalidConfiguration {…}
		/*Method for getting all ports (facets and receptacles)*/
		public ComponentPortDescription getAllPorts(Ice.Current __current) {…}
		public Proxy getComponentDef(Ice.Current __current) {return null;}
		/*Method for getting the PrimaryKey*/
		public PrimaryKeyBase getPrimaryKey(Ice.Current __current)
				throws NoKeyAvailable {return null;}
		/*Method for removing the implementation*/
		public void remove(Ice.Current __current) throws RemoveFailure {…}
		/*Method for setting the configuration. Is called by the home*/
		public void setConfiguration(String arg){
			String[] args={arg};
			ic = Ice.Util.initialize(args);
			Ice.Properties properties = ic.getProperties();
			objectName=properties.getProperty("SoundGenerator.objectName");
			String protocolData=properties.getProperty("SoundGenerator.protocolData");
			//Config Values
			theImpl.setdingSound(properties.getProperty("SoundGenerator.dingSound"));
			…
			adapter = ic.createObjectAdapterWithEndpoints(objectName+"_Adapter", protocolData);
			_SoundGeneratorTie servant = new _SoundGeneratorTie(this);
```

```
                basePrx = adapter.add(servant, ic.stringToIdentity(objectName));
                theExec.createFacetExecutors(adapter,basePrx);
                adapter.activate();
                new Thread(new Runnable(){
                        public void run(){
                        …
                        ic.waitForShutdown();
                        …).start();
                }
}
```

III.3-Implementación Java de Executor.

Ejecutor (*JSSSoundGeneratorExec.java*): Representa los mecanismos por los que otros componentes invocan las operaciones ofrecidas por las facetas. Básicamente está compuesto por los *adapter* o *servants,* que se instancian para proporcionar localmente los *threads* que ejecutan las operaciones invocadas.
El diagrama de clases de la figura 3.4, muestra los elementos que componen la clase Ejecutor:

- El ejecutor contiene una referencia a la implementación (*theImpl)* para invocar las operaciones ofrecidas por el código de negocio a través de sus facetas.
- Contiene un mapa (*theFacets*) formado por pares de valores <Nombre, ObjetoPrx> donde el nombre es el nombre de cada una de las facetas y ObjetoPrx el proxie a las mismas.
- Ofrece métodos para:
    - Creación de las facetas*: createFacetExecutors().*
    - Obtener la interfaz implementada por la faceta:*getPlayerPortFacet().*
    - Obtener una faceta determinada por su nombre: *providefacet().*
    - Obtener todas las facetas del componente: *getAllFacets().*
    - Obtener un conjunto de facetas especificadas por sus nombres: *getNamedFacets().*

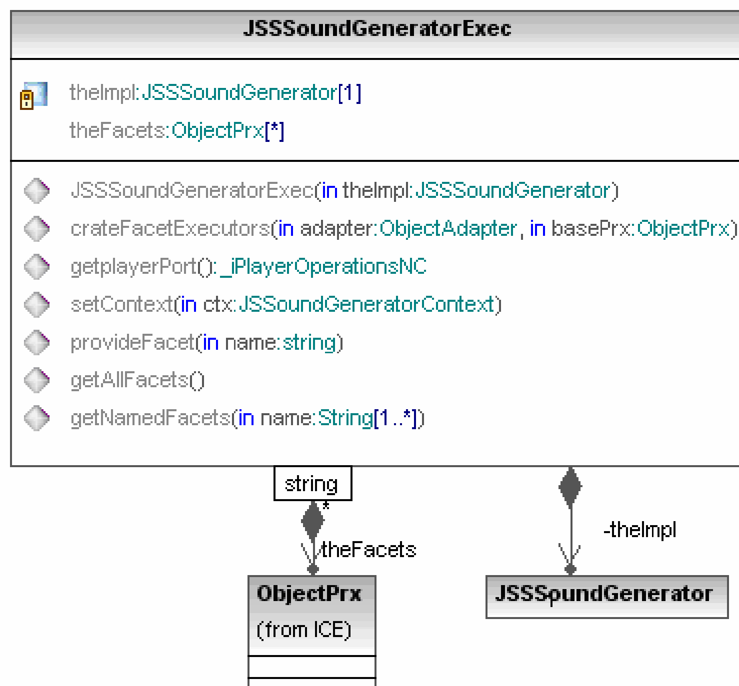

Figura 3.3. Diagrama de clases del Ejecutor del Componente SoundGenerator

File: ***JSSSoundGeneratorExec.java***

```
/*************************************************************************
 *                  PROYECTO JAVA-CCM
 *            Grupo Computadores y Tiempo Real (CTR)
 *                UNIVERSIDAD DE CANTABRIA
 *
 * Description: executor class of JJSoundGenerator
 *
 * @Autor  J.M Drake, Patricia López,Laura Barros
 * @Version 5/03/2008
 ***********************************************************************/
import java.util.*;
import Ice.*;
import ccm.*;
import multimedia.*;
public class JSSSoundGeneratorExec {
        /*Variables*/
        JSSSoundGenerator theImpl;
        …
        /*Constructor*/
        public JSSSoundGeneratorExec(JSSSoundGenerator impl) {…}
        /*Method for creating the facets*/
        public void createFacetExecutors(ObjectAdapter adapter, Ice.ObjectPrx basePrx) {…}
        /*Méthod for getting the facet PlayerPort*/
         public _iPlayerOperations getPlayerPortFacet() {
                return new _iPlayerOperations() {
                        public void playSound(AlarmSoundType sound, Ice.Current __current) {
                                theImpl.getplayerPortFacet().playSound(sound);
                        }
                        public void playMelody(String pathMelody, Ice.Current __current)
                                        throws FileIsNotFound, OperationIsNotImplemented {
                                theImpl.getplayerPortFacet().playMelody(pathMelody);
                        }
                        public void silent(Ice.Current __current) {
                                theImpl.getplayerPortFacet().silent();
                        }
                };
        }
        /*Method by mean the component has access to information about their receptacles*/
        public void setContext(JSSSoundGeneratorContext theContext) {this.theContext = theContext;}
        /*Method for obtaining a facet witch name is specified by the argument name*/
        public ObjectPrx provideFacet(String name) throws InvalidName {…}
        /*Method for obtaining all facets*/
        public FacetDescription[] getAllFacets() {…}
        /*Method for obtaining  the facet witch names are specified by the argument names */
        public FacetDescription[] getNamedFacets(String[] names) throws InvalidName {…}
        //activate of bussines code is executed from wrapper
        public void activate() {}
        //remove of bussines code is executed from wrapper
        public void remove() {}
        //passivate of bussines code is executed from wrapper
        public void passivate() {}
}
```

III.4-Implementación Java de Contexto.

Contexto (*JSSSoundGeneratorContext.java*): Representa los mecanismos a través de los que el código de negocio invoca las operaciones de las facetas que se han enlazado con los receptáculos del componente. Básicamente está compuesto por los correspondientes *proxies* o *stubs* que traducen la invocación local que realiza el código de negocio en una invocación (local o remota) basada en el middleware.

El diagrama de clases de la figura 3.4, muestra los elementos que componen la clase Contexto:

- Contiene un conjunto (*receptaclesNames*) formado por los nombres de los receptáculos.
- Contiene una clase que implementa la interfaz de las operaciones del componente (Logger) que es usada por el código de negocio para que el componente SoundGenerator puede acceder por su receptáculo (*iLoggerPort*) al componente que esté conectado a él.
- Ofrece métodos para:
    - Conectar el receptáculo del cliente (SoundGenerator) con la faceta del servidor (Logger): *connect()*.
    - Desconectar la conexión que el componente tiene a través de su receptáculo: *disconnect()*.
    - Obtener la interfaces de cada uno de los receptáculos:*getTheLogger()*.
    - Obtener las conexiones que el componente ha establecido a través de su receptáculo: *getConnections()*.
    - Para obtener los receptáculos del componente: *getAllReceptacles()*.
    - Obtener un conjunto de receptáculos especificados por sus nombres: *getNamedReceptacles*().

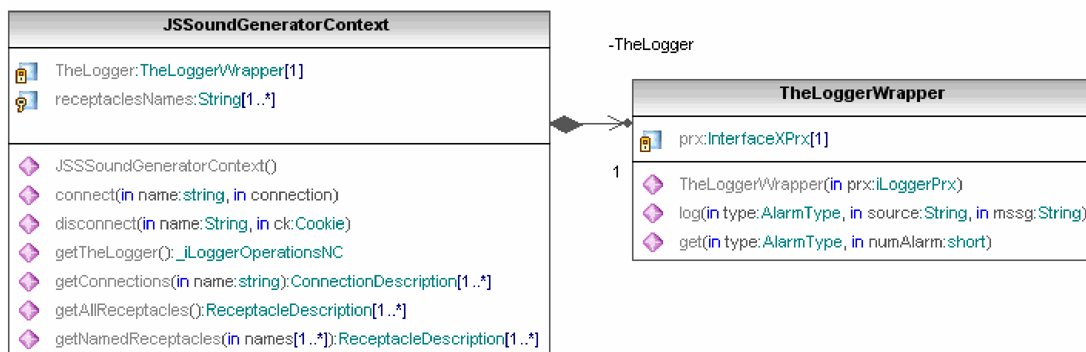

Figura 3.4. Diagrama de clases del Contexto del componente SoundGenerator

File: ***JSSoundGeneratorContext.java***

```
/*********************************************************************
 *                    PROYECTO JAVA-CCM
 *            Grupo Computadores y Tiempo Real (CTR)
 *                 UNIVERSIDAD DE CANTABRIA
 *
 * Description: context class of JJSSoundGenerator
 *
 * @Autor  J.M Drake, Patricia López,Laura Barros
 * @Version 5/03/2008
 *********************************************************************/
import java.util.*;
import ccm.*;
import database.*;
public class JSSSoundGeneratorContext {
        /*Variables*/
        ....
        /*Constructor*/
        public JSSSoundGeneratorContext() { …}
        /*Method for connecting the reference specified by the connection param with the receptacle specified
        by the param name*/
        public Cookie connect(String name, Ice.ObjectPrx connection){…}
        /*Method for disconnecting the component of the receptacle*/
```

```
        public void disconnect(String name, Cookie ck) throws CookieRequired,
                        InvalidConnection, InvalidName, NoConnection {…}
        /*Méthod for getting the receptacle*/
        public _iLoggerOperationsNC getTheLogger() {return theLogger;}
        /*Method for getting the connections of the component has by means of the receptacle specified by the
        name param*/
        public ConnectionDescription[] getConnections(String name)throws InvalidName {..}
        /*Method for getting all the receptacles of the component*/
        public ReceptacleDescription[] getAllReceptacles() {…}
        /*Method for getting the receptacles of the component specified by the names param*/
        public ReceptacleDescription[] getNamedReceptacles(String[] names)
                        throws InvalidName {…}
        /*Class that offers the interface of the operations of the receptacle*/
        private class TheLogger_x implements _iLoggerOperationsNC {
                iLoggerPrx prx;
                public TheLogger_x(iLoggerPrx prx) {this.prx = prx;}
                public void log(AlarmType type, String source, String mssg)
                                throws OperationIsNotImplemented { prx.log(type, source, mssg);}
                public element[] get(AlarmType type, short numAlarm)
                                throws OperationIsNotImplemented {  return prx.get(type, numAlarm);}
                public String getEventService(AlarmType type) {return prx.getEventService(type);}
                public boolean isThereChange(AlarmType type) {return prx.isThereChange(type);}
                public iLoggerPrx getPrx() {return prx;}
        }
}
```

III.5-Implementación Java del gestor de threads.

Para gestionar los puertos de activación que se hayan declarado en el componente se deberá crear un thread con sus correspondientes parámetros (el procedimiento a ejecutar, el parámetro de planificación correspondiente, el periodo en el caso de una activación periódica), y gestionarlo a lo largo de todo su ciclo de vida.

Para mostrar la implementación del gestor de threads, lo haremos a través del componente MediaSource, que permite la captura, transmisión y almacenamiento de una señal de vídeo. Si deseamos que el almacenamiento de la imagen se realice con un periodo determinado, deberemos definir un puerto de activación de tipo periódico en la descripción D&C de la interfaz del componente.

En el fichero que contiene la interfaz de gestión del componente (*MediaMonitor.ccd.xml*) declaramos un puerto de activación de tipo periódico. Como consecuencia, la interfaz *MediaMonitorMng.java*, que es generada de forma automática a partir de la especificación del componente, ofrecerá un método para obtener dicho puerto.

```
/* Method for obtaining the periodicActivation paPort0*/
public PeriodicActivation getpaPort0 ();
```

En la implementación del código de negocio, deberemos realizar la implementación de los puertos de activación:

```
private PeriodicActivation paPort0;
public PeriodicActivation getpaPort0() {
        return paPort0;
}
private class OaPort0 implements PeriodicActivation{
        public void update() {
                //Operaciones del código de negocio para salvar la imagen
                }
}
```

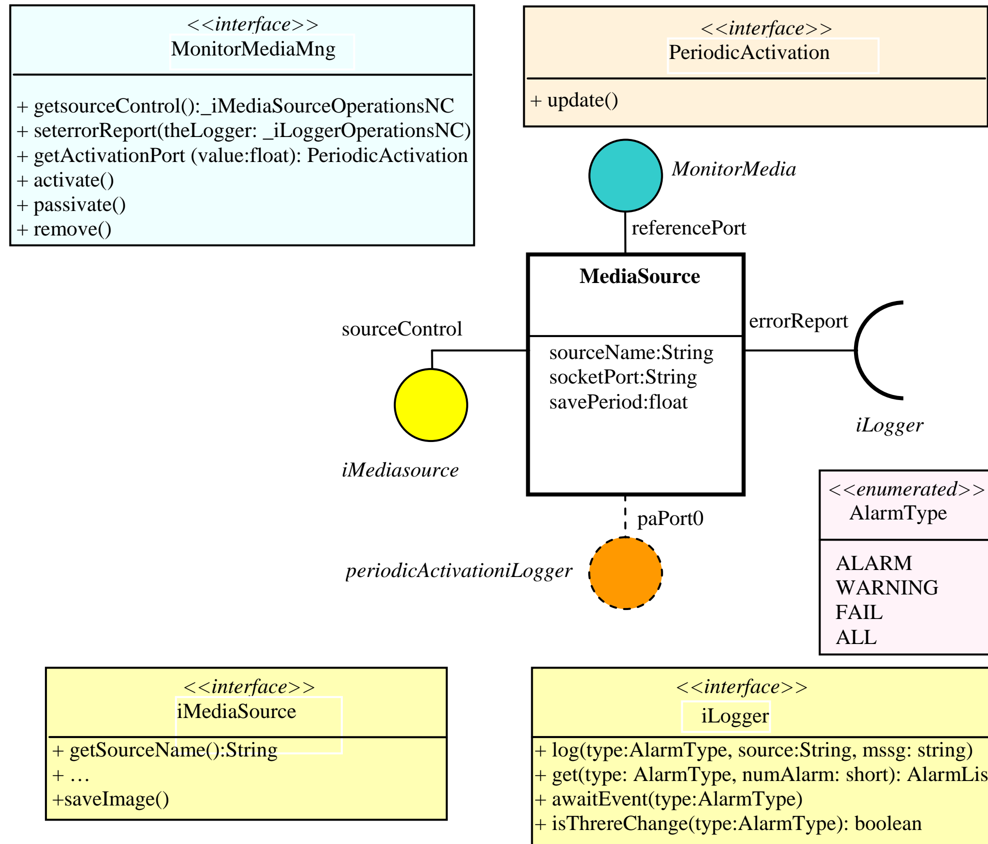

Figura 3.5. Elementos que definen la funcionalidad de negocio y de gestión de MediaSource.

En la clase Wrapper definimos un objeto del tipo *ScheduledExecutorService*, clase de java que nos permitirá crear una cola de Threads (en este caso cada puerto gestiona un thread, luego la longitud será 1), configurando el periodo de la tarea (que leeremos del plan de despliegue) y la planificación de su activación.

```
ScheduledExecutorService paPort0Scheduler;
long periodPaPort0;

        /*Constructor*/
        public GCImplementation0Wrapper() {
            …
          paPort0Scheduler=Executors.newScheduledThreadPool(1);
        }
        /*Method for completing the configuration*/
        public void configurationComplete(Ice.Current __current)throws InvalidConfiguration {
            …
          PeriodicShotThread paPort0=new PeriodicShotThread(theImpl.getpaPort0());
          paPort0Scheduler.scheduleAtFixedRate(paPort0,(long)0,periodPaPort0,MILLISECONDS);
        }
        /*Method for setting the configuration. Is called by the home*/
        public void setConfiguration(String arg){
          String[] args={arg};
          ic = Ice.Util.initialize(args);
          Ice.Properties properties = ic.getProperties();
          objectName=properties.getProperty("GenericComponent.objectName");
          …
         periodPaPort0=Long.getLong(properties.getProperty("paPort0.period"))*1000;//milliseconds
        …
        }
```

```
//Class that implements Runnable and contains an PeriodicActivation reference
private class PeriodicShotThread implements Runnable{
        PeriodicActivation myPeriodicActivation;
        public PeriodicShotThread (PeriodicActivation pa){
                this.myPeriodicActivation=pa;
        }
        public void run(){
                myPeriodicActivation.update();
        }
}
```

III.6 Implementación Java del Home.

Gestor del componente (JSSSoundGeneratorHomeWraper.java): Representa el mecanismo a través del que se pueden instanciar y configurar las implementaciones de los componentes en un determinado procesador de la plataforma. Ofrece un método estático a través del cual se puede crear el *Home* en un nudo. A partir de él se pueden establecer los parámetros de configuración, invocando la operación *setConfigurationValues()* y luego invocando la función *create()* se pueden crear un número arbitrario de instancias del componente con los valores de configuración previamente establecidos.

La clase Home deberá implementar la interfaz *_SoundGeneratorHomeOperations* que a su vez extiende de las interfaces definidas en la tecnología que definen los métodos de gestión del *home* del componente.

El diagrama de clases de la figura 3.6.1, muestra los elementos que componen la clase Home:

- Mecanismos para almacenar los valores de configuración del componente(*theConfigValues*).
- Ofrece métodos para:
    - La creación de una o varias instancias del componente con los valores de configuración establecidos en el plan de despliegue: *create() y createComponent().*
    - Completar la configuración del componente y desactivar el componente: *completeComponentConfiguration() y disableHomeConfiguration().*
    - Borrar un componente y evitar futuros accesos al mismo: *removeComponent().*
- Un método *main()* que crea el *servant* del *home* a través del cual, la instancia del componente, espera peticiones del entorno.

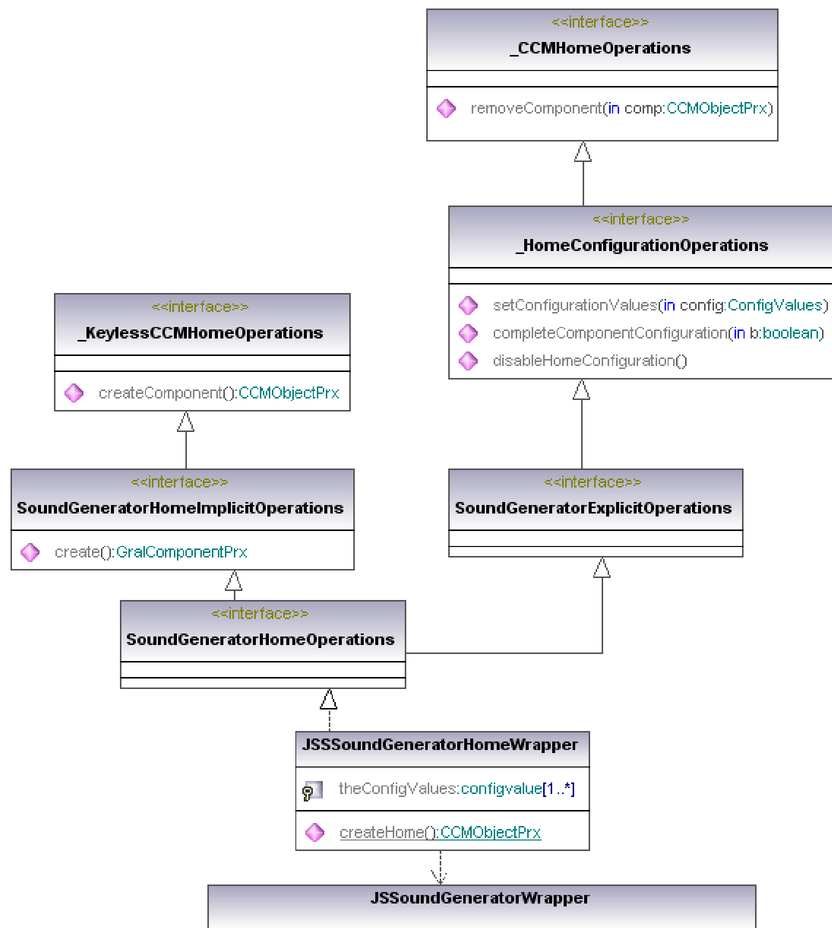

Figura 3.6.1. Diagrama de clases del Home

File: *"SoundGeneratorHomeWrapper.java"*

```
/*********************************************************************
 *                    PROYECTO JAVA-CCM
 *              Grupo Computadores y Tiempo Real (CTR)
 *                   UNIVERSIDAD DE CANTABRIA
 *
 * Description: Equivalent interface of the home.
 *
 * @Autor  J.M Drake, Patricia López,Laura Barros
 * @Version 5/03/2008
 *********************************************************************/
import Ice.*;
import multimedia.*;
import ccm.*;

public class JSSoundGeneratorHomeWrapper implements _SoundGeneratorHomeOperations{

        /*Variables*/
        Ice.ObjectAdapter adapter;
        static Ice.Communicator ic;
        private ConfigValue[] theConfigValues;

        /*Method for creating a new component*/
        public SoundGeneratorPrx create(Ice.Current __current) throws ccm.CreateFailure {
                …
        }
```

```
        /*Method for setting a configuration for home object*/
        public void setConfigurationValues(ConfigValue[] config, Ice.Current __current) {
                …
        }
        /*Method for determining if the configuration will be called*/
        public void completeComponentConfiguration(boolean b, Ice.Current __current) { }
        /*Method for disabling the configuration of the home*/
        public void disableHomeConfiguration(Ice.Current __current) {
                …
        }
        /*Method for creating a new component*/
         public CCMObjectPrx createComponent(Ice.Current __current) throws CreateFailure {
                …
        }
         /*Method for removing the component*/
        public void removeComponent(CCMObjectPrx comp, Current __current) throws RemoveFailure {
                …
        }
        /*Main method for creating a home servant*/
        public static void main(String[] args) {
                JSSSoundGeneratorHomeWrapper theHome =  new JSSSoundGeneratorHomeWrapper();
                ic = Ice.Util.initialize(args);
                Ice.ObjectAdapter adapter =
ic.createObjectAdapterWithEndpoints("JSSSoundGeneratorHomeWrapper","default -p 5001");
                _SoundGeneratorHomeTie servant = new _SoundGeneratorHomeTie(theHome);
                adapter.add(servant, ic.stringToIdentity("JSSSoundGeneratorHomeWrapper"));
                adapter.activate();
                System.out.println("JSSSoundGeneratorHomeWrapper is running");
                ic.waitForShutdown();
        }

}
```

En la figura 3.6.2 se muestra el proceso de generación de los ficheros de código que constituyen el componente ejecutable en una plataforma Java-CCM. Las descripciones Slice de las interfaces de los puertos de negocio del componente y de sus las interfaces de gestión así como de su constructor (*Home*) se procesan utilizando el pre-compilador Slice2Java de la empresa ZeroC. Como resultado se genera un conjunto de paquetes java que son requeridos por las clases principales para implementar sus capacidades de comunicación con otros elementos:

- **ccm**: contiene los ficheros resultantes de la compilación del fichero *ccmcomponentes.ice* que contiene las interfaces SLICE que define la tecnología CCM.
- **database**: contiene los ficheros resultantes de la compilación del fichero *iLogger.ice.*
- **multimedia:** contiene los ficheros resultantes de la compilación del fichero *iPlayer.ice* y *SoundGenerator.ice.*

Los ficheros D&C que describen el componente, junto con los ficheros que definen el contendor y su constructor son integrados con el código de negocio que ha sido diseñado con independencia de la tecnología, para dar lugar al componente instanciable, configurable y ejecutable en la plataforma Java-CCM.

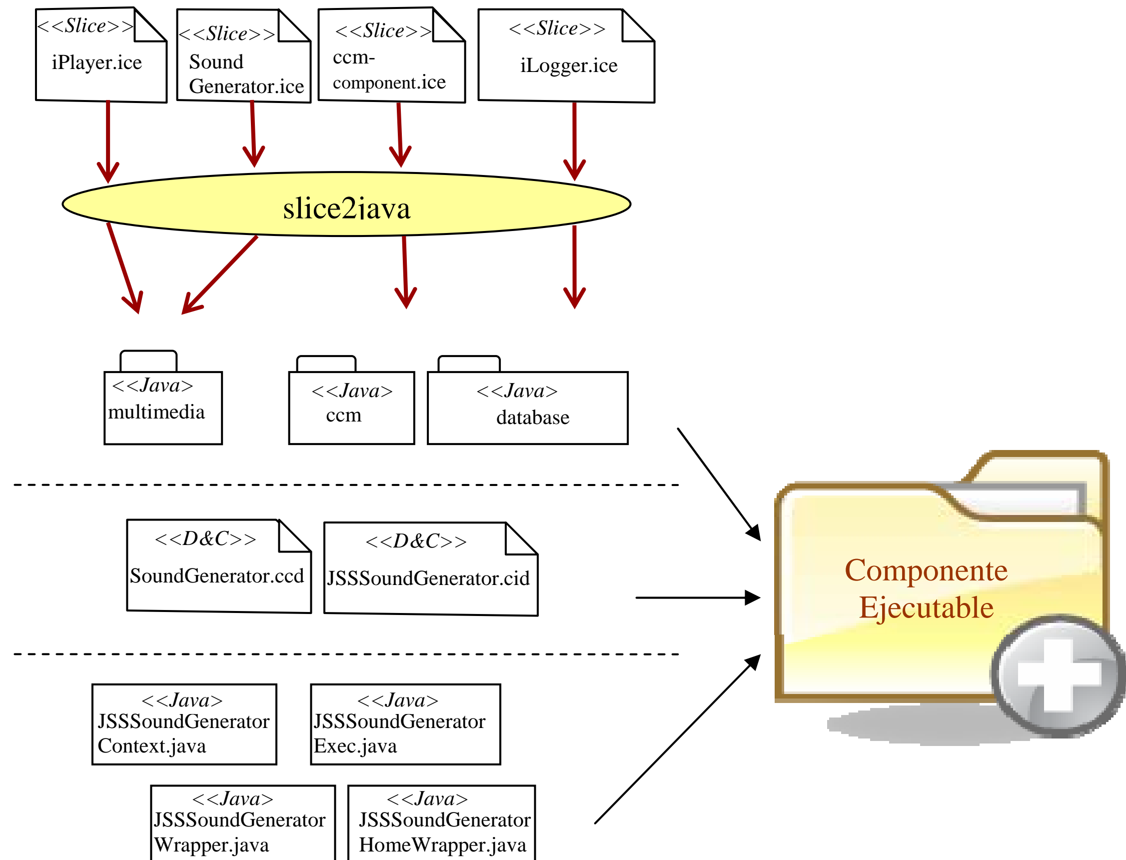

Figura 3.6.2 Ficheros del componente ejecutable Java-CCM.

III.7-Plan de despliegue y herramienta de despliegue local.

III.7.1-Plan de despliegue local.

El plan de despliegue local que sigue la especificación D&C, especifica las instancias (*OperatorBell*, *logger*) de los componentes que forman la aplicación, asociándolas al nudo (local) en el que se ejecutan. Además se indican las propiedades de configuración de dichas instancias: unas están declaradas en la interfaz del componente (p.e. *dingSound*) y otras suelen ser de configuración de aspectos relativos a la plataforma que están definidas por la tecnología CCM (p.e. *PROTOCOLDATA*). El plan de despliegue para una aplicación que contiene una instancia del componente *SoundGenerator* y otra del componente *Logger* (*SoundGenerator&LoggerProbe.ccd.xml,* compuesto por las descripciones de las interfaces de ambos componentes, *SoundGenerator.ccd.xml* y *Logger.ccd.xml)*, se muestra en la tabla siguiente y a cuyo contenido completo, se puede acceder en el Anexo A.

La descripción del modelo de ejecución la explicaremos de forma detallada en el capítulo IV.3.

File: LocalDeployment.cpd.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<DnCedm:deploymentPlan xmlns:DnCedm="http://ctr.unican.es/cbsdnc/DnCExecutionDataModel"
        xmlns:DnCbt="http://ctr.unican.es/cbsdnc/DnCBasicTypes"
        xmlns:DnCcdm="http://ctr.unican.es/cbsdnc/DnCComponentDataModel"
        xmlns:DnCct="http://ctr.unican.es/cbsdnc/DnCCommonTypes"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        xsi:schemaLocation="http://ctr.unican.es/cbsdnc/DnCExecutionDataModel

:\XML\ICE\DnCExecutionDataModel.xsd">
        <realizes>
                <ref>components/alarm/SoundGenerator&LoggerProbe.ccd.xml</ref>
        </realizes>
        <target UUID="http://ctr.unican.es/Java-CCM/hola/LocalProbe/"</target>
<instance name="player1" node="OperatorComputer"
           source="components/soundGenerator/SoundGenerator.ccd.xml::jssSoundGenerator">
           <configProperty name="OBJECTNAME">
<value><string>OperatorBell</string></value></configProperty>
           <configProperty name="PROTOCOLDATA"><value> <string>default -p 6000</string></value>
</configProperty>
           <configProperty
name="dingSound"><value><string>/rsrc/ding.wav</string></value></configProperty>
          …
           <deployedResource requirementName="run-time_Enviroment_Requirement"
              resourceName="theJavaVirtualMachine"
              resourceUsage="NONE">
           </deployedResource>
        </instance>
        <instance name="logger1" node="FieldComputer"
            source="components/logger/Logger.ccd.xml::MySQLJavaLogger">…</instance>
        <connection name="logger1Toplayer1">
                <internalEndpoint portName="iLogger" instance="logger1"/>
                <internalEndpoint portName="iPlayer" instance="player1"/>
        </connection>
</DnCedm:deploymentPlan>
```

III.7.2-Herramienta de despliegue local.

Los valores de configuración de los componentes, son definidos por CCM como valores de tipo “Any”. Al no existir dicho tipo en SLICE se tomó la decisión de utilizar las propiedades de configuración de ICE. Esto es, en un fichero de texto, el usuario puede definir cualquier propiedad que requiera su aplicación, simplemente estableciendo un par de elementos constituido por un nombre y un valor:

Ejemplo: MiComponente.nombre=SoundGenerator

Las únicas restricciones impuestas por “Ice” son el uso del carácter “#” y la palabra reservada “Ice”.

La transformación del plan de despliegue a un fichero de propiedades Ice, se puede realizar parseando el contenido del fichero .xml y generando los pares de elementos en un fichero temporal .txt.

En nuestra herramienta de despliegue local (LocalDeploymentToolJava-CCM.java), este fichero (config.Components), constituye un parámetro de ejecución para el mismo. Por ejemplo, para ejecutarlo se ejecutaría la siguiente instrucción:

java LocalDeploymentToolJava-CCM --Ice.Config=C:\Java-CCM...\config.Components

Toda la información necesaria para generar este fichero, se encuentra en el plan de despliegue, ya que existe una completa correspondencia.

File: config.Components

```
SoundGererator.objectName=OperatorBell
SoundGererator.protocolData=default -p 5000

SoundGererator.dingSound=/rsrc/ding.wav
SoundGererator.chordSound=/rsrc/chord.wav
SoundGererator.defSound= /rsrc/tada.wav
SoundGererator.chimesSound=/rsrc/chimes.wav
SoundGererator.failSound=/rsrc/fail.wav
SoundGererator.alarmSound= /rsrc/alarm.wav


Logger.objectName=Logger
Logger.protocolData=default -p 6000

Logger.baseName=MySQLlogger
Logger.login = root
Logger.password = unican.es
Logger.url = jdbc:mysql://127.0.0.1:3306/mysql
Logger.m_table = test
```

Los pasos que realiza la herramienta son:

- **Lanzamiento de los *home* de los componentes que se van a interconectar :** Cada llamada implica la creación del home, que se comportará como un “server” que espera peticiones: añadir a un adapter creado ad-hoc, el servant del home, que servirá para acceder al mismo, remotamente.
- **Obtener los proxies de los home:** Obtenemos los proxies de los home y comprobamos que son del tipo esperado (*KeyLessCCMHomePrx*).
- **Configuración del home:** en este paso, vemos si los proxies que obtuvimos del home, son del tipo que esperamos (*HomeConfigurationPrx*) y establecemos los valores de configuración.
- **Instanciación de componentes:** Creamos los componentes. Esto implica hacer una llamada a *createComponent()* sobre el home, que nos devolverá un proxie del componente, es decir, del tipo *CCMObjectPrx*.
- **Conexión de componentes:** Para realizar la conexión de los componentes, requerimos el nombre de la faceta del componente con el que deseamos establecer la conexión. Como es el ejecutor el que contiene el mapa de las facetas, *provideFacet()* sobre el proxie, redirecciona la llamada al ejecutor.
- **Interconecta los componentes:** Para interconectar los componentes, o lo que es lo mismo, conectar la referencia de objeto especificada por el parámetro *connection* al receptáculo especificado por el parámetro *name* del componente, aplicamos el método *connect()* sobre el proxie. El método *connect()* del contexto se encarga de obtener el proxie del tipo esperado (*iLoggerPrx*) y crea un objeto que implementa la interfaz *_iLoggerOperationsNC* que ofrece las operaciones de negocio del componente al que nos queremos conectar.
- **Activación:** Completamos la configuración con la llamada al método *configurationComplete()* sobre cada uno de los proxies. Este método se encuentra en el wrapper y consta de una llamada a una operación (*settheLogger()*), generada automáticamente para la obtención por parte del componente del acceso a los componentes conectados a sus receptáculos) y de la activación de la implementación mediante el método *activate()* sobre la misma.

File: LocalDeploymentToolJava-CCM.java

```
/*******************************************************************
 *                  PROYECTO JAVA-CCM
 *           Grupo Computadores y Tiempo Real (CTR)
 *                UNIVERSIDAD DE CANTABRIA
 *
 * Description: Test application
 *
 * @Autor  J.M Drake, Patricia López,Laura Barros
 * @Version 5/03/2008
 *******************************************************************/
import java.io.*;
import Ice.*;
import ccm.*;

//Imports for probe
import multimedia.AlarmSoundType;
import multimedia.iPlayerPrx;
import multimedia.iPlayerPrxHelper;

public class LocalDeploymentToolJava {
    public static void main(String[] args) {
            final String homeSoundName = "JSSSoundGeneratorHomeWrapper";
           //el nombre de la implementación se extrae del .pcd
            final String homeLoggerName = "JMySQLHomeWrapper";
           //el nombre de la implementación se extrae del .pcd
          final String loggerPort="theLogger";//se extrae del .ccd
          final String soundPort="playerPort";//se extrae del .ccd
            java.lang.Process homeSound=null;
            java.lang.Process homeLogger=null;
            …
            homeSound=Runtime.getRuntime().exec("java …"+homeSoundName);
            …
             homeLogger=Runtime.getRuntime().exec("java -…"+homeLoggerName);
            …
            Communicator ic = Ice.Util.initialize();
            Ice.ObjectPrx basePlayer = ic
                      .stringToProxy("JSSSoundGeneratorHomeWrapper:default -p 5001");
            CCMHomePrx thePlayerHomePrx=CCMHomePrxHelper.checkedCast(basePlayer);
            Ice.ObjectPrx baseLogger = ic
                      .stringToProxy("JMySQLHomeWrapper:default -p 5002");
            CCMHomePrx theLoggerHomePrx=CMHomePrxHelper.checkedCast(baseLogger);
            KeylessCCMHomePrx theLoggerHome = KeylessCCMHomePrxHelper
                      .checkedCast(theLoggerHomePrx);
            KeylessCCMHomePrx thePlayerHome = KeylessCCMHomePrxHelper
                      .checkedCast(thePlayerHomePrx);

            ConfigValue loggerConfValue1 = new ConfigValue(args[0]);
            ConfigValue[] loggerConfigValues = { loggerConfValue1 };

            ConfigValue playerConfValue1 = new ConfigValue(args[0]);
            ConfigValue[] playerConfigValues = { playerConfValue1 };

            HomeConfigurationPrx theLoggerHomeConf = HomeConfigurationPrxHelper
                      .checkedCast(theLoggerHomePrx);
            theLoggerHomeConf.setConfigurationValues(loggerConfigValues);
            HomeConfigurationPrx thePlayerHomeConf = HomeConfigurationPrxHelper
                      .checkedCast(thePlayerHomePrx);
            thePlayerHomeConf.setConfigurationValues(playerConfigValues);
            CCMObjectPrx theLoggerPrx = null;
            …
            theLoggerPrx = theLoggerHome.createComponent();
            …
            CCMObjectPrx thePlayerPrx = null;
            …
            thePlayerPrx = thePlayerHome.createComponent();
```

```
            …
            Ice.ObjectPrx theLoggerFacet = null;
           …
            theLoggerFacet = theLoggerPrx.provideFacet(loggerPort);
            …
            thePlayerPrx.connect(loggerPort, theLoggerFacet);
            …
            …
            thePlayerPrx.configurationComplete();
            theLoggerPrx.configurationComplete();
            …
      }
}
```

IV Ejemplo de aplicación Homeland Alarm

IV.1-Demostrador Homeland: Especificación y arquitectura.

La aplicación HomelandAlarm (HOLA) tiene como objeto verificar si el estado de un conjunto de elementos que se supervisan, pasa a ser uno de lo definidos como estados de alarma. Cuando se pasa de un estado de no alarma a un estado de alarma el sistema ejecuta el conjunto de acciones de respuesta que se encuentra especificado para esa transición.
La aplicación es de carácter general y puede utilizarse para implementar cualquier sistema de supervisión que pueda ser formulado en base a verificaciones de estados de objetos y control de respuestas cuando se detectan transiciones a los estados definidos como alarmas.
La configuración de la aplicación para controlar un sistema concreto se realiza a través de una estructura de datos que debe ser cargada por la aplicación antes de comenzar a operar. En la figura 4.1.1 se muestra con un diagrama de clases UML la estructura de datos que corresponden a una configuración:

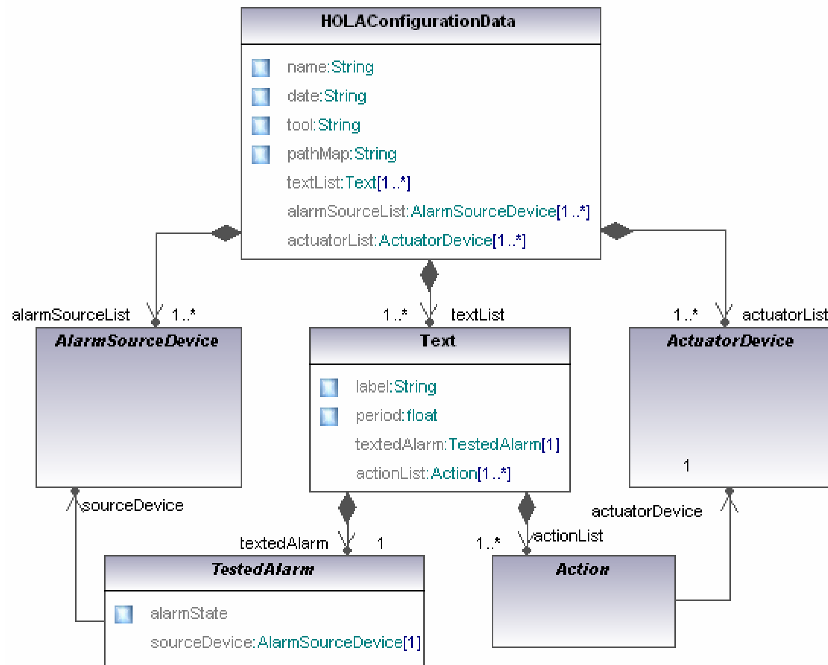

Figura 4.1.1.Arquitectura del demostrador Homeland

- Un conjunto de elementos que son fuente de alarma, para los cuales se puede programar que su estado sea periódicamente supervisado. En la versión actual del demostrado existen tres tipos de elementos fuente de alarma:
  - Línea digital de entrada: Se verifica si su estado lógico coincide con el estado establecido como estado de alarma.
  - Señal analógica de entrada: Se verifica si el valor instantáneo de la tensión analógica es inferior a un umbral alto, o si es superior a un umbral bajo, o si se encuentra dentro o fuera del rango definido por los umbrales alto y bajo.
  - La hora actual: Se verifica si el instante actual está dentro o fuera de un intervalo horario establecido.
- Un conjunto de elementos **actuadores** que tienen capacidad de ejecutar las acciones de respuesta que serán ordenadas por el sistema al detectarse una alarma. Los actuadores que se definen en esta versión inicial, son:
  - Logger: permite el registro de un mensaje que contiene un tipo de alarma, la fuente en que se ha detectado y el instante de tiempo en que se ha producido.

    - Generador de sonido: Permite generar un sonido temporal de aviso o una señal acústica persistente de alarma. La naturaleza del sonido se define en la acción que la invoca.
    - Señal digital de salida: puede ser establecida a un estado determinado (se define en la acción que la invoca) cuando se detecta un estado de alarma (setting). De forma alternativa, se puede programar como acción el que la línea se mantenga en un determinado estado mientras que persista el estado de alarma (holding).
    - Monitor de video: como respuesta a una alarma puede establecerse que una señal de video procedente de una determinada fuente se visualice en el monitor o cese de ser visualizada.
    - Registro de imágenes: permite establecer como respuesta a una alarma el que se almacene en un registro de imágenes la imagen que en ese instante está siendo adquirida por una fuente de vídeo. La imagen se almacena con un nombre generado automáticamente por la aplicación y que hace referencia a la fuente de vídeo de la que procede y al instante en que se ha tomado.
- Un **conjunto de verificaciones** que el sistema debe llevar a cabo. Cada verificación se define mediante tres elementos:
    - La fuente de alarma que supervisa, así como el estado que se define como estado de alarma.
    - El periodo con que debe realizarse la supervisión.
    - La lista de acciones de respuesta que debe ejecutarse si en el componente se detecta estado de alarma.

La estructura de datos de configuración se codifica utilizando XML, y su contenido y formato se define utilizando una plantilla de W3C-*Schema* que se denomina "*HomelandAlarmConf.xsd*".

En diagrama de clases UML de la figura 4.1.2 se muestra la estructura general de la aplicación, los componentes que se admiten, las interfaces que cada componente ofrece y las interconexiones que se permiten.

Tiene una estructura de tres capas compuesta por:

1. Capa cliente: AlarmGUI => Constituye el componente que lanza y repliega a la aplicación y ofrece la única interfaz de interacción con el operador.
2. Capa de intermediación: AlarmManager => es el componente activo que implementa la funcionalidad básica de la aplicación. Configurada por el cliente, gestiona de forma autónoma todos los elementos que implementan las fuentes de alarma y los dispositivos actuadores.
3. Capa terminal: Esta constituida por un número ilimitado de componentes que, o bien ofrecen que su estado sea supervisado, y por tanto pueden considerarse como fuentes de alarma, o bien ofrecen operaciones para actuar sobre el entorno, y por tanto constituyen elementos actuadores.

El componente Logger conlleva un doble papel. Por un lado, es un elemento actuador más que ofrece la acción de registrar un mensaje en él cuando se detecta una alarma. Pero por otro lado, es el mecanismo que utiliza el cliente para supervisar el estado de la aplicación y así estar informado de los eventos que se producen. A tal fin, todos los componentes tienen acceso a él, ya que pueden registrar cualquier error que en ellos se produzca con un mensaje de tipo FAIL. El cliente lo puede supervisar de dos modos:

1. A demanda del operador, el cliente puede obtener del logger una lista de los últimos eventos que se han producido en el sistema.

2. El cliente puede constituirse en escuchante de los eventos que se generan en el logger, y tras cada uno de ellos requerir y mostrar de forma automática los eventos y mensajes registrados.

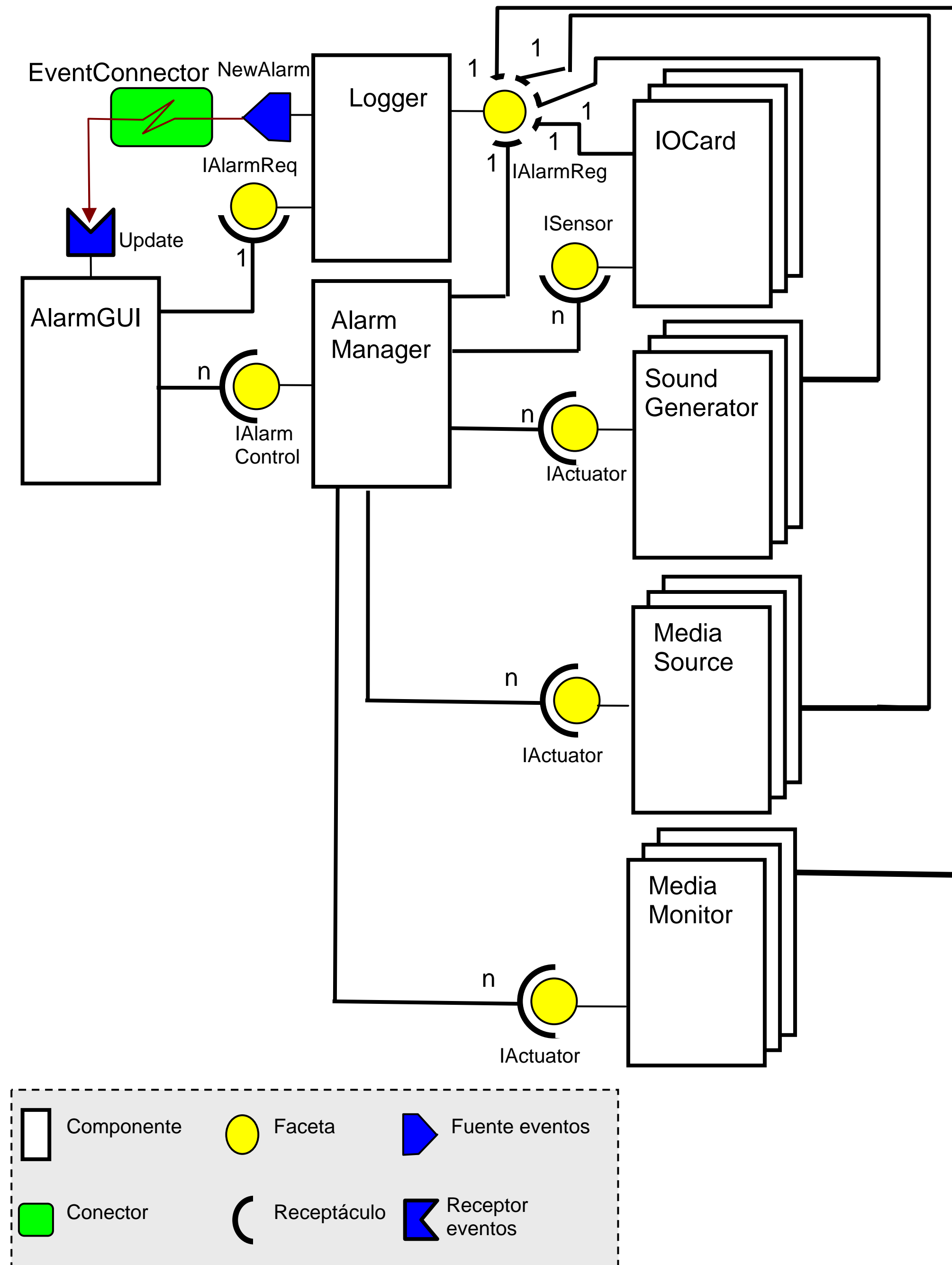

Figura 4.1.2. Arquitectura software de la aplicación Homeland Alarm

IV.2-Catálogo de componentes.

En el demostrador HomelandAlarm se utilizan los siguientes componentes

- Componente **AlarmGUI**: Representa el componente cliente que lanza la aplicación e implementa la interfaz de interacción con el operador. Es un componente ligero y autocontenido que permite con muy poco esfuerzo modificar el aspecto externo de la aplicación a fin de adaptarlo al sistema específico al que se aplica. Se ha desarrollado

una implementación en Java utilizando los recursos que proporciona el IDE Eclipse (*JAlarmGUI*).

- Componente **AlarmManager**: Representa el componente que constituye el nivel intermedio de la aplicación e integra en él toda la funcionalidad del dominio HomelandAlarm. Es un módulo complejo diseñado para permanecer inalterado durante gran parte de la vida de la aplicación. Dado que la tercera capa de fuentes de alarma y actuadores puede ser extendida con nuevos elementos futuros, se ha desarrollado implementando un patrón de diseño *Factory*, esto es, se ha diseñado en dos secciones. El núcleo independiente de esos cambios y una permanente y otra muy modular que puede ser fácilmente extendida para que admita nuevos componentes. Se ha desarrollado una implementación en Java (*JAlarmManager*).
    - Facetas (interfaces que ofrece): iAlarm.
    - Receptáculos (interfaces que requiere): iLogger, iIo, iPlayer, iMediaSource e iMediaMonitor.
- Componente **PCI9111IOCard**: Permite la gestión de la tarjeta PCI9111-DG de la empresa ADLink Company Inc. Esta tarjeta permite manejar hasta 24 líneas digitales de entrada, hasta 20 líneas digitales de salida, hasta 16 líneas analógicas de entrada y sólo una línea analógica de salida.
    - Facetas (interfaces que ofrece): iDigitalIO e iAnalogIO.
    - Receptáculos (interfaces que requiere): iLogger.

  Se dispone de dos implementaciones distintas de este componente:
    - *PCI9111DASKIOCard*: ofrece la funcionalidad completa de la tarjeta a través de los drivers DASK XP constituidos por librerías C y sólo disponibles para la plataforma Windows XP.
    - *PCI9111ComediIOCard*: utiliza un *driver* de la familia Comedi de drivers abiertos para Linux. No ofrece la funcionalidad completa, ya que no gestiona el puerto extendido de la tarjeta y, en consecuencia, sólo permite el control de las l6 primeras líneas digitales de entrada y de las 16 primeras líneas digitales de salida.
- Componente **MediaSource**: Permite controlar la captura de la señal de una cámara de vídeo y su transmisión a través de la red utilizando un protocolo RTP/RTSP y la captura de la imagen que corresponde al marco de la señal de vídeo que se produce en un instante dado, así como su codificación en formato JPEG y su almacenamiento como un fichero. La implementación realizada de este componente (*JMFMediaSource*), utiliza la tecnología JMF (Java Media Framework) de Sun Microsystems, que puede instalarse en la plataforma Java 2 SE 5.0. y que ofrece una funcionalidad muy completa de este dominio. Así mismo, se utilizan los *drivers* propios de Windows que tratan la cámara de video como un dispositivo media estándar.
    - Facetas (interfaces que ofrece): iMediaSource.
    - Receptáculos (interfaces que requiere): iLogger.
- Componente **MediaMonitor**: Permite la monitorización de la señal de video en vivo procedente de una determinada fuente, ya sea local o transferida a través de la red por un protocolo RTP/RTSP. Se utiliza para su implementación (*JMFMediaMonitor*) la tecnología JMF y se utiliza como elemento reproductor el que proporcionan las librerías AWT de Java.
    - Facetas (interfaces que ofrece): iMediaMonitor.
    - Receptáculos (interfaces que requiere): iLogger.
- Componente **SoundGenerator**: Es un sencillo generador de sonidos implementado utilizando los recursos que ofrece la propia plataforma Java 2 SE 5.0 a través de las clases incluidas en el paquete javax.Sound.Sampled. Sirve para interpretar segmentos de sonidos sobre la tarjeta de sonidos del propio PC.
    - Facetas (interfaces que ofrece): iPlayer.
    - Receptáculos (interfaces que requiere): iLogger.

- Componente **Logger**: Implementa un logger soportado mediante una base de datos comercial libre MySQL. El logger admite el registro de mensajes con tres características: "naturaleza", que puede tomar uno de entre los siguientes tres valores: FAIL, WARNING, ALARM; "fuente", que hace referencia al objeto desde el que se registra, y "mensaje", que se especifica de forma concreta en cada mensaje. Bajo demanda, el componente puede retornar la lista de los últimos eventos registrados que se le solicite, y así mismo, generar un evento cada vez que se registra un nuevo mensaje, al cual puede suscribirse como escuchante cualquier objeto que lo desee utilizar.
  - Facetas (interfaces que ofrece): iLogger.

  Se dispone de dos implementaciones distintas de este componente:
  - *MySQLJavaLogger*: Este componente se ha implementando utilizando el conector "JDBC" para acceso desde Java a una base de datos MySQL instalada sobre Windows XP.
  - *MySQLLinuxCLogger*: Implementa un logger también soportado por la base de datos MySQL y tiene una funcionalidad semejante a la del componente anterior. Se ha implementado utilizando conectarla librería "MySQL.h" para acceso desde programas en lenguaje C a la base de datos MySQL instalada sobre Linux.

IV.3-Modelo de despliegue de la plataforma. Descripción D&C.

En la figura 4.3.1 se muestran las clases raíces que se utilizan en la especificación D&C para describir una plataforma. Las estructuras de datos que corresponden a estas clases se definen en la plantilla *DnCPlatformDataModel.xsd*, cuyo contenido puede encontrarse en el Anexo C.

El modelo de datos de la plataforma contiene la información relativa a una plataforma para desplegar sobre ella cualquier aplicación. Está compuesto por un conjunto de nudos, interconectados por redes de comunicación, pudiendo haber puentes entre las redes.

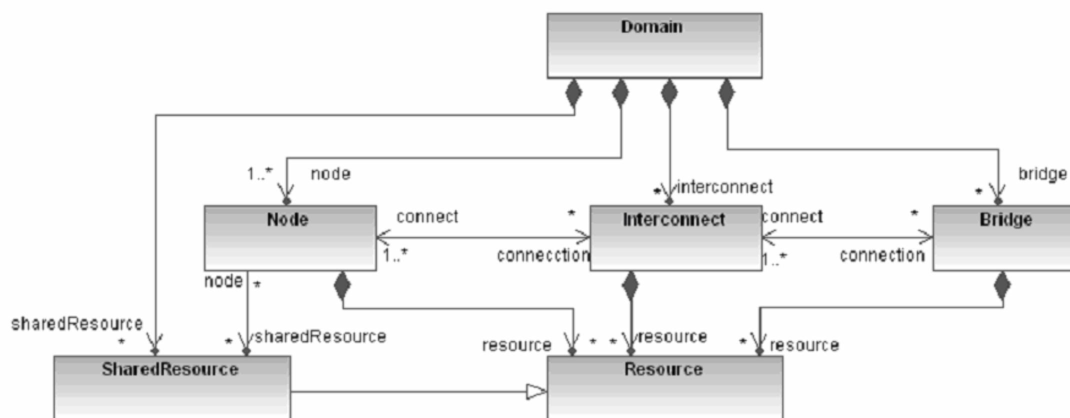

Figura 4.3.1. Clases raíces para la descripción de una plataforma.

- **Domain:** Representa el contenedor de más alto nivel que contiene toda la información disponible sobre la plataforma de ejecución. Un dominio está compuesto de uno o más nudos procesadores, redes de comunicación y puentes entre ellos. Así mismo, los recursos compartidos por uno o más nudos, también se consideran agregados directamente al dominio.

- **Node**: Los nodos tienen capacidad de procesamiento y en ellos se ejecutan los componentes. Estos nodos están interconectados mediante canales de comunicación, a través de los cuales intercambian información y acceden a servicios. Además, tienen agregados recursos, los cuales cualifican y cuantifican sus capacidades.
- **Interconnect**: Describen los canales de comunicación entre nudos. Pueden tener asignados recursos, los cuales cualifican y cuantifican las capacidades de los canales.
- **Bridge:** Representan los *switches* o *routers* que interconectan los canales de comunicación y permiten complejas capacidades de comunicación entre nudos. Los puentes pueden tener asignados recursos, los cuales cualifican y cuantifican sus capacidades.
- **Resource**: Los recursos pueden estar asociados a los nudos, a los canales de comunicación o a los puentes, y representan elementos que están agregados a ellos y que proporcionan las capacidades que pueden ser requeridas por los componentes para poder instalarse en ellos.
- **SharedResource**: Representan recursos que por su naturaleza son compartidos por diferentes nudos de la plataforma. Al no estar directamente agregados a ningún elemento, se consideran agregados al dominio.

La plataforma en que se despliega y ejecuta la aplicación Homeland Alarm, es distribuida y tiene procesadores repartidos por todo el área de trabajo del sistema. Los elementos de entorno que se verifican, el logger, los elementos que generan las alarmas sonoras y los elementos sobre los que se establecen las líneas de alarma, pueden estar localizados en cualquier nudo del sistema en el que existan recursos hardware de soporte necesarios. Se dispone de Plataforma Java sobre Windows XP, plataforma Linux Ubuntu 6.0, plataforma Java sobre Linux Ubuntu 6.0. Los componentes se han desarrollado utilizando los lenguajes de programación Java y C/C++.

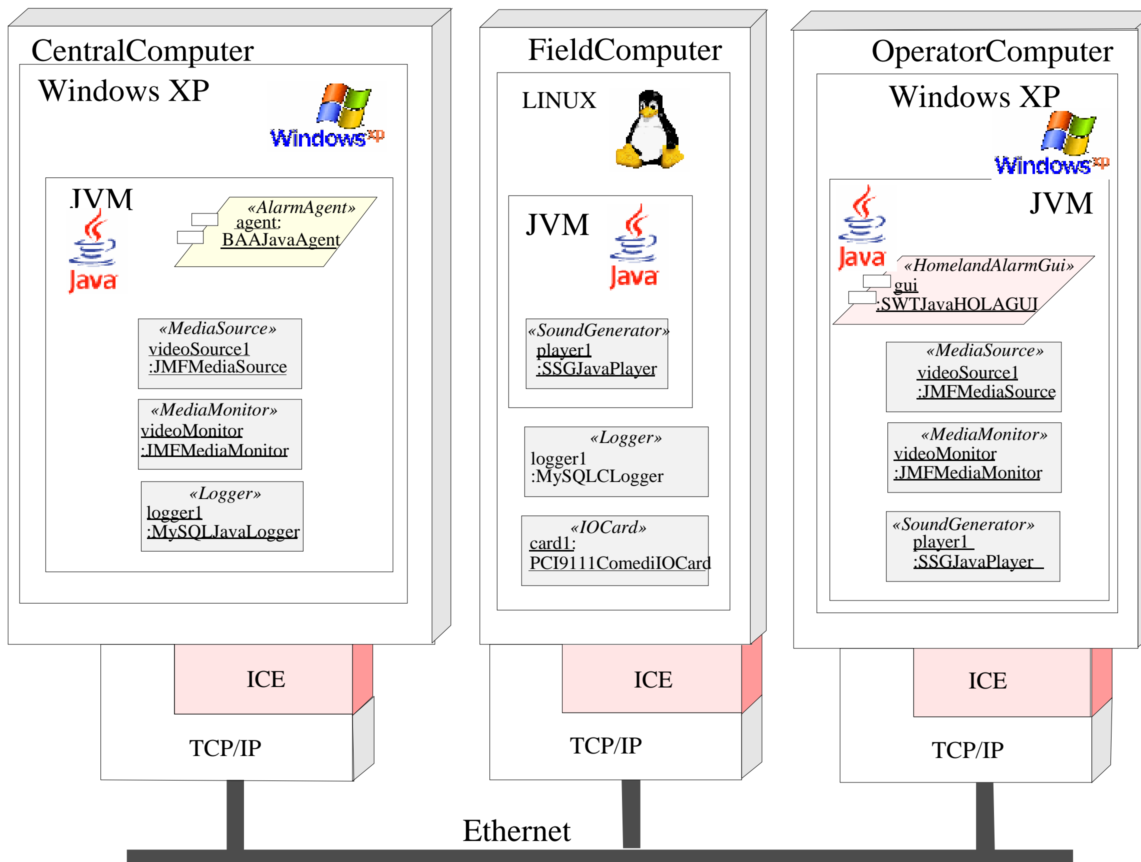

Figura 4.3.2.Plataforma y distribución de las instancias de los componentes

A continuación, se muestra un ejemplo de la descripción de una plataforma. Esta es la que corresponden a la configuración "*ParkingSystem*" que se describe en el Anexo B.

File: "ParkingSystem.tdm.xml"

```
<?xml version="1.0" encoding="UTF-8"?>
<DnCtdm:domain xmlns:DnCtdm="http://ctr.unican.es/cbsdnc/DnCTargetDataModel"
xmlns:DnCbt="http://ctr.unican.es/cbsdnc/DnCBasicTypes"
xmlns:DnCct="http://ctr.unican.es/cbsdnc/DnCCommonTypes"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ctr.unican.es/cbsdnc/DnCTargetDataModel
C:\XML\ICE\DnCTargetDataModel.xsd" UUID="http://ctr.unican.es/Java-CCM/hola/ParkingSystem/">
        <node name="FieldComputer" label="Installation node of the PCI9111IOCard">
                <resource resourceType="os" name="theLinuxOs">
                        <property name="type" kind="ATTRIBUTE"><value>Ubuntu</value></property>
                        <property name="version" kind="ATTRIBUTE"><value>6.06</value></property>
                        <property name="libs" kind="ATTRIBUTE"><value>comedilib-0.7.22 comedi-
0.7.73</value>

</property>
                </resource>
                <resource resourceType="run-time_Environment" name="theRunTimeEnvironment">
                        <property name="type" kind="ATTRIBUTE"><value>ZEROC
ICE</value></property>
                        <property name="version" kind="ATTRIBUTE"><value>3.1.0</value></property>
                </resource>
                <resource resourceType="IO_Card" name="theIOCard">
                        <property name="type"
kind="ATTRIBUTE"><value>PCI9111</value></property>
                        <property name="provider"
kind="ATTRIBUTE"><value>ADLINK</value></property>
                </resource>
                <connection>HomeLandAlarmNetwork</connection>
        </node>
……….
    <interconnect name="HomeLandAlarmNetwork" label="LocalNetWork">
                <connect>FieldComputer</connect>
                <connect>OperatorComputer</connect>
                <connect>CentralComputer</connect>
                <resource name="theProtocol" resourceType="protocol">
                        <property name="type" kind="ATTRIBUTE">
                                <value>TCP/IP</value>
                        </property>
                </resource>
        </interconnect>
</DnCtdm:domain>
```

(La declaración completa de la plataforma puede encontrase el Anexo B)

La declaración de la plataforma *ParkingSystem* corresponde a la aplicación *ParkingSystem* que se describe en el Anexo B. En él, se reúne en un documento la declaración de los nudos procesadores que la componen (*FieldComputer, OperadorComputer* y *CentralComputer*) y la red de comunicación que los interconecta (*HomeLandAlarmNetwork*).

Cada elemento de la plataforma de caracteriza mediante los recursos que ofrece (p.e. el nudo *FielComputer* declara los recursos del sistema operativo (*OS*), el *middleware* (*run-time_Environment*) y una tarjeta de I/O (*IO_Card*)).

Cada recurso se caracteriza por un conjunto de propiedades (p.e. el recurso sistema operativo *theLinuxOS* del nudo *FieldComputer* tiene las propiedades *type: Ubuntu*, *version:6.6* y *lib: comedilib-0.7.22 comedi-0.7.73).*

Estos recursos y propiedades de cada nudo de la plataforma serán contrastados con los recursos que requieren los componentes, a fin de seleccionar la implementación y para validar un determinado plan de despliegue.

IV.4-Plan de despliegue. Descripción D&C.

Una aplicación basada en componentes se describe definiendo las instancias de los componentes que constituyen la aplicación, estableciendo para cada instancia los valores que deben asignarse a las propiedades de configuración, definiendo y calificando las conexiones entre los componentes y asignando a cada instancia el nudo de la plataforma en que debe instanciarse. Toda esta información se reúne en el documento que denominamos "plan de despliegue". Este plan se puede generar de forma efímera para ser utilizado desplegando y ejecutando inmediatamente la aplicación sobre la plataforma, o de forma más permanente, para ser almacenado y facilitar la ejecución de la aplicación en la plataforma de forma ágil en el futuro. El plan de despliegue es autocontenido, esto es, contiene toda la información que se necesita para desplegar la aplicación, sin necesidad de recurrir de nuevo a las diversas bases de datos en las que se almacena la información de los componentes y de la plataforma de ejecución. A partir de él, el gestor de ejecución (*Executor*) puede generar los objetos que una vez desplegados constituyen la aplicación.

En la siguiente figura 4.4, se muestran el diagrama con las clases raíces del modelo de ejecución. Las estructuras de datos que corresponden a estas clases, se definen en la plantilla *DnCExecutionDataModel.xsd*, cuyo contenido puede encontrarse en el Anexo C.

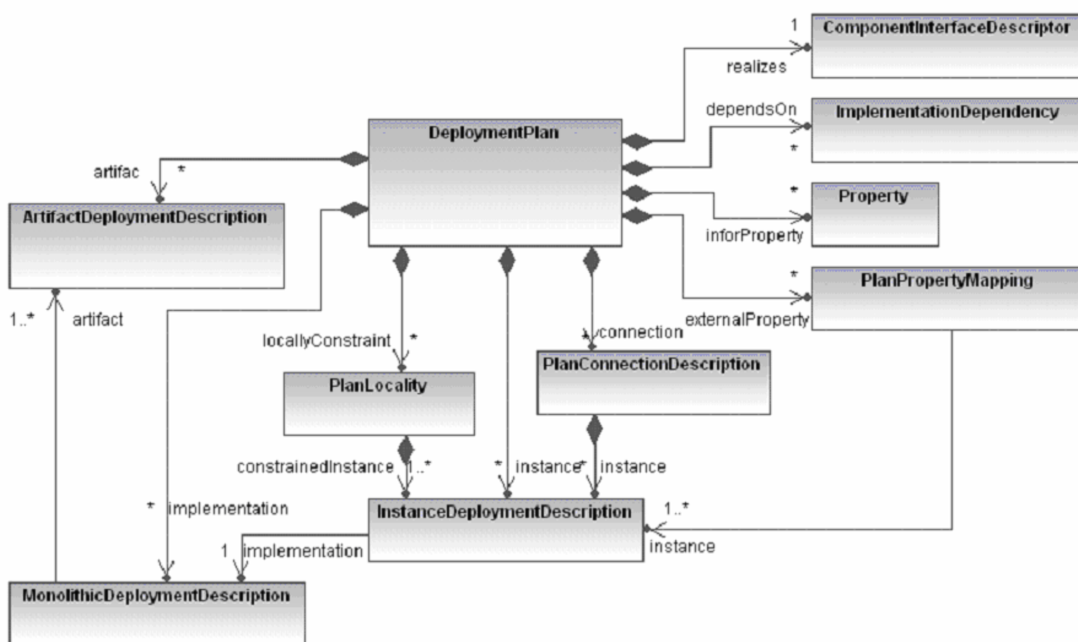

Figura 4.4. Clases raíces que sirven para definir el plan de despliegue de una aplicación.

- **Deployment Plan**: Representa el despliegue de una aplicación sobre una determinada plataforma. Contiene la información relativa a los ficheros de código que son parte del despliegue (*ArtifactDeploymentDescription*), la forma de crear las instancias de los componentes (*MonolithicDeploymentDescription*), y donde se han de instanciar (*InstanteDeploymentDescription*). Contiene también información relativa a la conexión entre los subcomponentes (*Assembly ConnectionDescription*) y a la correspondencia entre los puertos externos y los de los subcomponentes. Por último, contiene la descripción de la interfaz que es implementada por la aplicación que se despliega (*ComponentInterface Descriptor*). El plan de despliegue es análogo a la descripción de un componente compuesto (*ComponentAssemblyDescription*) en el modelo de datos de un componente, salvo que en el despliegue la composición es plana, esto es, está sólo compuesta por componentes monolíticos. En el plan, todos los

componentes compuestos tienen que ser recursívamente reemplazados por sus implementaciones concretas, las cuales han sido especificadas para cada subcomponente. También pueden tener información adicional de localización (*PlanLocality*) introducida con el plan de despliegue de instancias (*Instance DeploymentDescription*).

- **ComponentInterfaceDescriptor**: Describe la interfaz de la aplicación que se despliega, esto es, la descripción de su comportamiento visto desde fuera.
- **ImplementationDependency**: Formula una dependencia de la implementación de un componente respecto del entorno de la plataforma. Indica qué otras instancias de componentes o servicios deben estar instaladas en la plataforma antes de que la implementación sea desplegada.
- **PlanPropertyMapping**: identifica la correspondencia entre una propiedad o puerto de la aplicación que se despliega y la propiedad o puerto del subcomponente en el que delega.
- **ArtifactDeploymentDescription**: Describe un fichero relativo a la información de un componente que debe ser desplegado como parte del plan de despliegue de la aplicación. Contiene el localizador (*URL*) del fichero y los parámetros y requisitos de despliegue de cada componente, lo que hace autocontenido el plan de despliegue (*ImplementationArtifactDescription*).
- **MonolithicDeploymentDescription**: Describe el despliegue de un componente como parte de un plan. Referencia la descripción de los elementos que son parte del despliegue (*ComponentImplementacionDescription*)
- **PlanLocality**: Define las restricciones de instanciación de los diferentes componentes en el plan de despliegue. Permite especificar si los componentes de la aplicación que se despliegan deben ser instanciados en un mismo proceso, en diferentes procesos o libremente.
- **PlanConnectionDescription**: Describe una conexión que se establece entre puertos de los componentes que constituyen la aplicación.
- **InstanceDeploymentDescription**: Contiene la información que es necesaria para desplegar una instancia de componente simple. Hace referencia a una descripción de un componente monolítico (*MonolithicDeploymentDescription*) e incluye el nombre de los nodos en los que es instanciado el componente. Además, contiene propiedades que son usadas para configurar la instancia del componente.

En la tabla se muestra un ejemplo de plan de despliegue de una aplicación basada en componentes. Esta es la aplicación "*Parking Control*" que se puede encontrar descrita en el Anexo B.

File: "HolaParking.cdp.xml"

```
<?xml version="1.0" encoding="UTF-8"?>
<DnCedm:deploymentPlan xmlns:DnCedm=”http://ctr.unican.es/cbsdnc/DnCExecutionDataModel”
        xmlns:DnCbt=”http://ctr.unican.es/cbsdnc/DnCBasicTypes”
        xmlns:DnCcdm=”http://ctr.unican.es/cbsdnc/DnCComponentDataModel”
        xmlns:DnCct=”http://ctr.unican.es/cbsdnc/DnCCommonTypes”
        xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
        xsi:schemaLocation="http://ctr.unican.es/cbsdnc/DnCExecutionDataModel

:\XML\ICE\DnCExecutionDataModel.xsd">
          <realizes>
                  <ref>components/alarm/HolaParking.ccd.xml</ref>
          </realizes>
          <target UUID="http://ctr.unican.es/Java-CCM/hola/ParkingSystem/"</target>
<instance name="player1" node="OperatorComputer"
```

```
                source="components/soundGenerator/SoundGenerator.ccd.xml::jssSoundGenerator">
                <configProperty name="OBJECTNAME">
<value><string>OperatorBell</string></value></configProperty>
                <configProperty name="PROTOCOLDATA"><value> <string>default -p 6000</string></value>


</configProperty>
                <configProperty
name="dingSound"><value><string>/rsrc/ding.wav</string></value></configProperty>
                <configProperty
name="chordSound"><value><string>/rsrc/chord.wav</string></value></configProperty>
<configProperty name="defSound"><value><string>/rsrc/defsound.wav</string></value></configProperty>
                            <configProperty
name="chimesSound"><value><string>/rsrc/chimes.wav</string></value></configProperty>
                <configProperty
name="alarmSound"><value><string>/rsrc/alarm.wav</string></value></configProperty>
                <configProperty
name="failSound"><value><string>/rsrc/fail.wav</string></value></configProperty>
                <deployedResource requirementName="run-time_Enviroment_Requirement"
                    resourceName="theJavaVirtualMachine"
                    resourceUsage="NONE">
                </deployedResource>
            </instance>
            <instance name="videoMonitor1" node="OperatorComputer"
                source="components/media/MediaMonitor.ccd.xml::JMFMediaMonitor">…</instance>

            <instance name="videoSource1" node="OperatorComputer"
                source="components/media/MediaSource.ccd.xml::JMFMediaSource">…</instance>
            <instance name="gui" node="OperatorComputer"
                source="components/home/AlarmGui.ccd.xml::SWTJavaHOLAGUI">…</instance>
            <instance name="logger1" node="FieldComputer"
                source="components/logger/Logger.ccd.xml::MySQLJavaLogger">…</instance>
            <instance name="ioCard" node="FieldComputer"
                source="components/iocard/IOcard.ccd.xml::PCI9111ComediIOCard">…</instance>
            <instance name="agent" node="CentralComputer"
                source="components/alarmagent/AlarmAgent.ccd.xml::BAAJavaAgent" >…</instance>
            <instance name="videoSource2" node="CentralComputer"
                source="components/media/MediaSource.ccd.xml::JMFMediaSource">…</instance>
            <instance name="videoMonitor2" node="CentralComputer"
                source="components/media/MediaMonitor.ccd.xml::JMFMediaMonitor">…</instance>
            <instance name="logger2" node="CentralComputer"
                source="components/logger/Logger.ccd.xml::MySQLJavaLogger">…</instance>
            <connection name="agentToplayer1">
                    <internalEndpoint portName="theSoundGenerator" instance="agent"/>
                    <internalEndpoint portName="iPlayer" instance="player1"/>
            </connection>
            <connection name="agentTovideoSource1">…</connection>
            <connection name="agentTovideoSource2">…</connection>
            <connection name="agentTovideoMonitor1">…</connection>
            <connection name="agentToVideoMonitor2">…</connection>
            <connection name="agentTologger1">…</connection>
            <connection name="agentTocard1">…</connection>
            <connection name="logger1TovideoSource1">…</connection>
            <connection name="logger1TovideoSource2">…</connection>
            <connection name="logger1TovideoMonitor1">…</connection>
            <connection name="logger1TovideoMonitor2">…</connection>
            <connection name="logger1Toagent">…</connection>
            <connection name="loggerToplayer1">
                    <internalEndpoint portName="theSoundGenerator" instance="logger1"/>
                    <internalEndpoint portName="iPlayer" instance="player1"/>
            </connection>
            <connection name="agentTogui">…</connection>
            <connection name="guiToLoggerEvent">…</connection>
</DnCedm:deploymentPlan>
```

El plan de despliegue corresponde al ejemplo *HolaParking* que se encuentra descrito en el Anexo B. En él se describen todos los elementos que constituyen la aplicación y proporciona la información que necesita la herramienta de ejecución para instanciar, configurar, enlazar y ejecutar la aplicación distribuida en la plataforma.

El documento declara las instancias que son parte de la aplicación (p.e. *player_1*). Para cada una de ellas detalla el nodo de la plataforma en que se instancia (p.e. *OperatorComputer*) y el descriptor de la implementación del componente del que es instancia (p.e. *components/soundGenerator/SoundGenerator.ccd.xml::jssSoundGenerator*). Así mismo, para cada una de ellas declara los valores de sus propiedades de configuración, las cuales, unas están declaradas en la interfaz del componente (p.e. *dingSound*) y otras suelen ser de configuración de aspectos relativos a la plataforma que están definidas por la tecnología CCM (p.e. *PROTOCOLDATA*).

También declara las conexiones entre componentes (p.e. *agentToplayer1*), que indican en sus atributos las instancias que interconectan y las propiedades de la conexión.

Conclusiones y trabajo futuro

En este trabajo se ha propuesto la tecnología de componentes Java-CCM que sigue la especificación LwCCM de OMG y que implementa el paradigma (Container Component Model). Con esta tecnología se ha demostrado la utilidad de este paradigma para desarrollar aplicaciones basadas en componentes que se ejecutan en sistemas distribuidos construidos alrededor de un middleware estándar que no soporta directamente los componentes. El middleware que se ha utilizado en este trabajo ha sido ICE de la empresa ZeroC.

Características propias de las tecnologías de componentes que se conseguido con la tecnología Java-CCM propuesta son:

- Los Componentes se describen mediante ficheros XML con formato y contenido que corresponde a la especificación D&C de OMG.
- La tecnología no requiere que el diseñador del código de los componentes que tenga conocimientos de ICE, ni de las características de la plataforma distribuida en que se vaya a ejecutar. Él desarrolla el código de negocio como si se fuese a ejecutar en un entorno monoprocesador estándar.
- Los componentes que se generan son compatibles con cualquier tecnología Container Componen Model, aunque no esté basada en el middleware ICE.
- Los componentes se distribuyen en un único paquete jar que incluye el código, el modelo y la información instropectiva del componente. El componente puede ser reutilizado en diferentes aplicaciones sin que se necesite conocer ni modificar su código.
- Se han especificado las herramientas que de forma automática generan el código del contenedor que hace posible que el componente se ejecute en una determinada plataforma y se comunique con otros componentes en el contexto de una aplicación.
- Las aplicaciones basadas en componentes de la tecnología propuesta se describen mediante un documento XML que sigue el estándar D&C de OMG.
- Se ha desarrollado un prototipo básico de programa de lanzamiento local, que tiene capacidad de instanciar, configurar y lanzar la ejecución de los componentes de una aplicación correspondiente a un nodo de la plataforma, en función de la información contenida en su plan de despliegue final.

La tecnología Java-CCM que se muestra en este trabajo se encuentra reducida a componentes con código de negocio formulado en lenguaje Java, y a componentes que se ejecutan en JVM (Java Virtual Machine). Aunque los componentes son compatibles con componentes implementados en otros lenguajes (por ejemplo CPP_CCM) y desarrollados dentro del mismo proyecto.

En la versión actual, se ha previsto la gestión explícita de los threads por parte del contenedor, pero no se han elaborado mecanismos para controlar la planificación de los threads, de forma que se puedan diseñar aplicaciones con requisitos de tiempo real, ya sean laxos o estrictos. Justamente en esta dirección se tiene planificada la continuación de este trabajo:

- ***Extensión de la tecnología Java-CCM a tiempo real estricto***: Para poder desarrollar aplicaciones de tiempo real estricto se requiere poder predecir el comportamiento temporal de las respuestas a eventos externos o temporizados de las aplicaciones. Actualmente esto no es posible por dos razones:
    - La JVM de SunMicrosystem que se usa no es de tiempo real, por lo que habrá que migrar a otra que si ofrezca predecibilidad de sus servicios. Una

línea abierta sería la utilización de la JVM desarrollada por la Universidad de York sobre el sistema operativo Marte OS.

- El *middleware* ICE que se usa no es de tiempo real, y deberá se modificado para que, por un lado, incorpore protocolos y servicios de comunicación de tiempo real (bus CAN, RTEP, etc), y por otro, los algoritmos en que se basan sus servicios tengan tiempo de respuesta limitada.

- ***Incorporación de la planificación jerárquica y contratos de reserva de recursos***. Si cada nudo de la plataforma de ejecución tiene un único planificador, como ocurre en la versión actual, no es posible formular modelos de comportamiento de tiempo real para cada componente de forma independiente. Bajo esta situación el modelo de comportamiento temporal de un componente depende del comportamiento temporal de los componentes de que se sirve, y de la carga del procesador. Por ello, el modelo de tiempo real de un componente, sólo se puede generar y analizar en el contexto de una aplicación y para un modelo de carga de la plataforma. Esto es siempre complejo para una plataforma que ejecuta diferentes aplicaciones, e imposible para plataformas abiertas en las que la carga de trabajo no es conocida. Con la incorporación a cada componente de un planificador local que gestione la capacidad de procesamiento (o capacidad de comunicación) que se le asigna, y con la utilización de servicios de reserva de recursos que puede ofrecer la plataforma, el modelo de comportamiento de los componentes y de las aplicaciones se pueden hacer auto-contenidos y los procesos de modelado y análisis del comportamiento temporal se simplifican.

Se pretende que las plataformas, al igual que los componentes estén descritas mediante modelos, y meta-herramientas tengan capacidad de generar las herramientas que se necesitan para desplegar los aplicaciones (tecnología MDA).

Referencias

[1] Proyecto HESPERIA: **H**omeland s**E**curity: tecnología**S** **P**ara la Seguridad integ**R**al en espac**I**os públicos e infr**A**estructuras. Proyecto CENIT- 2005. https://www.proyecto-hesperia.org/.

[2] M. Henning y M. Spruiell: "Distributed Programming with ICE". http://www.zeroc.com.

[3] "Differences between Ice Vs Corba" http://www.zeroc.com/iceVsCorba.html

[4] Szyperski C.: "Component Software: Beyond Object-Oriented Programming" Addison Wesley, 1998.

[5] OMG: "CORBA Components Version 3.0" Formal/02-06-65. (2002).

[6] OMG: "CORBA Component Model Version 4.0" Formal/2006-04-01. (2006).

[7] ITEA project MERCED (Market-Enabler for Retargetable COTS components in Embedded Domain). http://www.itea-merced.org.

[8] IST projects: "COMPARE (Component-based approach for real-time and embedded systems)". http://www.ist-compare.org.

[9] IST project: "FRESCOR (Framework for Real-time Embedded Systems based on Contracts)". http://www.frescor.org

[10] OMG: Lightweight Corba Component Model, ptc/03-11-03, November 2003

[11] THALES Land & Joint Systems, France: "Experience Report on Implementing and Applying a Standard Real-Time Embedded Component Platform", OMG RTE Systems Workshop ,Washington, July 2007

[12] P.López, J.M. Drake, P. Pacheco, J.L. Medina: "An Ada 2005 Technology for Distributed and Real-Time Component-Based Applications" 13th International Conference on Reliable Software Technologies – Ada Europe (2008).

[13] L.Barros, P.López, J.M. Drake :"Tecnología de componentes CCM basada en conectores" XVI Jornadas de Concurrencia y Sistemas Distribuidos, Albacete 2008.

[14] OMG: Deployment and Configuration of Component-Based Distributed Applications Specification, version 4.0, Formal/06-04-02, April 2006

[15] P.López, J.L. Medina and J.M. Drake, Real-Time Modelling of Distributed Component-based Applications, EUROMICRO-SEAA'06, Agosto 2006

[16] OMG: "Quality of Service for CORBA Components", ptc/06-04-05. April 2006

[17] OMG: Real-Time CORBA Specification, v1.2 formal/05-01-04. Enero 2005

[18] I. Crnkovic y M. Larsson: "Building Reliable Component-based Software Systems" Artech House Publishers, 2002.

[19] Wallnau K. y otros: "Technical Report CMU/SEI:"

Vol. I: "Market Assessment of Component-Based Software Engineering" CMU/SEI-2000-TN-007.

Vol. II: Technical Concepts of Component-Based Software Engineering" CMU/SEI-2000-TR-008, ESC-TR-2000-007.

Vol. III: Technical Concepts of Component-Based Software Engineering" CMU/SEI-2003-TR-009, ESC-TR-2003-009.

Pittsburgh, PA: Software. Engineering Institute, Carnegie Mellon University.