ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Proyecto Fin de Carrera

Ada-CCM: Tecnología de componentes basada en el lenguaje de programación Ada 2005

(Ada-CCM: Components technology based on Ada 2005 programming language)

Para acceder al Titulo de

INGENIERO DE TELECOMUNICACIÓN

Autor: Pablo Pacheco Munguía

Octubre – 2007

ÍNDICE

TECNOLOGÍA DE COMPONENTES CCM..... 1

1.1 TECNOLOGÍAS DE COMPONENTES.....1

1.1.1 Concepto de componente y aplicaciones basadas en componentes 1

1.1.2 Beneficios y problemas.....2

1.2 TECNOLOGÍAS DE COMPONENTES CCM. MODELO DE REFERENCIA. 5

1.2.1 Tecnología CCM origen como extensión de la tecnología CORBA. 5

1.2.2 Elementos básicos del modelo de referencia CCM 7

1.3 GENERALIZACIÓN DE LA TECNOLOGÍA CCM 9

1.3.1 Problemas del uso de CORBA en sistemas empotrados mínimos.....9

1.3.2 Elementos básicos del modelo de referencia CCM generalizado 9

1.4 ESPECIFICACIÓN DE DESPLIEGUE Y CONFIGURACIÓN DE OMG 11

1.5 OBJETIVOS DEL PROYECTO 12

PROCESO DE DESARROLLO DE UNA APLICACIÓN..... 15

2.1 PROCESO DE DESARROLLO DE UNA APLICACIÓN BASADA EN COMPONENTES 15

2.2 DESCRIPCIÓN IDL DE UN COMPONENTE: CORRESPONDENCIA ENTRE TIPOS IDL Y ADA. 17

2.2.1 Descripción de una interfaz y de un componente en IDL 17

2.2.2 Correspondencia entre estructuras IDL y Ada.....19

2.3 ESTRUCTURA GENÉRICA DE LOS COMPONENTES.....20

2.4 ELEMENTOS DEL CONTENEDOR DE UN COMPONENTE.....22

2.5 FASES DEL PROCESO DE DESARROLLO DE UNA APLICACIÓN BASADA EN COMPONENTES 24

2.5.1 Fase de diseño de la aplicación.....24

2.5.2 Fase de despliegue de una aplicación 26

2.5.3 Fase de lanzamiento de la aplicación.....28

ESTRUCTURA DE LOS CONTENEDORES DE COMPONENTE 29

3.1 Estructura del Ejecutor....29
3.2 Estructura del Wrapper....33
3.3 Estructura de la implementación....38
3.3.1 El paquete [NombreComponente]_Exec_Impl....39
3.3.2 El paquete [NombreComponente]_Impl....41

Estructura de los conectores entre componentes....45

4.1 Estructura genérica de un conector....45
4.1.1 Fragmento Proxy del conector....47
4.1.2 Fragmento Servant del conector....47
4.2 Estructura de un conector que implemente el DSA de Ada....47
4.2.1 Introducción a GLADE....47
4.2.2 Implementación del fragmento Servant con GLADE....48
4.2.3 Implementación del fragmento Proxy con GLADE....49
4.2.4 Proceso de despliegue de los conectores....50

Herramientas para la generación automática de código....53

5.1 Estructura del repositorio....53
5.2 Estrategia de generación de código y herramientas de generación....54
5.2.1 Ejemplo de funcionamiento de la herramienta de generación de código....56
5.3 Generación del código Ada del contenedor de un componente....58
5.3.1 Paquetes generados correspondientes a un componente....58
5.3.2 Paquetes generados correspondientes a los conectores....59

JetFollower: Ejemplo de aplicación basada en componentes....61

6.1 Funcionalidad y arquitectura de la aplicación....61
6.2 Descripción de los componentes que se utilizan....63
6.2.1 El componente TrackDriver....63

6.2.2 El componente ServosController *65*

6.2.2.1 La interfaz I_Control *66*

6.2.3 El componente SoundGenerator *68*

6.2.3.1 La interfaz I_Player *68*

6.2.4 El componente Database *70*

6.2.4.1 La interfaz I_Logger *70*

6.2.5 El componente PCI9111Card *71*

6.3 DESCRIPCIÓN DE LA PLATAFORMA 71

6.4 PLAN DE DESPLIEGUE DE LA APLICACIÓN 73

6.5 FICHEROS QUE CONSTITUYEN LA APLICACIÓN 75

CONCLUSIONES Y LÍNEAS FUTURAS DE TRABAJO 78

REFERENCIAS BIBLIOGRÁFICAS 80

ANEXO: PLANTILLAS UTILIZADAS POR LAS HERRAMIENTAS DE GENERACIÓN DE CÓDIGO 82

PLANTILLA PARA LA GENERACIÓN DE LAS INTERFACES 82

PLANTILLAS PARA LA GENERACIÓN DE LOS PAQUETES ADS/ADB [NOMBRECOMPONENTE]_TYPES 84

PLANTILLA PARA LA GENERACIÓN DEL PAQUETE ADS [NOMBRECOMPONENTE]_EXEC 86

PLANTILLAS PARA LA GENERACIÓN DE LOS PAQUETES ADS/ADB [NOMBRECOMPONENTE]_EXEC_IMPL 89

PLANTILLAS PARA LA GENERACIÓN DE LOS PAQUETES ADS/ADB [NOMBRECOMPONENTE]_IMPL 93

PLANTILLAS PARA LA GENERACIÓN DE LOS PAQUETES ADS/ADB [NOMBRECOMPONENTE]_WRAPPER 105

PLANTILLA PARA LA GENERACIÓN DE LA INTERFAZ EQUIVALENTE DEL COMPONENTE 115

PLANTILLA PARA LA GENERACIÓN DE LOS PAQUETES ADS DE LAS FACETAS 116

PLANTILLAS PARA LA GENERACIÓN DEL LOS PAQUETES ADS/ADB DE LOS ADAPTER DE LOS CONECTORES 117

PLANTILLAS PARA LA GENERACIÓN DEL LOS PAQUETES ADS/ADB DE LOS RCI DE LOS CONECTORES 120

PLANTILLA PARA LA GENERACIÓN DEL LOS PAQUETES [NOMBRECONECTOR]_IMPL DE LOS CONECTORES 122

ÍNDICE DE FIGURAS

FIGURA 1.1: ELEMENTOS DE UNA TECNOLOGÍA DE COMPONENTES. 2
FIGURA 1.2: EVOLUCIÓN DE LA FUNCIONALIDAD DE CORBA. 6
FIGURA 1.3: ESQUEMA DE LOS PUERTOS DE UN COMPONENTE CCM. 8
FIGURA 1.4: ESQUEMA DE LOS ELEMENTOS BÁSICOS DEL MODELO CCM GENERALIZADO. 9
FIGURA 1.5: FASES DEL PROCESO DE DESPLIEGUE SEGÚN D&C. 12
FIGURA 2.1: FASES DEL DESARROLLO EN LA TECNOLOGÍA ADA-CCM. 16
FIGURA 2.2: ESTRUCTURA GENÉRICA DE UN COMPONENTE. 21
FIGURA 2.3: DETALLE DE LA ESTRUCTURA DEL CÓDIGO DE NEGOCIO EN EL COMPONENTE. 22
FIGURA 2.4: ESTRUCTURA DEL WRAPPER. 23
FIGURA 2.5: FASE DE DISEÑO DE UNA APLICACIÓN. 25
FIGURA 2.6: FASE DE DESPLIEGUE DE UNA APLICACIÓN. 26
FIGURA 2.7: GENERACIÓN DE LOS CONECTORES (DIAGRAMA SIMPLIFICADO). 27
FIGURA 2.8: FASE DE LANZAMIENTO DE LA APLICACIÓN. 28
FIGURA 3.1: DIAGRAMA UML DE LA *MONOLITHIC EXECUTOR INTERFACE*. 31
FIGURA 3.2: DIAGRAMA UML DE LA INTERFAZ DE CONTEXTO DEL PAQUETE DEL EJECUTOR. 32
FIGURA 3.3: DIAGRAMA UML DE LAS CLASES CORRESPONDIENTES AL HOME DEL EJECUTOR. 33
FIGURA 3.4: DIAGRAMA UML DE LAS CLASE CORRESPONDIENTE AL WRAPPER. 35
FIGURA 3.5: DIAGRAMA UML DE LAS CLASES DEL CONTEXTO DEL WRAPPER. 36
FIGURA 3.6: DIAGRAMA UML DE LAS CLASES CORRESPONDIENTES AL HOME DEL WRAPPER. 37
FIGURA 3.7: DIAGRAMA DE SECUENCIAS DE LA INSTANCIACIÓN DE UN COMPONENTE. 38
FIGURA 3.8: DIAGRAMA UML DE LA CLASE INSTANTE DEL PAQUETE EXEC_IMP. 40
FIGURA 3.9: DIAGRAMA UML DE LA CLASE HOME DEL PAQUETE EXEC_IMP. 40
FIGURA 3.10: DIAGRAMA UML DE LA CLASE INSTANCE DE LA IMPLEMENTACIÓN. 41
FIGURA 3.11: DIAGRAMA UML DE LA CLASE CORRESPONDIENTE AL ESTADO. 44
FIGURA 4.1: ESQUEMA DE CONEXIONES DE UN CONECTOR CON LOS COMPONENTES. 46
FIGURA 4.2: ESQUEMA DEL PROCESO DE REINVOCACIÓN DE PETICIONES EN EL SERVANT. 49
FIGURA 4.3: ESQUEMA DEL PROCESO DE REINVOCACIÓN DE PETICIONES EN EL PROXY. 50
FIGURA 4.4: ESQUEMA DEL DESPLIEGUE DE LOS CONECTORES GLADE. 51
FIGURA 5.1: PROCESO DE GENERACIÓN DE PAQUETES ADA. 59
FIGURA 6.1: ARQUITECTURA DE LA APLICACIÓN. 62
FIGURA 6.2: DESCRIPCIÓN UML DEL COMPONENTE TRACKDRIVER. 63
FIGURA 6.3: DIAGRAMA DE SECUENCIAS DEL COMPONENTE TRACKDRIVER. 64
FIGURA 6.4: DIAGRAMA UML DE LAS CLASES QUE FORMAN EL COMPONENTE SERVOSCONTROLLER. 66
FIGURA 6.5: DIAGRAMA UML DE LA INTERFAZ I_CONTROL. 67
FIGURA 6.6: DIAGRAMA UML DE LA INTERFAZ I_PLAYER. 69
FIGURA 6.7: DIAGRAMA UML DE LA INTERFAZ I_LOGGER. 70
FIGURA 6.8: DIAGRAMA UML DE LA INTERFAZ I_ANALOGIO. 71
FIGURA 6.9: ESQUEMA DE LA INFORMACIÓN CONTENIDA EN LA DESCRIPCIÓN DE LA PLATAFORMA. 72
FIGURA 6.10: UBICACIÓN DE LAS INSTANCIAS DE LOS COMPONENTES Y SUS CONEXIONES. 73
FIGURA 6.11: ESTRUCTURA DEL REPOSITORIO. 75
FIGURA 6.12: FICHEROS EN EL REPOSITORIO DEL COMPONENTE SOUNDGENERATOR. 75
FIGURA 6.13: LOCALIZACIÓN DEL FICHERO QUE DEFINE LA PLATAFORMA. 76
FIGURA 6.14: LOCALIZACIÓN DEL PLAN DE DESPLIEGUE EN EL REPOSITORIO. 77
FIGURA 6.15: LOCALIZACIÓN DEL CÓDIGO DE LOS CONECTORES. 77

CAPÍTULO 1

TECNOLOGÍA DE COMPONENTES CCM

1.1 TECNOLOGÍAS DE COMPONENTES

1.1.1 Concepto de componente y aplicaciones basadas en componentes

El rápido crecimiento del mercado de las tecnologías de la información, y dentro de él, el espectacular auge de los sistemas abiertos y distribuidos, han hecho necesario cambiar la metodología de desarrollo de las aplicaciones informáticas, y es la causa directa de la demanda de la 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.

En la siguiente figura se representan los conceptos básicos utilizados en las tecnologías de componentes [1][2]. Un componente (1) 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.

A fin de incrementar la reusabilidad, la funcionalidad de los componentes se describe a través de interfaces (2), que son definidas con independencia de ellos. Las aplicaciones se diseñan ensamblando entre sí, instancias de componentes que ya están definidos y diseñados, con independencia de ellas. A fin de que las interacciones entre componentes desarrollados independientemente sean efectivas, cada componente formula como contratos (3) las obligaciones y responsabilidades que debe satisfacer en la interacción con los otros componentes y con el entorno de ejecución.

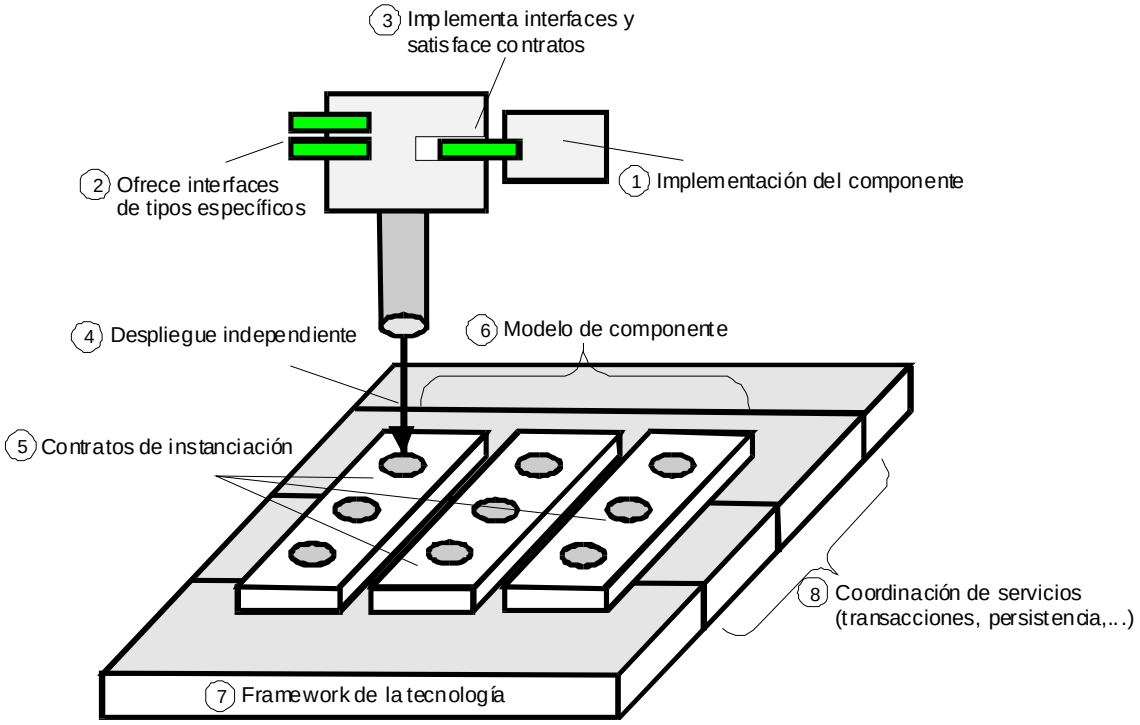

Figura 1.1: Elementos de una tecnología de componentes.

Cada componente puede ser desplegado independientemente en un entorno de ejecución (4), que se encuentra definido por la tecnología a la que pertenece. Los requisitos que el componente requiere de la plataforma de ejecución se formulan como un contrato de instanciación (5). Un sistema basado en componentes se construye utilizando múltiples instancias de un pequeño conjunto de tipos de componentes, cada tipo juega un papel especializado en el sistema y funcionalmente se describe por una interfaz. El modelo de componente (6) de una tecnología, define los elementos que deben utilizarse para describir cada tipo de componente, esto es, como formular sus interfaces y los patrones de interacciones con otros componentes que pueden realizar. El modelo de referencia (framework) (7) del componente describe los mecanismos de comunicación entre los componentes y los servicios del entorno de ejecución que hacen posible que un componente pueda instanciarse en diferentes nudos ofreciendo un mismo comportamiento, y así mismo, resuelve muchos de los aspectos de bajo nivel que conciernen a la ejecución del componente (identidad, persistencia, gestión de errores, etc.).

1.1.2 Beneficios y problemas

La confianza que actualmente existe sobre las tecnologías de componentes software es consecuencia de los siguientes beneficios que se esperan de ellas:

- Los componentes mejoran la productividad de programador: La tecnología de componentes es una tecnología orientada a la reutilización del software que se produce. Un principio general de cualquier ingeniería, es que con la

reutilización se incrementa siempre la calidad y la productividad del producto que se genera. Existen estudios que avalan este incremento [3]. En estos análisis, se demuestra que el incremento de productividad no es inmediato, sino que sólo aparece cuando se sobrepasa un cierto umbral de reutilización. En las primeras generaciones de los productos se reduce la productividad, ya que hay que invertir cierto trabajo en potenciar su reutilización posterior, sin embargo, cuando la empresa tiene productos desarrollados, o cuando el mercado de componentes ya existe, la productividad se incrementa en mas del 50% [4].

- Los componentes reducen los tiempos de acceso al mercado: Este beneficio está muy ligado a la mejora de la productividad, aunque existen estudios que afirman que en el caso de la tecnología de componentes existen razones adicionales a esta ventaja [5]:

    - Las tecnologías de componentes generan líneas verticales de producción dentro de cada dominio de aplicación, lo que contribuye con la definición de arquitecturas y contextos muy bien definidos. En ellos, los componentes constituyen bloques constructivos que se ensamblan de una forma eficiente entre si, dentro del contexto restringido que se crea.
    - El uso de las tecnologías de componentes eleva el nivel de abstracción de los elementos que se desarrollan, relegando al modelo de referencia de la propia tecnología, muchas de las fuentes de complejidad, que son aspectos de bajo nivel (por ejemplo, persistencia, transacciones, gestión de errores, seguridad, etc.). Cuando se utiliza una tecnología de componentes, el programador puede centrar todo su esfuerzo en la lógica y en la funcionalidad de la aplicación.

- Los componentes contribuyen a incrementar la calidad del software: Existen estudios [6] de que el incremento de calidad ocurre cuando se opera en un mercado estable de componentes, el cual está sometido a autoridades confiables que pueden certificar la calidad de los componentes que se utilizan.

- Los componentes proporcionan la base del mercado de software reutilizable: Este beneficio de naturaleza económica, es la principal causa que impulsa el desarrollo de la tecnología de componentes. Las tecnologías de componentes proporcionan mecanismos para empaquetar todos los elementos que se requieren, con el fin de diseñar y utilizar un módulo software como una unidad que puede ser distribuida de forma independiente, para ser utilizada de forma fiable en diferentes aplicaciones que son desarrolladas por terceros. Esta característica intrínseca de los componentes proporciona dos valores complementarios:

    - Desde el punto de vista económico permite generar productos que pueden ser comercializados independientemente.

- Desde el punto de vista logístico, constituye unidades de distribución que pueden ser desplegadas de forma flexible por una plataforma distribuida.

Conviene resaltar que junto a los beneficios que se derivan de la utilización de los componentes existen actualmente otros factores que inhiben su implantación y limitan la expansión del mercado. De todas estas condiciones, vamos a analizar las cuatro más relevantes:

- La disponibilidad de componentes: La principal dificultad que encuentran las empresas que tratan de implantar metodologías de diseño basadas en componentes es la dificultad de encontrar suministradores de componentes de su dominio con el estándar industrial requerido. Es posible, que esto sea una consecuencia de la inmadurez de la infraestructura del mercado, o de que éste, no haya alcanzado aun una masa crítica suficiente. En este momento, parece que la tecnología de componentes sólo triunfa en sectores verticales bien definidos, como ofimática, e-comercio, o interfaces de usuario (GUIs), en los que la disponibilidad de componentes está muy bien surtida. En otros sectores como control industrial, robótica, etc. su despegue es mucho mas lento, y además no presenta una naturaleza inter-empresarial marcada.

- La falta de estándares estables para las tecnologías de los componentes: Un requisito indispensable para conseguir la confianza de los diseñadores de software en la tecnología de componentes, y consolidar el mercado subyacente, es que el estándar en el que se basa la tecnología, permanezca estable a lo largo del tiempo. Actualmente las tecnologías de componentes no son estables como consecuencia de dos tendencias contrapuestas: por un lado las empresas requieren estabilidad para ampliar el mercado de componentes, y en contraposición, necesitan introducir diferencias en sus productos para conseguir ventajas competitivas. Solo las tecnologías de componentes soportadas por una única empresa como COM, COM+, DCOM, ActiveX y .net de Microsoft, y JavaBeans y EJB de Sun Microsystem han conseguido una estabilidad suficiente para desarrollar su mercado.

- La falta de componentes certificados y organizaciones con autoridad de certificación: La adopción de una nueva tecnología se lleva a cabo en dos fases: primero se trabaja en demostrar su viabilidad, pero inmediatamente después, se requiere trabajar para garantizar la calidad de los productos. Dada la opacidad intrínseca que presentan los componentes, la garantía de su calidad no puede ser avalada por el análisis del propio componente, sino que tiene que ser establecida por una autoridad de certificación reconocida.

- La falta de métodos y procesos de ingeniería para desarrollar sistemas con calidad: Actualmente está aceptado que el desarrollo de aplicaciones utilizando las tecnologías de componentes requiere un nuevo proceso de ingeniería, con nuevas fases y nuevos agentes. Una organización que desee adoptar una tecnología de componentes software tiene miedo de abordarla por dos razones: el costo que implica el cambio de la organización interna

que requiere la nueva ingeniería, y sobre todo en este momento, la falta de experiencia y modelos contrastados que deben ser introducidos.

Desde 1998 la tecnología de componentes ha tenido un crecimiento explosivo, que para los más moderados ha superado el 25% anual. Este crecimiento inicial ha sido generado principalmente por las empresas Microsoft y Sun Microsystem, que a través de sus tecnologías de componentes COM y EJB ha representado más del 80% del mercado de componentes disponibles. Las tecnologías de Microsoft y Sun Microsystem han sido desarrolladas siguiendo los intereses comerciales que tenían las empresas, y ambas tienen un sesgo característico. Las tecnologías COM, DCOM, +COM y .net son específicas de la plataforma Windows y se encuentran integradas en ella. Las tecnologías *JavaBeans* o EJB (*Enterprise JavaBeans*) sólo están disponibles para ser utilizadas en aplicaciones Java.

1.2 TECNOLOGÍAS DE COMPONENTES CCM. MODELO DE REFERENCIA.

1.2.1 Tecnología CCM origen como extensión de la tecnología CORBA.

En 1989 se fundó la OMG (*Object Management Group*) con el objetivo de fijar estándares para establecer un marco común de desarrollo de software que hiciera posible la reusabilidad, portabilidad, e interoperatividad de software basado en objetos sobre plataformas distribuidas heterogéneas. Se estableció la OMA (*Object Management Architecture*) como infraestructura conceptual, dentro de la cual se definió la especificación CORBA como un estándar avanzado de middleware orientado al desarrollo de aplicaciones distribuidas portables a cualquier plataforma distribuida, que pudiese operar en diferentes sistemas operativos, y con software desarrollado en diferentes lenguajes de programación. La última revisión es la 2.6, que fue aprobada en mayo de 2002.

Sin embargo, la OMG detectó a finales de los 90, limitaciones en CORBA en base a la aplicación de los nuevos paradigmas de programación basados en modelos MDA (*Model Driven Architecture*). Entre las limitaciones más importantes que se detectaron en el modelo de objetos que estaba definido se encontraban:

- Sólo se definía la funcionalidad de los objetos a través de interfaces declaradas como contratos entre clientes y servidores, siendo un requerimiento que los componentes que dependiesen de otros objetos deban conectarse explícitamente a ellos. Ésta conexión explícita conduce a implementaciones frágiles y no reusables.

- No se especificaba 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.

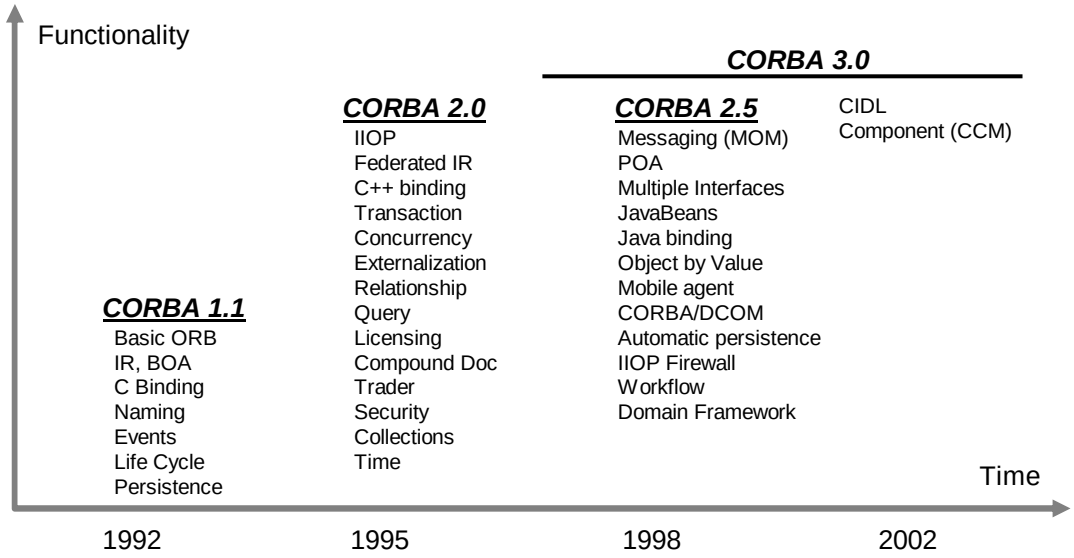

Figura 1.2: Evolución de la funcionalidad de CORBA.

Para solucionar estas limitaciones se creó un grupo con el fin de evolucionar hacia un modelo basado en componentes. En 2006 nace la nueva especificación de CORBA 3.0, en la cual se define una nueva tecnología de componentes denominada CCM (*Component CORBA Model*), que da soporte al concepto de componente como una extensión del modelo de objeto, y establece un estándar que define la implementación, el empaquetado, el ensamblado y el despliegue de las instancias de los componentes en aplicaciones distribuidas. Para un cliente, un componente CCM es una extensión de un objeto CORBA que engloba varios modelos de interacción a través de diferentes interfaces y conexiones. Para un servidor en cambio, es una unidad de implementación que puede ser instanciado de forma independiente, haciendo uso de la información estandarizada que lleva incluida.

Actualmente CCM es el único estándar para desarrollar aplicaciones basadas en componentes codificados en diferentes lenguajes de programación y con capacidad de ser desplegados e interoperar en diferentes plataformas. En su especificación se ha prestado atención particular en la inclusión del modelo de objetos de EJB, .NET, ActiveX y CORBA 2.

CCM ha desarrollado un modelo de despliegue que ofrece los recursos necesarios para el control de todos los aspectos de la arquitectura de la aplicación y la capacidad de configurar las características funcionales y no-funcionales de los componentes, sin tratar aspectos de bajo nivel relacionadas con el código.

1.2.2 Elementos básicos del modelo de referencia CCM

Un componente CCM es una entidad que ofrece un conjunto de puertos, a través de los cuales ofrecen servicios a los clientes y obtienen servicios de los servidores para proporcionar su funcionalidad. En CCM existen los siguientes tipos de puertos:

- Facetas (*facets*): Son puntos de conexión a través de los cuales se ofrece a los clientes los servicios implementados. Un componente puede tener asociado varias facetas y por cada faceta tendrá asociada una interfaz que describirá su funcionalidad.

- Receptáculos (*receptacles*): Son los puntos de conexión a través de los cuales el componente obtiene los servicios necesarios para conseguir su funcionalidad. Como en el caso de las facetas, cada receptáculo tiene asociado una interfaz que describe su funcionalidad.

- Fuente de evento (*event source*): Son puntos de conexión que definen un tipo específico de eventos que el componente puede generar, y que pueden ser recogidos de forma asíncrona por otros componentes, con el consecuente intercambio de información que esto produce entre los componentes.

- Receptor de evento (*event sink*): Son puntos de conexión a través de los cuales se puede recibir un tipo específico de eventos generado por un componente.

En un componente CCM también se pueden definir atributos, que son valores o características que forman parte del estado del componente, y que normalmente son utilizados para realizar su configuración. El acceso y modificación de los atributos de un componente se realiza mediante las operaciones *accessor* y *mutator* definidas en su interfaz.

Cada componente tiene definida una referencia a objeto que se denomina *equivalent interface*, y representa el elemento con el que se hace referencia al componente desde otros componentes de la misma aplicación. A través de ellas, los clientes del componente pueden conocer las facetas ofertadas por el componente y conectarse a los correspondientes puertos para acceder a la funcionalidad que oferta.

La relación entre el componente y las facetas que oferta se rige por las siguientes características:

- La implementación de las interfaces de las facetas que ofrece se encuentran encapsuladas dentro del componente, y forman parte del mismo, sin embargo ésta implementación es completamente opaca a los clientes.

- Los clientes pueden navegar desde cualquier interfaz correspondiente a una faceta a la interfaz equivalente del mismo y viceversa.

- Desde el cliente se puede determinar si dos referencias a facetas de un componente se corresponden con la misma instancia del mismo.

- El ciclo de vida de una faceta es la misma que la del componente al que pertenece.

En la siguiente figura se representa un componente CCM y todos sus tipos de puertos:

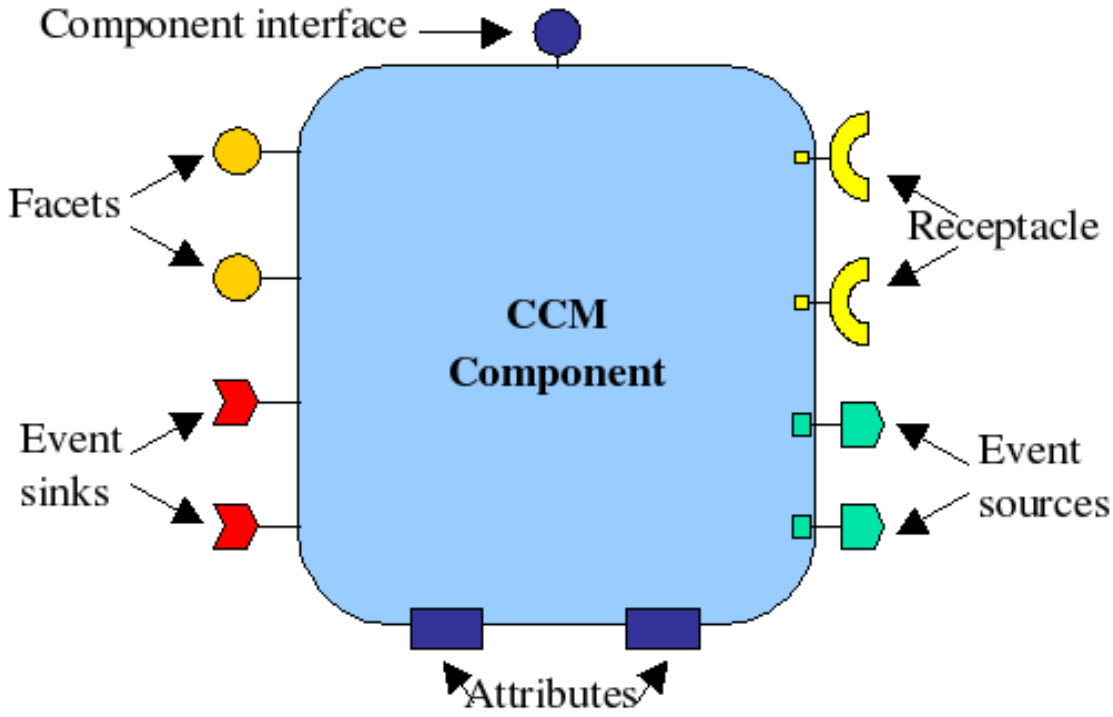

Figura 1.3: Esquema de los puertos de un componente CCM.

Los componentes poseen una clave (*primary key value*) a la cual están asociados. La funcionalidad de esta clave es distinguir las posibles diferentes instancias de un mismo componente y para obtener referencias a dicha instancia del mismo. Esta clave no forma parte del componente, sino que es un dato utilizado por el gestor del componente (*Home*).

La declaración de un gestor se realiza en la especificación IDL del componente. Para cada tipo de componente pueden definirse varios tipos de gestores, pero cada instancia del componente está únicamente asociada al gestor que lo creó, entre las características que gestiona el gestor están el ciclo de vida de las instancias y el establecimiento de asociaciones entre instancias y claves.

IDL es el lenguaje en el que se especifican los objetos CORBA, La descripción IDL se sitúa entre el cliente y la implementación de manera que sea opaca al mismo y de esta manera se consiga independencia frente al lenguaje de programación utilizado en la propia implementación. En nuestro caso, debido a la utilización de IDL para la especificación de componentes, hemos utilizado la versión IDL3 que extiende la especificación IDL para la utilización con componentes. Esta extensión se reduce básicamente a incluir nuevas palabras clave que permitan dar soporte a los nuevos conceptos introducidos con relación a la idea de componente.

1.3 GENERALIZACIÓN DE LA TECNOLOGÍA CCM

1.3.1 Problemas del uso de CORBA en sistemas empotrados mínimos

En la actualidad, las aplicaciones evolucionan creciendo en su complejidad, requiriendo nuevos requisitos no funcionales y necesitando ser ejecutada sobre sistemas distribuidos complejos y sistemas empotrados con recursos limitados. En particular, las tecnologías basadas en componentes que actualmente se encuentran en el mercado (EJB de Sun MicroSystems, COM+ y .NET de Microsoft o CORBA 3 de OMG) no son tecnologías apropiadas para soportar requisitos de tiempo real, y mucho menos, si las plataformas sobre las que se ejecutan disponen de recursos limitados como en el caso de las plataformas embebidas.

En el modelo de referencia CCM, se especifica a CORBA como el middleware de comunicaciones que debe utilizarse. En los últimos años se han realizado varios proyectos europeos MERCED [7], COMPARE [8] y FRESCOR [9], que desde diferentes puntos de vista buscan alternativas a CORBA menos pesadas, y que sean compatibles con sistemas embarcados y de perfil mínimo. En ellos se ha sustituido CORBA por otro middleware de comunicaciones menos pesado compatible con requisitos de tiempo real, pasando a formar una nueva extensión de CCM denominada Container Component Model.

1.3.2 Elementos básicos del modelo de referencia CCM generalizado

La idea básica del modelo de referencia generalizado es sustituir la comunicación basada en el middleware CORBA que propone OMG, por mecanismos de comunicación entre componentes que son apropiados a cada plataforma de ejecución que se utilice y que se definen como nuevos componentes que se denominan conectores.

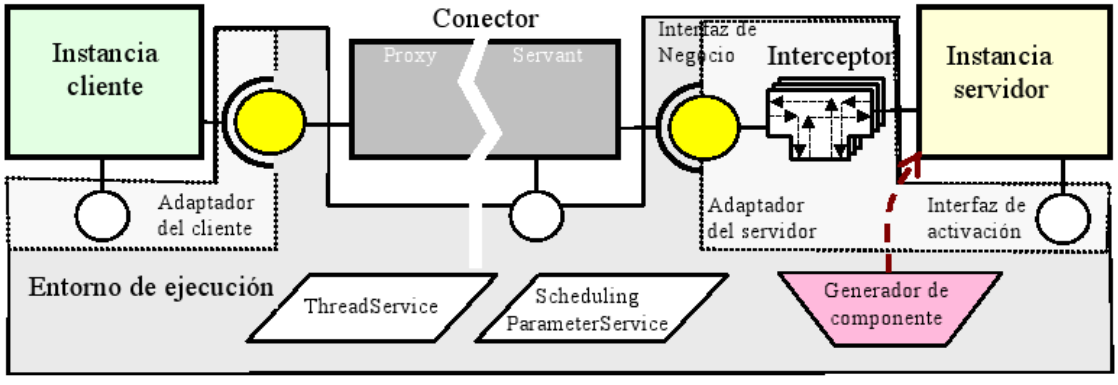

Figura 1.4: Esquema de los elementos básicos del modelo CCM generalizado.

En la figura 1.4, se muestran los elementos básicos que constituyen este modelo. Destaca como novedad la utilización contenedores como mecanismo de adaptación de los componentes al entorno de ejecución, de los conectores como el medio por el que los

componentes se comunican entre sí, y los interceptores que permiten independizar las características no funcionales del código de negocio.

La funcionalidad de los elementos más destacables:

- Adaptador (*container*): Representa la parte del código de un componente que proporciona el soporte de la plataforma de ejecución al código de negocio. Constituye la forma a través de la que se libera al diseñador del componente, de conocer los detalles de la tecnología y de los recursos de la plataforma de ejecución. Todos aquellos aspectos derivados de la tecnología o de la plataforma se incluyen en el adaptador y su código se genera mediante herramientas automáticas, permitiendo que el diseñador se dedique únicamente al desarrollo del código de negocio.

- Conector (*connector*): La función de este elemento es realizar la comunicación entre los puertos de los componentes. A través de él, en una conexión distribuida, un componente que se encuentra en un procesador, puede realizar las invocaciones requeridas a otro componente situado en otro procesador. Si ambos componentes se encontraran en el mismo procesador la conexión es local, y el conector se reduce a facilitar la referencia del servidor al cliente. Un conector es un tipo de componente mas, constituido por dos elementos: el *proxy* (que se instancia en el procesador del cliente) y el *servant* (que se instancia en el procesador del servidor), y que se comunican entre sí mediante un mecanismo propio que es adecuado a la plataforma distribuida en la que se ejecuta. La naturaleza de este conector depende en gran medida de la tecnología de componentes en que se implementa y de los recursos de comunicaciones de que dispone la plataforma de ejecución. Un conector comunica dos puertos (faceta y receptáculo) que se encuentran interconectados, y por tanto, están íntimamente relacionados con la interfaz que dichos puertos implementan.

- Interceptor (*interceptor*): Se trata de un mecanismo que permite incorporar los servicios del entorno en el proceso de invocación de una operación, de manera que se realicen determinadas acciones antes y después de la invocación de dicha operación. La introducción de este elemento es opcional y depende del plan de despliegue que instancia los componentes de la aplicación. Las actividades realizadas en el interceptor dependen de la tecnología de componentes utilizada, no obstante no deberán modificar la lógica de negocio, pudiendo por ejemplo realizar modificaciones de los recursos que el entorno proporcione para la ejecución de dicha operación.

- Interfaces de Activación: Los componentes se tratan ahora de elementos pasivos que sólo ofrecen operaciones a través de sus puertos, y a través de estas interfaces el entorno le proporciona los hilos de ejecución que requiere para su ejecución. Existen dos tipos de interfaces de activación: las denominadas *OneShotActivation* que se tratará de un hilo cuyo lanzamiento se realizará una vez en el proceso de lanzamiento de la aplicación; y las denominadas *PeriodicActivation*, que ejecutaran su operación de forma periódica según la configuración obtenida del componente sobre el hilo.

- Servicios de hilos de ejecución (*thread service*): Tiene como función crear y gestionar todos los hilos que ejecutan actividades en el componente. Algunos de estos son aplicados por el entorno para implementar respuestas asíncronas a la invocación de una determinada operación. Este servicio puede realizar la creación de los hilos bajo demanda, esto es se crearán según vayan siendo necesitados, o puede disponer de un grupo configurable de hilos que aplica a una u otra instancia de un componente en función de la aplicación.

1.4 ESPECIFICACIÓN DE DESPLIEGUE Y CONFIGURACIÓN DE OMG

Con objeto de desarrollar la especificación de despliegue y configuración de las aplicaciones basadas en componentes, la OMG mantiene un grupo de trabajo que se encarga de facilitar la interoperatividad entre componentes, de los recursos y servicios que las plataformas deben proporcionar, y de las herramientas que soportan su ciclo de desarrollo.

Desde el punto de vista del despliegue, un componente es un paquete que lleva asociado un modelo, esto es la información necesaria sobre el componente para que las herramientas automáticas puedan instanciar el componente en la plataforma, así como enlazar éste con otros componentes. Un componente posee tantos modelos como módulos de código compilado tenga.

Cada instancia creada ha de ser asignada al procesador que le corresponda de manera que una vez situada, pueda ser enlazada y ejecutada correctamente. Para realizar esta tarea, son necesarias herramientas que en base al modelo asociado a los módulos de código, gestionen de forma adecuada esta información para lograr el despliegue.

El proceso de despliegue de una aplicación basada en componentes comienza después de que el software haya sido desarrollado, empaquetado y publicado por el proveedor de software y adquirido por el propietario del software, que lo desplegará.

Las fases del proceso de despliegue según la especificación D&C son:

- Instalación: representa la tarea de importar los paquetes con el que se distribuye el código de los componentes y registrarlo en el repositorio del entorno de desarrollo.

- Configuración: En esta fase se asignan los valores apropiados a los atributos de configuración de las instancias de los componentes para su adaptación a los requerimientos de la aplicación.

- Planificación: Establece las instancias de componentes que se ensamblan en la aplicación y se asignan los procesadores del entorno de ejecución en los que deben instanciarse y ejecutarse. En esta fase se toman en consideración los requisitos de la aplicación, así como los recursos disponibles en la plataforma en la que se ejecutará.

- Preparación: Corresponde a las tareas de comprobación que se realizan en los procesadores de la plataforma de ejecución, con el fin de garantizar que estén disponibles los recursos necesarios para poder ejecutar la aplicación.

- Lanzamiento: En esta fase se instalan las instancias de los componentes en los procesadores, se realiza la interconexión entre las instancias, se reservan los recursos necesarios tanto en la red como en la plataforma y se inicia la ejecución.

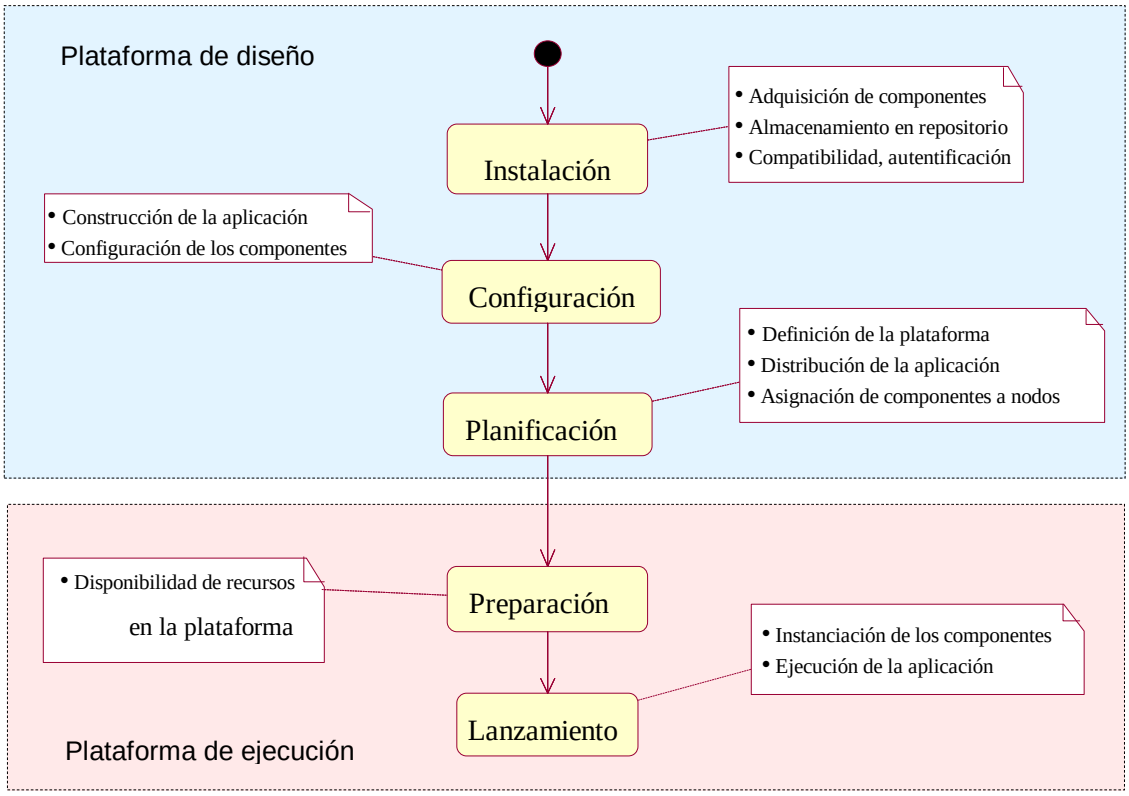

Figura 1.5: Fases del proceso de despliegue según D&C.

1.5 OBJETIVOS DEL PROYECTO

Actualmente el Grupo Computadores y Tiempo Real de la Universidad de Cantabria, está desarrollando dentro del proyecto europeo FRESCOR [9] y en colaboración con la empresa francesa THALES la tecnología RT-CCM que tiene como objeto el desarrollo de una tecnología de componentes CCM compatible con sistemas embarcados y de perfil mínimo en los que recursos disponibles está muy limitados, y en los que se presenta requisitos de tiempo real. Esta tecnología se está elaborando utilizando el lenguaje de programación C++ y plataformas RT-Linux.

Recientemente ha sido aprobada la nueva especificación Ada 2005, que abre la posibilidad de desarrollar con ella la tecnología de componentes. Ada es un lenguaje apropiado para el desarrollo de aplicaciones de tiempo real en sistemas embarcados, ya que desde el propio lenguaje se puede gestionar la concurrencia de los procesos, la

distribución entre los procesadores y la asignación del código de los elementos lógicos a la configuración hardware de la plataforma en que se ejecuta.

El objetivo de este proyecto ha sido explorar la potencialidad del lenguaje Ada 2005 para implementar la tecnología de componente RT-CCM que se está desarrollando en el proyecto de investigación. Con ello, se trata de crear una metodología de implementar la tecnología de componentes RT-CCM que hemos denominado Ada-CCM, que al basarse en un lenguaje de programación moderno que cubre todos los aspectos de la tecnología, da lugar a implementaciones mas sencillas y eficientes que otras basadas en múltiples recursos (lenguaje de programación, sistema operativo y middleware de comunicaciones).

Entre los objetivos concretos que se han llevado a cabo en este proyecto, se pueden destacar las siguientes:

- Se ha desarrollado una arquitectura de clases y objetos Ada que implementan el modelo de referencia (*framework*) de la tecnología de componentes RT-CCM. La arquitectura especifica los elementos que constituye un componente y para cada uno de ellos, se describe la funcionalidad que debe implementar, y los patrones de interacción que requieren. El criterio principal que ha guiado la especificación de la arquitectura ha sido facilitar la generación automática del código. La mayoría de las clases están relacionadas con los aspectos definidos en la tecnología y son comunes a todos los componentes. De estas clases se han definido un código parametrizado que sirve de base para la generación automática del código final de los componentes. En el caso de los componentes en que reside la funcionalidad de negocio, la arquitectura define su estructura y delimita los segmentos en los que el programador debe introducir el código de negocio.

- Se ha propuesto una extensión para Ada 2005, de las interfaces, componentes y tipos de datos que se formulan en el lenguaje IDL e IDL3. OMG tenía definida la correspondencia entre IDL y el lenguaje Ada'95. La introducción de la nueva versión de Ada ha requerido revisar esta correspondencia. Dado que Ada 2005 es un lenguaje en la que se han incorporado muchos de los conceptos modernos de la ingeniería software, la correspondencia que se ha introducido es más sencilla, eficiente y directa que la que hasta ahora estaba definida.

- Se ha desarrollado una herramienta para la generación automática del código Ada 2005 de los componentes que se implementan en la tecnología Ada-CCM. La herramienta procesa como entrada las especificaciones IDL de los componentes y genera el código Ada de los elementos que constituye el componente: Genera el código completo de los elementos del adaptador, y los esqueletos de los elementos que contienen el código de negocio. La herramienta se basa en disponer de un conjunto de plantillas parametrizadas de código, y en introducir elementos de las mismas en el código final del componente, de acuerdo con la información encontrada en los ficheros IDL de especificación.

- Se ha propuesto un modelo de comunicación entre componentes distribuidos entre diferentes nudos que se basa en el modelo ADA distribuido. Se ha definido un mecanismo de comunicación basado en las interfaces remotas (RCI Remote Call Interface) definido en el anexo de distribución de ADA. Así mismo se ha generado una herramienta que tomando como entrada el plan de despliegue de la aplicación genera el código de los conectores entre componentes que se especifican en él.

- Se ha utilizado como demostrador de la tecnología un ejemplo denominador JetFollower que ha sido desarrollado en el proyecto FRESCOR. El ejemplo es completo y permite describir muchas de las características que se implementan en la tecnología.

CAPÍTULO 2

PROCESO DE DESARROLLO DE UNA APLICACIÓN

2.1 PROCESO DE DESARROLLO DE UNA APLICACIÓN BASADA EN COMPONENTES

Como se muestra en la figura 2.1, el proceso de desarrollo de aplicaciones basadas en componentes tiene dos fases diferenciadas:

- Desarrollo de componentes. En esta fase se desarrollan los componentes, se diseña su código lógico y se empaquetan junto a la información complementaria que constituye su modelo. Esta fase se realiza en función de un dominio o área de aplicaciones, y antes de que se haya planteado su utilización en una aplicación concreta. El resultado de esta fase son componentes elaborados que se incluyen en un catálogo y que podrán ser utilizados en el desarrollo de futuras aplicaciones.

- Desarrollo de aplicaciones. En esta fase el diseñador de una nueva aplicación, escoge del catalogo los componentes que necesita y los ensambla de acuerdo con su modelos para conseguir la funcionalidad que tiene especificada. Este diseñador no necesita conocer el código de los componentes que utiliza, sino que los elige, compone y utiliza de acuerdo con la información complementaria que llevan asociada.

El proceso de desarrollo de aplicaciones basada en componentes es un proceso de diseño basado en modelos que requiere herramientas y entornos de desarrollo automatizados para facilita su gestión. Como se muestra en la siguiente figura, en el

proceso específico que se sigue para desarrollar componentes y aplicaciones basadas en ellos, intervienen diferentes agentes, que a veces son humanos y a veces son herramientas automáticas:

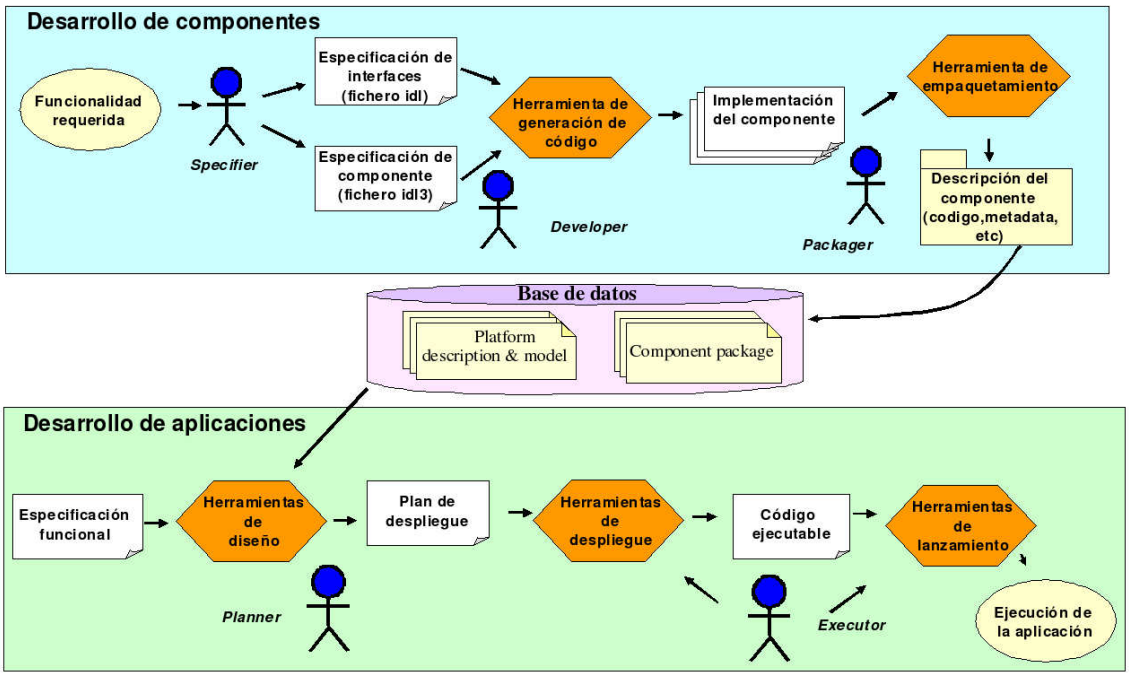

Figura 2.1: Fases del desarrollo en la tecnología Ada-CCM.

- El especificador (*specifier*), es un agente especializado cuya función es definir las funcionalidades (interfaces) y los componentes (asociaciones de funcionalidades que son útiles) dentro de un dominio de aplicaciones. Para ello, utiliza el lenguaje IDL que es abstracto y puede ser traducido de forma unívoca en cualquiera de los lenguajes de programación habituales (C, C++, Java, Ada, etc.).

- El implementador de componentes (*developer*), genera el código de los componentes concretos que van a ser utilizados. Habitualmente la función del implementador se realiza de forma combinada por herramientas automáticas y programadores humanos. Las primeras son herramientas generadoras automáticas que interpretan los ficheros IDLs que describen funcionalmente los componentes y todos aquellos aspectos que son propios de la tecnología que se está utilizando. El programador humano introduce el código de negocio que implementa la funcionalidad final del componente.

- El empaquetador (*packager)*, se encarga del empaquetar toda la información asociada a un componente, ya sea la relativa al código ejecutable o a la información complementaria que constituye su modelo. La función del empaquetamiento es agrupar y formatear toda la información, de acuerdo a una normativa estandarizada que facilite su posterior utilización por desarrolladores de aplicaciones y herramientas de gestión de los componentes.

- El diseñador de las aplicaciones (*planner*), inicia la fase del desarrollo de una aplicación. Haciendo uso de herramientas de diseño, explora la base de datos que contiene el catálogo de componentes y diseña la aplicación especificando, que componentes van a utilizarse, como deben configurarse, como deben ensamblarse y en que plataforma va a ejecutarse. El plan de despliegue se genera usando la normativa definida en la especificación D&C[11] de OMG.

- La herramienta de despliegue, procesa la información contenida en el plan de despliegue: selecciona el código de los componentes adecuado a la plataforma, genera el código del entorno necesario para dar soporte a la ejecución de los componentes en cada nodo y elabora los requisitos a negociar (contratos) con la plataforma para su procesamiento y comunicación.

- La herramienta de lanzamiento realiza la última fase del proceso: transporta e instancia el código ejecutable de todos los componentes en los procesadores que se hayan asignado en el plan de despliegue e iniciará la ejecución de la aplicación.

2.2 DESCRIPCIÓN IDL DE UN COMPONENTE: CORRESPONDENCIA ENTRE TIPOS IDL Y ADA.

2.2.1 Descripción de una interfaz y de un componente en IDL

Como se ha comentado en anteriores apartados, se utiliza IDL para la especificación de las interfaces, y así mismo utilizaremos su extensión IDL3 para la especificación de los componentes.

Una interfaz representa la descripción funcional de un servicio, formulado con independencia del componente que lo implementa. Si dos componentes diferentes ofertan una misma interfaz, eso significa que ofrecen un mismo tipo de servicio, por lo que pueden ser utilizados de forma intercambiable por cualquier cliente que lo requiera. Los elementos centrales de la descripción de una interfaz constituyen la descripción del conjunto de operaciones que un cliente puede invocar sobre el servicio. En las interfaces, sólo se especifican las características que se requieren para su invocación (*signature*), esto es, su nombre, el tipo del valor que retorna y los parámetros que deben ser pasados cuando se invoca, así como su tipos. El algoritmo con que se implementa cada operación no es parte de la interfaz, sino que será específico del componente que la implementa. Además de las operaciones, una interfaz puede describir atributos, declaración de excepciones, constantes, tipos de datos definidos por el usuario, tipos simples primitivos, tipos enumerados, tipos estructura, tipos array, etc. Las interfaces no tienen calificadores de accesibilidad, ya que todos los elementos definidos en ella, tienen acceso público para cualquier cliente que necesite utilizarlos.

En IDL la herencia entre interfaces se introduce añadiendo detrás del interfaz dos puntos y el nombre del interfaz base. La herencia múltiple está permitida separándose los

identificadores de las clases base por comas. Debe aclararse que la herencia de interfaces no implica de ninguna manera la herencia de implementaciones, que en caso de haberla se definiría en la fase de implementación del componente.

A continuación se muestra un ejemplo sencillo de una descripción IDL de una interfaz:

```
interface I_Control{
     struct ControlParamSet{
          float sensorGain;
          float actuatorGain;
          float integralGain;
          float derivativeGain;
          float integralGain;
          float controlPeriod;
          unsigned short servoOutputChannel;
          unsigned short servoOutputDevice;
          unsigned short actuatorInputChannel;
          unsigned short actuatorInputDevice;
     };
     void setControlParamSet(in ControlParamSet paramSet,in unsigned short servo);
     struct TrajectoryParamSet{
          float maxVelocity;
          float positionRhreshold;
     };
     void setTrajectoryParamSet(in TrajectoryParamSet paramSet,in unsigned short
servo);
     void putControlPeriod(in float thePeriod);
     typedef sequence<float> Position;
     void goToPosition(in Position newPosition);
     Position getCurrentOisition();
     unsigned short ao_NumberOfLine();
     void startLogging();
     void cancelLogging();

};
```

Se puede observar la declaración de tres tipos de datos: “ControlParameterSet” y “TrajectoryParmSet” que son de tipo *struct*, y “Position” que es de tipo *sequence*, y así mismo se declaran 8 operaciones *setControlParamSet*, *setTrajectoryParamSet*, *putControlPeriod*, etc.

IDL es un lenguaje de especificación de objetos, y sólo permite describir interfaces. Para describir componentes, se utiliza la extensión IDL3 (o CIDL), que incorpora palabras clave que permiten describir los elementos que se definen en el modelo de referencia CCM. Entre estas, destacan las palabras clave *component* para declarar los componentes, *provides* para declarar facetas que ofrece un componente y *uses* para declarar los receptáculos que requiere.

Los gestores de un componente se declaran mediante la palabra clave *home*, definiendo el componente al que gestionan mediante la orden *manages*. Dentro del cuerpo del gestor podemos encontrar definiciones de atributos y tipos, al igual que en el caso de la declaración de las interfaces. Los constructores (*factory*) de un gestor con los que se instancian los componentes, y las operaciones de localización de las instancias (*finder*), se declaran mediante las palabras clave *factory* y *finder* respectivamente.

A continuación se muestra un ejemplo de declaración con IDL3 del componente "GralComponent":

```
include "Interface_A.idl"
include "Interface_B.idl"
include "Interface_X.idl"
include "Interface_Y.idl"
component GralComponent : RTComponent
{

        typedef short Attr_1_Type;
        typedef short Attr_2_Type;

        provides Interface_A thePort_A_1;
        provides Interface_A thePort_A_2;
        provides Interface_B thePort_B_1;

        // Environment interfaces
        provides OneShotActivation theOSAPort_1;
        provides OneShotActivation theOSAPort_2;
        provides PeriodicActivation thePAPort_1;
        provides PeriodicActivation thePAPort_2;

        uses Interface_X thePort_X_1;
        uses Interface_X thePort_X_2;
        uses multiple Interface_Y thePort_Y_1;

        attribute Attr_1_Type attr_1;
        attribute Attr_2_Type attr_2;
};

home GralComponent_home manages GralComponent
{
      factory theFactory;
};
```

En la declaración se especifica que el componente ofrece 3 facetas, dos de ellas ("thePort_A_1" y "thePort_A_2") implementan la interfaz "Interface_A" y otra ("thePort_B_1") implementa la interfaz "Interface_B". El componente tiene tres receptáculos: "thePort_X_1" y "thePort_X_2" asociados a la interfaz "Interface_X", y el receptáculo "thePort_Y_1" que implementa la interfaz "Interface_Y". En este componente se declaran dos atributos "attr_1" y "attr_2" que pueden representar parámetros de configuración del componente. El gestor del componente se denomina "GralComponent_home" y tiene declarado un constructor denominado "theFactory".

2.2.2 Correspondencia entre estructuras IDL y Ada

Por cada uno de los posibles tipos encontrados en la descripción IDL, tendremos implementaciones Ada correspondientes, basándonos en el *Ada Lenguage Mapping Specification* [12].

A continuación se muestra en la siguiente tabla la correspondencia en ADA de algunos ejemplos de tipos que pueden ser definidos en IDL:

Tipo definido en IDL	Correspondencia en ADA
`enum Color {Red, Green, Blue};`	`type Color is (Red, Green, Blue);`

`struct Example {`
`  long member1, member2;`
`  boolean member3[4][8];`
`};`
	`type Member3_Array is array(0..3, 0..7) of CORBA.Boolean;`
	`type Example is record`
	`  Member1: CORBA.Long;`
	`  Member2: CORBA.Long;`
	`  Member3: Member3_Array;`
	`end record;`

`typedef string Name;`	`type Name is new CORBA.String;`
`const double Pi = 3.1415926535;` `const short Line_Buffer_Length = 80;`	`Pi : constant CORBA.Double := 3.1415926535;` `Line_Buffer_Length : constant CORBA.Short := 80;`
`exception null_exception{};`	`Null_Exception: exception;`
`typedef fixed<8,2> Megabucks [3];`	`type Fixed_8_2 is delta 0.01 digits 8;` `type Megabucks is array(Integer range 0 .. 2) of Fixed_8_2;`

Para la implementación de determinados tipos, se han utilizado las extensiones que se han incluido en Ada 2005, como por ejemplo en el caso del tipo sequence cuya correspondencia se muestra a continuación:

Tipo definido en IDL	Correspondencia en ADA 2005

`typedef sequence<sequenceType> SequenceName;`	`package SecuenceName_pkg is new`
	`  Ada.Containers.Vectors(`
	`Natural,SequenceType);`
	`type SecuenceName is new`
	`  SecuenceName_pkg.Vector`
	`with null record;`

Lo más relevante de la extensión de la correspondencia de IDL con los nuevos tipos introducidos en Ada 2005, es la correspondencia entre los conceptos de interfaz. En la siguiente tabla se muestra a través de un ejemplo esta correspondencia:

Interfaz definida en IDL	Correspondencia en ADA 2005

`interface I_Name{`
`  void operationName(in float Arg1);`
`};`
	`package I_Name is`
	`  type Face is interface;`
	`  type Ref is access all Face'Class;`
	`  procedure operationName(`
	`  Self: access Face;`
	`  Arg1:in float);`
	`  is abstract;`
	`end i_Name;`

2.3 ESTRUCTURA GENÉRICA DE LOS COMPONENTES

El aspecto central de la tecnología Ada-CCM es facilitar la generación automática de código de todos los elementos que sea posible y de esta manera, centrar el esfuerzo del implementador en el diseño del código de negocio, liberándolo de conocer detalles de la tecnología, del entorno o las comunicaciones entre componentes. Este objetivo ha guiado el diseño de la estructura de los componentes que se ha planteado.

Para describir la estructura de los componentes en la tecnología Ada-CCM tal como la ve el implementador de un componente, se utiliza el componente genérico que se muestra en la figura 2.2, que ofrece los elementos mas relevantes de un componente: facetas, receptáculos, atributos de configuración y activaciones PeriodicActivation y OneShotActivation.

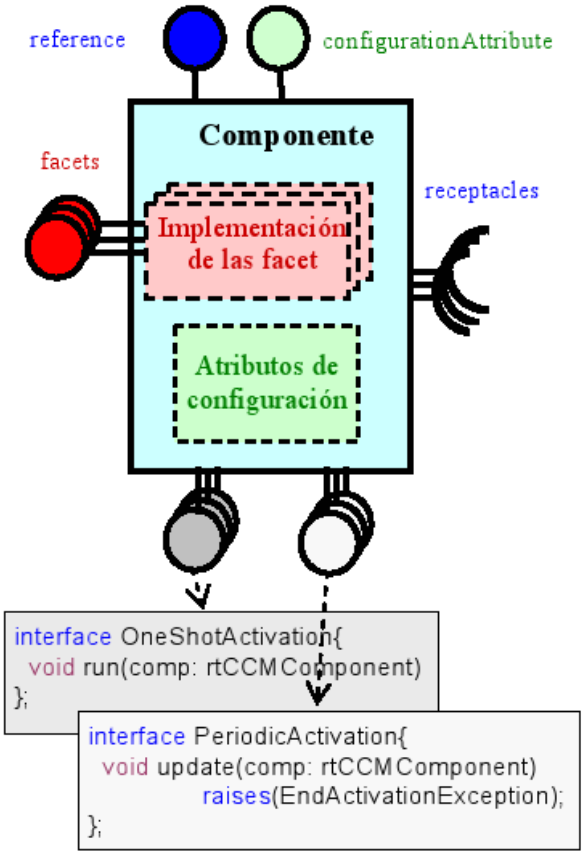

Figura 2.2: Estructura genérica de un componente.

El componente se concibe como un contenedor que agrupa los siguientes elementos:

- Estado del componente: Representa la información compartida por todos los elementos del componente. Es definido como una estructura independiente para poder ser accedida desde todos los demás elementos sin que se generen referencias cíclicas. El estado contiene:
  - Los atributos de configuración del componente.
  - Las variables internas del componente que representan el estado instantáneo del componente.
  - Las referencias a los receptáculos, a través de las que los diferentes elementos del componente invocan los servicios de otros componentes asociados a él.

- Los bloques de información asociados a las tareas correspondientes a los puertos de activación.

El código de negocio del componente se incluye dentro de su implementación. Deberá implementar las siguientes interfaces:

- Interfaces funcionales (business interfaces): contienen el conjunto de operaciones que implementan un servicio funcional del componente (facetas). Incluye los accessor y mutator relativos a los atributos declarados en la implementación de las facetas.

- Interfaces de activación (Activation interfaces) contienen las operaciones que los threads del entorno ejecutarán ya sea periódicamente (operación *update* de una *PeriodicInterface*) o una sola vez (operación *run* de *OneShotActivation*)

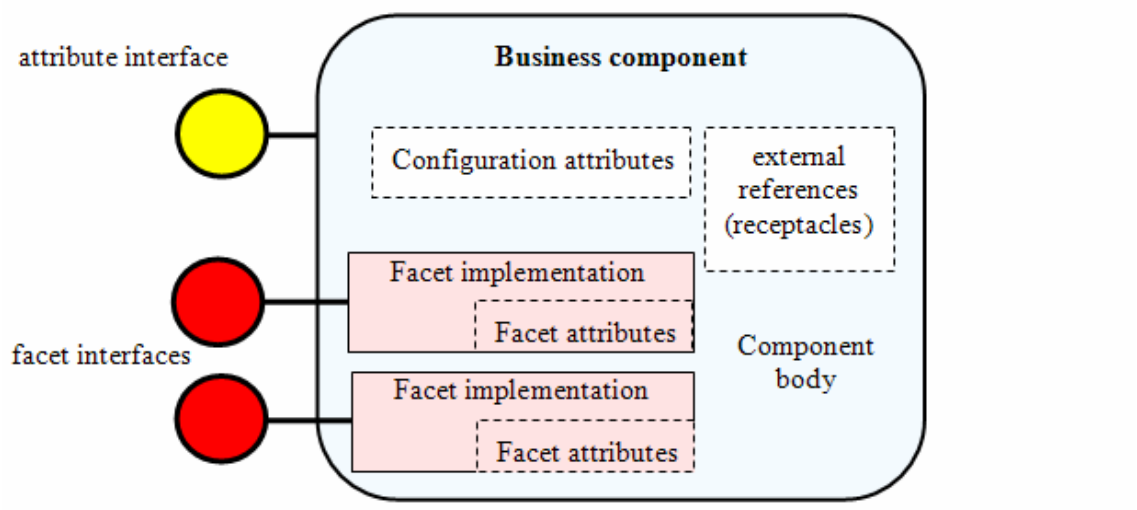

Figura 2.3: Detalle de la estructura del código de negocio en el componente.

Las Interfaces de atributos (*attribute interface*), serán generadas en cambio, de manera automática por la herramienta de generación. La implementación generada contendrá las funciones de lectura de los atributos públicos de configuración (accessor) y los procedimientos de escritura de los atributos públicos que no sean constantes (mutator).

2.4 ELEMENTOS DEL CONTENEDOR DE UN COMPONENTE

El contenedor de un componente está constituido por un conjunto de recursos que constituyen el soporte del componente desde el entorno y la conexión con otros componentes de la aplicación. Representa el adaptador del componente al entorno de ejecución en que se instancia. Su código será generado de forma completa por las herramientas de generación del código.

En esta apartado se describe la estructura del contenedor de un componente. Se basa en la clase que hemos denominado *Wrapper*, encargada de adaptar el componente a

la plataforma en que se ejecuta, haciendo que este ofrezca hacia el exterior la interfaz equivalente del componente, que es la que emplean para acceder a él los clientes que usan sus servicios.

En el *Wrapper se* encuentran:

- La referencia al contexto del componente. El contexto del componente, es el lugar donde se almacenan las referencias a las implementaciones de las interfaces requeridas por los receptáculos, las cuales son establecidas a través de la conexión en la fase de despliegue de los mismos a otro componente por medio de los conectores.

- El ejecutor del componente, tal y como se muestra en la siguiente figura. Posee visibilidad sobre el contexto, y representa la implementación real del componente, tal como se indica en la especificación CCM. Una operación recibida a través de la interfaz equivalente del componente será tomada por el ejecutor cuya implementación podrá utilizar los receptáculos requeridos debido a la visibilidad que posee sobre el contexto.

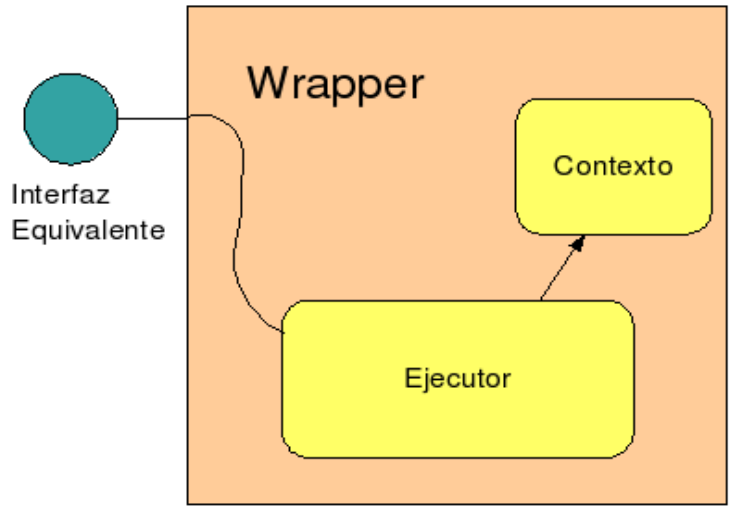

Figura 2.4: Estructura del Wrapper.

- El gestor del componente (*Home*), cuya función principal será la de crear instancias de los componentes. El *Home* representa una clase que podrá ser accedida por las herramientas de despliegue y por otros componentes. Cada *Home* dispone de operaciones para acceder a las instancias del componente que han sido creadas por él. Estas operaciones retornan referencias a la interfaz del componente (*equivalent interface*), mediante la cual se puede obtener la referencia de cualquier otra interfaz del componente.

2.5 FASES DEL PROCESO DE DESARROLLO DE UNA APLICACIÓN BASADA EN COMPONENTES

El proceso de desarrollo de una aplicación basada en componentes utilizando la tecnología Ada-CCM se divide en tres fases.

2.5.1 Fase de diseño de la aplicación

El agente que interviene en la realización de esta fase es el diseñador de la aplicación. Concibe un componente como un elemento con una descripción externa completa y bien definida. Para la realización de su tarea necesitará de la siguiente información:

- La especificación funcional: vendrá dada por la descripción IDL del componente, en ella se describirán los atributos globales de configuración y la descripción de la funcionalidad de los gestores asociados al componente. Por otra parte se expresará la funcionalidad de las facetas que ofrece mediante la propia especificación IDL de las interfaces que implementa y la funcionalidad de los receptáculos requeridos también descritos por las especificaciones IDL de las interfaces correspondientes.

- La descripción de la plataforma: incluye el conjunto de recursos de los elementos que la componen (nodos, sistema operativos, redes de comunicación, etc.).

- La especificación de tiempo real: describirá el comportamiento temporal que tendrá la aplicación, con el fin de poder realizar la ejecución de la misma con los tiempos necesarios para realizar las acciones de manera acotada temporalmente. Requisito indispensable para la ejecución de aplicaciones de tiempo real.

- La información de la base de datos: contendrá toda la información asociada a los componentes disponibles, ya sean, como consecuencia de nuevos desarrollos o de adquisiciones de otros desarrollados por terceros.

Con toda esta información, el agente diseña la aplicación como composición de componentes interconectados entre sí a través de sus facetas y receptáculos. Si no existen en la base de datos componentes que implementen alguna de las funcionalidades requeridas por la aplicación, será necesario desarrollarlo para esta aplicación.

El diseñador cuenta con herramientas de diseño que automatizan este proceso, de manera que como resultado del mismo se genera un fichero con el plan de despliegue de la aplicación, formulado de acuerdo a la especificación D&C. En dicho fichero se describe la aplicación, y contiene la siguiente información:

- Instancias de los componentes que intervienen en la aplicación.

- Localización de los nodos en los que se ejecutarán las instancias de los componentes.
- Asignación de los valores de partida de los parámetros de configuración que se asignan a cada instancia.
- Conexiones a realizar entre los puertos de instancias que lo requieran para realizar su funcionamiento.

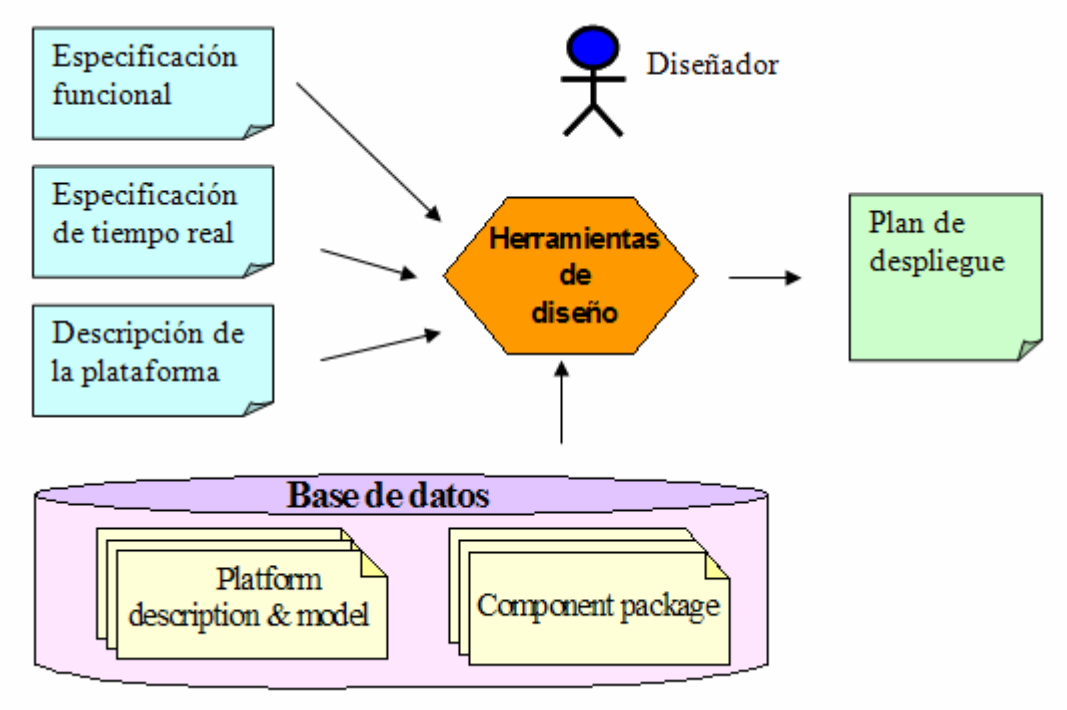

Figura 2.5: Fase de diseño de una aplicación.

A esta información, que es la que se define en el estándar D&C, es necesario añadir información específica de la tecnología Ada-CCM:

- A cada instancia declarada en el plan de despliegue, se le han de asignar propiedades de configuración especiales que permitan configurar los puertos de activación declarados en el componente, asignando valores a los parámetros de planificación con los que debe ejecutar la tarea que el entorno ofrezca para ejecutarlos y valor para el periodo de ejecución en el caso de una activación periódica.
- Por cada instancia, se han de configurar el conjunto de interceptores que se requieren para la gestión de los parámetros de planificación con los que se ejecutan las invocaciones recibidas en las facetas del componente

2.5.2 Fase de despliegue de una aplicación

En la figura siguiente se muestra los elementos que intervienen en la fase de despliegue de una aplicación.

El agente de despliegue (*planner*) a través de la herramienta de despliegue procesa el plan de despliegue de la aplicación, llevando a cabo las siguientes tareas:

- Toma de la base de datos (repositorio) el código de aquellos componentes que van a ser empleados en la aplicación.
- Genera el código de los conectores, que representan el mecanismo de comunicación entre componentes.

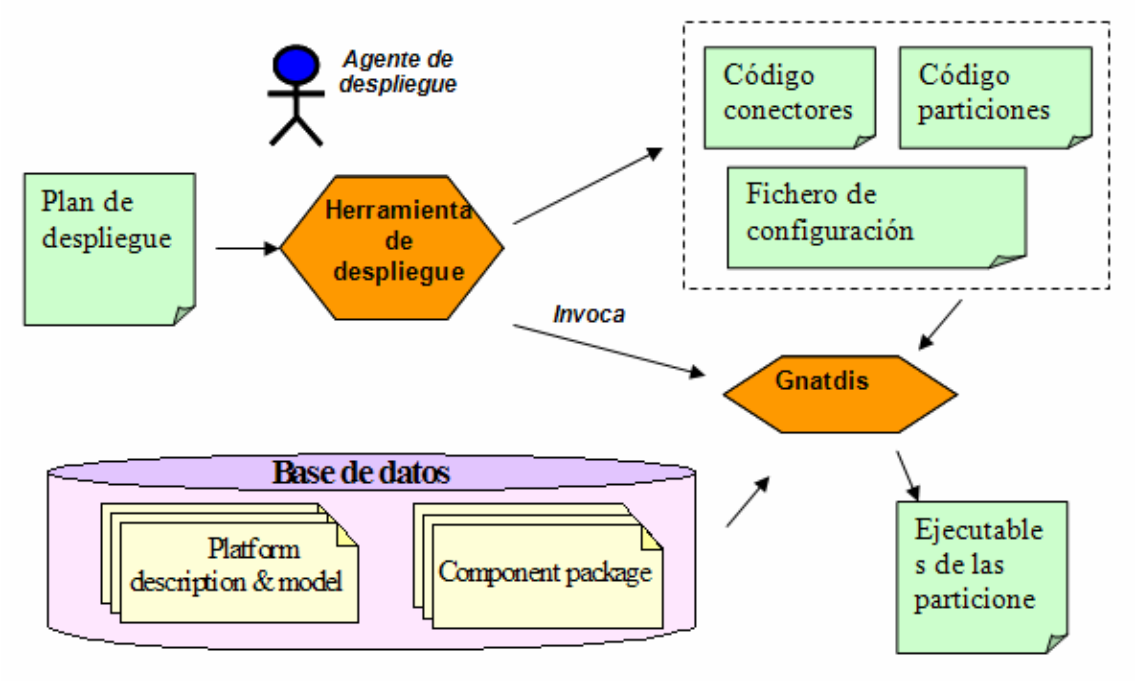

Figura 2.6: Fase de despliegue de una aplicación.

En el diagrama de actividad de la figura 2.7, se muestra un diagrama resumido de cómo se lleva a cabo esta generación. Por cada conexión entre componentes se evalúa si el código del conector asociado a dicho puerto ya ha sido generado previamente, en caso negativo es necesario generar su código para lo cual se utiliza la misma herramienta de generación de código empleada en el caso de un componente, la cual describiremos en el capítulo dedicado a la herramienta de generación de código.

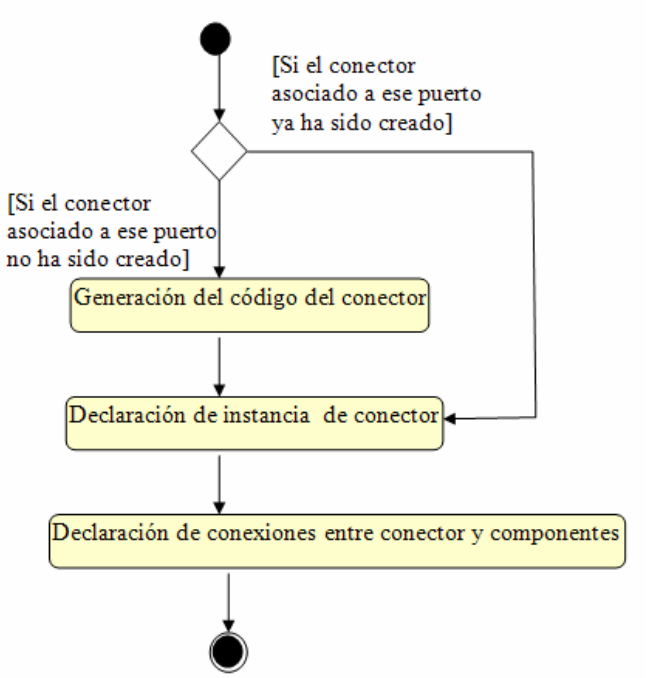

Figura 2.7: Generación de los conectores (Diagrama simplificado).

Una vez generado su código, se declarará la correspondiente instancia del conector, necesario ya que un conector no es más que un componente con un código de negocio especial generado automáticamente, y las conexiones de dicha instancia con los correspondientes componentes que conecta. Como se dijo, la naturaleza de los conectores depende en gran medida de la tecnología de componentes que lo desarrolle. En el caso de la tecnología Ada-CCM, hemos utilizado los mecanismos que nos proporciona GLADE para la comunicación de aplicaciones basadas en Ada. Veremos en el capítulo referido a los conectores una explicación con detalle de cómo se realiza esta comunicación basada en GLADE.

- Genera el fichero de configuración de GLADE, para lo cual toma como entrada tanto la información obtenida en el plan de despliegue, así como de la descripción de la plataforma. Esta información la ampliaremos en el capítulo dedicado a los conectores.

- Genera el código de los procedimientos principales de cada una de las particiones o nodos en que se haya de ejecutar la aplicación. El método principal de cada una de las particiones llevará a cabo la instanciación, configuración, conexión y activación de cada una de las instancias de componentes y conectores que estén instalados en dicho nodo.

Una vez que se tiene todo el código de la aplicación (componentes, conectores y métodos principales de las particiones), la herramienta de despliegue ejecutará la herramienta *gnatdist* sobre el fichero de configuración GLADE generado, obteniendo como resultado la generación de los ejecutables de cada una de las particiones.

2.5.3 Fase de lanzamiento de la aplicación

Figura 2.8: Fase de lanzamiento de la aplicación.

Esta última fase en el desarrollo de la aplicación es llevada a cabo por el agente de lanzamiento. Consiste básicamente en el lanzamiento de la aplicación, para ello una herramienta automática se encargará de trasladar los ejecutables de las particiones a los nodos correspondientes y lanzarlas, con lo cual se ejecutará la aplicación.

CAPÍTULO 3

ESTRUCTURA DE LOS CONTENEDORES DE COMPONENTE

NOTA: Durante este capítulo utilizaré para realizar los ejemplos y diagramas de clases el componente GralComponent cuya especificación IDL se mostró en el capitulo 2.2.1.

3.1 ESTRUCTURA DEL EJECUTOR

En el ejecutor del componente se definen los skeletons (clases raíces abstractas) y las interfaces que sirven de base para desarrollar las implementaciones de los componentes y de los home del componente.

La definición del ejecutor de un componente viene impuesta en el estándar CCM, y dado que Ada-CCM es una tecnología basada en dicho estándar, deberá ceñirse a dichas normas. CCM establece la creación de una serie de clases e interfaces a partir de la especificación IDL de un componente, que o bien automáticamente o bien por el usuario, deberán ser implementadas para desarrollar la funcionalidad del componente y el entorno.

Según CCM, las interfaces pueden ser de 2 tipos:

- Internal interfaces: son interfaces que ofrece el entorno (container), por tanto generadas automáticamente, y que son utilizadas por los componentes para desarrollar su funcionalidad.

- Callback interfaces: Son interfaces que deben implementar los componentes, serán invocadas por el contenedor para gestionar los componentes. Deben ser implementadas por el usuario (aunque parte de su código puede ser generado también de forma automática).

En esta tecnología, el conjunto de interfaces que, siguiendo la normativa CCM, forman el ejecutor de un componente se agrupan en un paquete Ada al que se ha denominado *[NombreComponente]_Exec*. Todas las interfaces que se generan en este paquete sirven de clases raíces para las clases que realmente van a implementar los componentes.

Haciendo uso del concepto de interfaz que proporciona la nueva especificación Ada 2005, en este paquete definiremos en primer lugar, una serie de interfaces locales, denominadas *facet executor interfaces* que tendrán como nombre *CCM_[NombreInterfaz]* y heredarán de las correspondientes interfaces de las facetas. No será necesaria su definición en el caso de las interfaces de activación pues serán gestionadas de otro modo por el entorno, al no tratarse de facetas que se ofrecen hacia el exterior.

En este paquete también se definirá, siguiendo la normativa CCM, una interfaz local de tipo *callback* denominada *monolithic executor interface*. El nombre que tendrá esta interfaz será CCM_*[NombreComponente]*. En el caso de que el componente herede de otro componente, esta interfaz heredará de la correspondiente interfaz CCM_*ComponentePadre*, en caso contrario heredará de la interfaz *EnterpriseComponent*, definida en el estándar CCM.

Como consecuencia de la herencia de *EnterpriseComponent* esta interfaz implementa la operación *set_session_context*, a través de la cual el contenedor le entregará al ejecutor del componente una referencia al contexto, mediante el que el componente puede tener acceso a la información acerca de las conexiones que se han realizado en el componente (esta clase se explica con mas detalle a continuación).

Además de las operaciones heredadas, ésta interfaz deberá incluir:

- Operaciones de lectura (*accesor*) y escritura (*mutator*) de cada uno de los atributos definidos en el componente: *get_[NombreAtributo]* y *set_[NombreAtributo]* respectivamente.

- Una operación adicional por cada faceta, denominada *get_[NombreFaceta]*, que posee el componente para poder acceder a ellas y que devuelven una referencia a la *facet ejecutor interface* correspondiente.

- Un método *configuration_complete* que será invocado por el contenedor para finalizar la configuración del componente. Esta operación no ha sido impuesta por CCM, se trata de una estrategia de implementación.

A continuación se muestra el diagrama UML de la interfaz *monolithic executor interface* así como de las facet executor interfaces para el caso del componente GralComponent, que en su especificación ofrece tres facetas al exterior, y posee dos atributos. En la interfaz *CCM_GralComponent* se observan las operaciones que se añaden como consecuencia de ello.

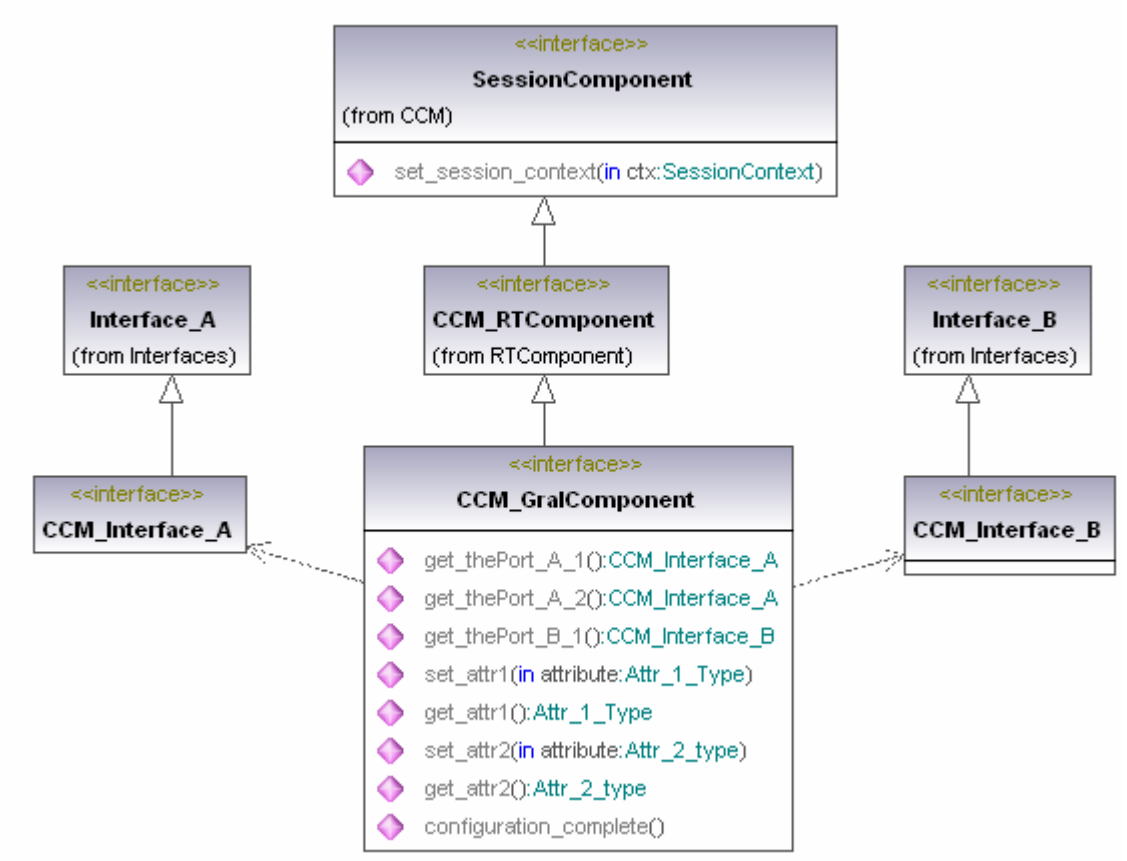

Figura 3.1: Diagrama UML de la *monolithic executor interface*.

Dentro del paquete del ejecutor se definirá también otra interfaz, en este caso de tipo *internal*, que será la interfaz de contexto del componente. Tendrá por nombre *CCM_[NombreComponente]_Context* y, al igual que en el caso anterior, heredará de la correspondiente interfaz *CCM_[ComponentePadre]_Context* en caso de que el componente herede de otro componente, o de la interfaz *CCMContext*, en caso contrario. Esta interfaz ofrecerá operaciones a través de las cuales un componente podrá obtener las referencias a los componentes que se encuentren conectados a sus receptáculos. Las operaciones serán del tipo *get_connection_[NombreReceptáculo]* y devolverán la referencia a un objeto del tipo de la interfaz esperada en el receptáculo si hay algún objeto conectado a él.

A continuación se muestra el diagrama UML de la interfaz de contexto del mismo componente GralComponent que hemos tomado como ejemplo. Cabe destacar que en la especificación del mismo se indica que posee tres receptáculos, dos de ellos receptáculos simples, correspondientes a referencias a implementaciones de interfaces de tipo *Interfaz_X* y otro receptáculo múltiple del tipo *Interfaz_Y*.

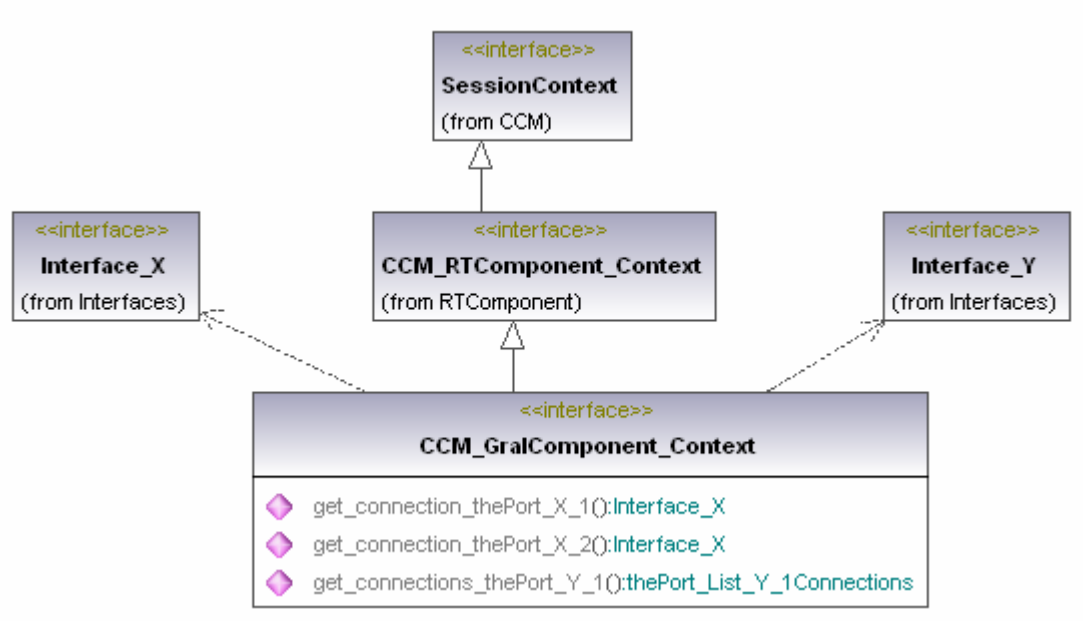

Figura 3.2: Diagrama UML de la interfaz de contexto del paquete del ejecutor.

Finalmente, se definirán tres interfaces correspondientes al gestor, o home, del componente. Todas ellas serán del tipo Callback.

- La primera de las interfaces se denomina *Home Explicit Executor Interface*, y en nuestra tecnología definiremos una interfaz Ada 2005 de nombre *CCM_[NombreHome]Explicit*, que contendrá todas las operaciones definidas explícitamente para el home (factorys y finders para la creación de instancias del componente), así como los accesor y mutator correspondientes a los atributos definidos explícitamente en el gestor. De la misma forma que antes, heredará de la correspondiente interfaz *CCM_[HomePadre]Explicit* si el componente tiene un home raíz o de la interfaz primitiva *HomeExecutorBase en caso contrario*.

- La segunda de las interfaces del home será la *Home Implicit Executor Interface*, que denominaremos *CCM_[NombreHome]Implicit* y que ofrecerá la operación *create()*, que será invocada por el contenedor para generar una nueva instancia del componente.

- Por último estará la *Home Main Executor Interface*, a la que hemos llamado CCM*_[NombreHome]* que implementará las dos anteriores.

A continuación se muestra el diagrama UML de las clases correspondientes al gestor del componente GralComponent, definido en el ejecutor del mismo:

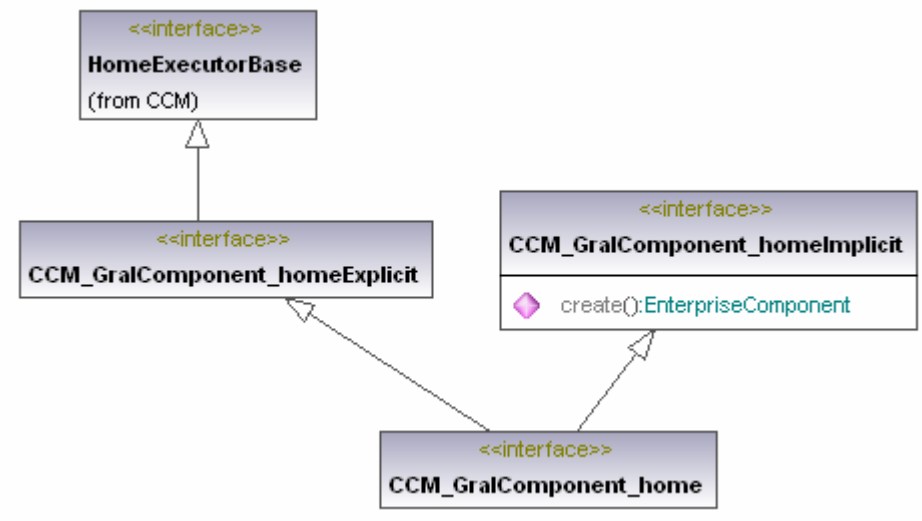

Figura 3.3: Diagrama UML de las clases correspondientes al Home del ejecutor.

3.2 ESTRUCTURA DEL WRAPPER

Como habíamos introducido en el capítulo 2.4, el *wrapper* representa al adaptador del componente a la tecnología. Requiere la definición de tres clases, que se agrupan en un paquete Ada denominado *[NombreComponente]_wrapper*:

- La clase que encapsula al componente, denominada *[NombreComponente]_Wrapper*, haciendo que éste ofrezca hacia el exterior la interfaz equivalente del componente, que será usada por los componentes clientes para acceder a él (y por tanto, a las implementaciones de sus facetas).

- Una clase que representa la implementación del contexto, denominada *[NombreComponente]_Context* , donde se almacenarán las referencias a los receptáculos requeridos.

- La clase que implementa la interfaz equivalente del home del componente, *[NombreComponente]_Home_Wrapper*, que será empleada por los componentes clientes para realizar la instanciación de componentes.

Todo el código de estas clases se generará totalmente de forma automática por la herramienta de generación de código que comentaremos en el capítulo 5.

La clase *[NombreComponente]_Wrapper*, nos ofrece la mencionada interfaz equivalente del componente, esto es, implementa la interfaz *CCMObject* definida en el estándar CCM. Para implementarlo hacemos que además herede de la clase *Object* que

hemos definido en nuestra tecnología y que nos permite hacer que el componente además ofrezca un método *get_component*, que devuelve un objeto del tipo Object y que es requerido por la normativa CCM.

La implementación de la interfaz *CCMObject* hará que esta clase implemente una serie de operaciones, entre las que destacan:

- configuration_complete: será invocada por el entorno para verificar la correcta configuración del componente durante la fase de instanciación de los componentes.

- provide_facet: Es una operación ofrecida al exterior del componente, mediante la cual un cliente puede obtener una referencia a la implementación de la faceta del componente cuyo nombre se pasa como parámetro.

- connect: Realiza la conexión receptáculo-faceta, pasándole la referencia a una faceta, obtenida por ejemplo a través de *provide_facet*, y el nombre del receptáculo del componente al que se va a conectar. La referencia a la faceta se almacena en el contexto, para que posteriormente el componente pueda acceder a ella.

- disconnect: pone a null las referencias almacenadas en el contexto correspondientes al receptáculo cuyo nombre se pasa como parámetro.

Para desarrollar esta funcionalidad, en la parte privada de esta clase, se almacenará la siguiente información:

- Se almacenarán referencias a los envoltorios de las facetas (denominadas *proxy_[TipoInterfaz]*. Las clases correspondientes a estos envoltorios o *wrapper* de las facetas estarán definidos en la parte privada del paquete del *wrapper*. Estas clases serán las que capturen las llamadas que se realicen sobre las facetas del componente, y las trasladen a las correspondientes implementaciones. En ellas se incluirán los posibles interceptores que se añadan a las operaciones de un componente con el objetivo de gestionar propiedades no funcionales de la aplicación.

- Se almacenará una referencia al ejecutor del componente que será instanciado en el *wrapper* y a través del cual, el wrapper tiene acceso a la implementación del componente.

- Existirá una referencia al contexto, necesaria para almacenar la información acerca de todas las conexiones del componente (realizadas a través de las operaciones connect y disconnect).

- Por último existirá una referencia a la interfaz equivalente del home del componente.

A continuación se muestra el diagrama UML de la clase *GralComponent_Wrapper*, correspondiente al componente que hemos puesto de ejemplo en el apartado anterior:

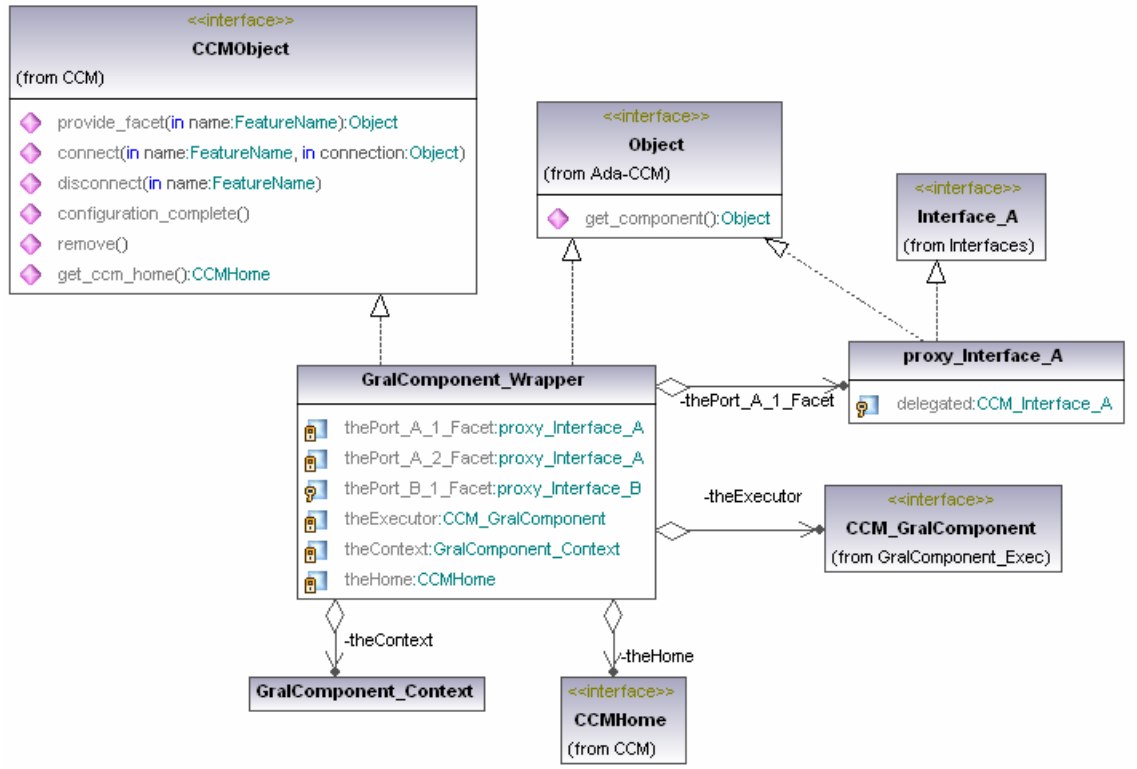

Figura 3.4: Diagrama UML de las clase correspondiente al wrapper.

La clase que representa la implementación concreta del contexto, de nombre *[NombreComponente]_Context*, implementa la interfaz *CCM_[NombreComponente]_Context* definida en el paquete del ejecutor comentado en el apartado anterior.

Debido a que realiza la implementación de la interfaz del contexto definida en el ejecutor, implementará las operaciones *get_connection_[NombreReceptaculo]* a través de las cuales accederá a las referencias de los receptáculos que se encuentran almacenados en él, y que se establecieron a través de las operaciones *connect* de la clase *[NombreComponente]_Wrapper* que hemos comentado antes.

A continuación se muestra el diagrama UML de esta clase para el caso del componente GralComponent que hemos puesto de ejemplo durante el capítulo. En él se muestran las operaciones ofrecidas por la clase, así como los atributos que se necesitan para poder almacenar la información correspondiente a cada receptáculo:

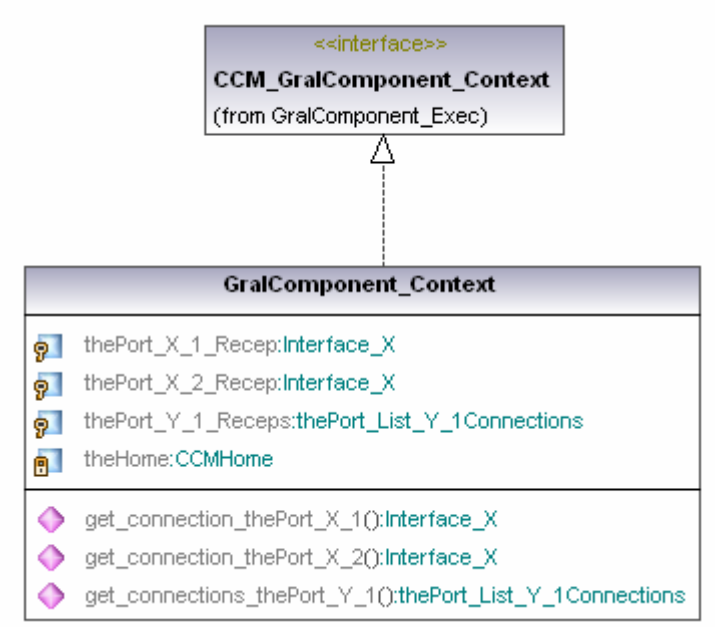

Figura 3.5: Diagrama UML de las clases del contexto del wrapper.

Por último en el paquete del *wrapper* se genera una clase que ofrece la interfaz equivalente del home del componente, *[NombreComponente]_Home_Wrapper* que hereda de *HomeConfiguration* (para poder realizar configuración de los componentes), y de las interfaces *KeylessCCMHome* (para poder crear la instancia del componente) e *I_Set_Home_Executor*. Estas tres clases/interfaces de las que hereda/implementa no dependen de los componentes de la aplicación, sino de la propia tecnología, no es código generado sino reutilizado.

Se muestra a continuación el diagrama UML de la clase *GralComponent_Home* para el caso del ejemplo:

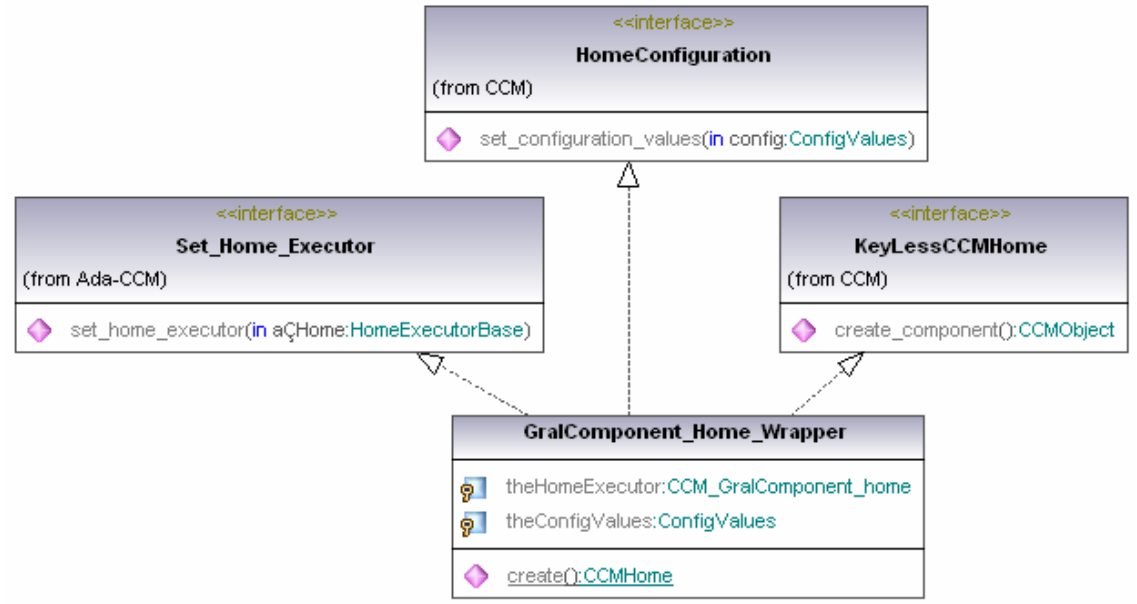

Figura 3.6: Diagrama UML de las clases correspondientes al Home del wrapper.

Hay que incluir en esta clase un procedimiento de creación para las home, ya que se utilizará como punto de entrada (entry Point), para desde la herramienta de despliegue crear la instancia del home y poder comenzar la creación de componentes.

Esta clase almacenará dos elementos, una referencia al home del ejecutor y una lista con los valores de configuración (*config values*) del componente. La herramienta de despliegue se encargará de fijar estos valores de configuración a través de la operación *set_configuration_values* definida en esta clase como herencia de *HomeConfiguration.* Posteriormente, estos valores serán empleados para configurar el componente, a través de los procedimientos *set_[attributeName]* definidos para cada uno de los atributos del componente. Será también tarea de la herramienta de despliegue, almacenar la referencia al home del ejecutor una vez que lo cree, mediante la operación *set_home_executor* implementada en esta clase como consecuencia de la implementación de la interfaz *I_Set_Home_Executor*.

Como herencia de *KeyLessCCMHome* deberá implementar la operación *create_component*, que será ejecutada también por parte de la herramienta de despliegue y que realizará la instanciación del componente, siguiendo los siguientes pasos:

- Crea el wrapper.
- Crea el ejecutor (a través de la referencia *theHomeExecutor*) y almacena una referencia al mismo en el wrapper.
- Crea el contexto, pasándole su referencia al ejecutor (a través de la operación set_session_context, que dicho ejecutor implementa).

- Crea las instancias correspondientes a cada una de las facetas (de las correspondientes clases *proxy_[TipoInterfazFaceta]*).

- Configura el componente con los valores de configuración previamente introducidos por la herramienta de despliegue.

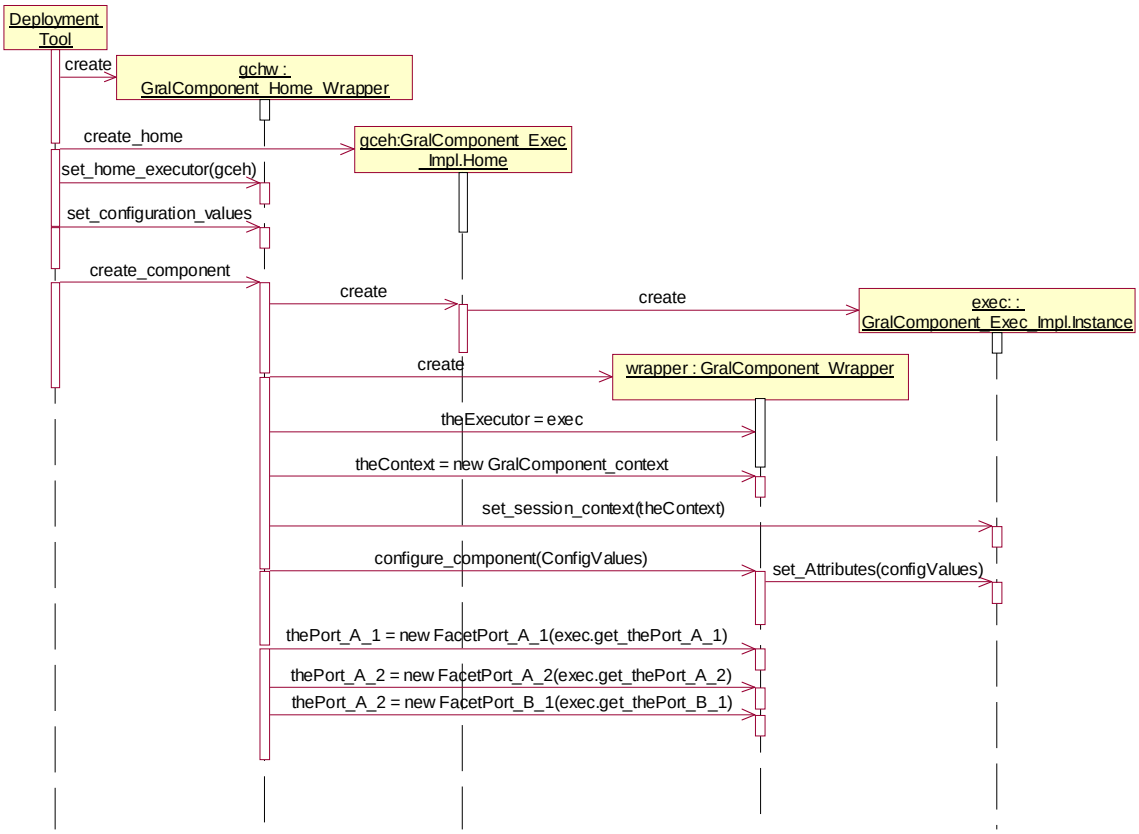

Figura 3.7: Diagrama de secuencias de la instanciación de un componente.

3.3 ESTRUCTURA DE LA IMPLEMENTACIÓN

En el apartado 3.1 se explican el conjunto de interfaces que se definen en el paquete correspondiente al ejecutor y que sirven de base para la implementación de los componentes y de los home. Como implementación de dichas interfaces se declaran un conjunto de clases que sirven para desarrollar una implementación concreta de un componente y que en la tecnología Ada-CCM se agrupan en dos paquetes Ada:

- El paquete *[NombreComponente]_Exec_Impl*, cuyo código será generado de forma automática por la herramienta de generación de código.

- El paquete *[NombreComponente]_Impl*, que será en parte generado automáticamente y será el fichero en el que el implementador introduzca el código de negocio del componente.

La razón de haber tomado esta estrategia ha sido principalmente la de realizar la mayor separación posible entre el código de negocio y el código generado automáticamente, con el fin de simplificar la tarea del implementador haciendo opaco al mismo el resto de código generado. En el futuro, con el desarrollo de una herramienta de diseño visual propia de esta tecnología, se limitará la visión del implementador al código de negocio, consiguiendo simplificar la tarea del mismo, y centrar su esfuerzo únicamente en el desarrollo de dicho código.

3.3.1 El paquete [NombreComponente]_Exec_Impl

En este fichero se declara la clase que representa la implementación del componente. En realidad se trata de una clase intermedia, que contiene como agregada la clase que realmente incorpora el código que implementa la funcionalidad del componente (patrón tie). Esta clase es necesaria para implementar el componente siguiendo la normativa impuesta por CCM, según la cual todo componente debe heredar de los correspondientes ejecutores definidos en el fichero anterior. Todo su código es generado automáticamente. En este paquete se declaran dos clases: la clase *Instance* y la clase *Home*.

La clase *Instance* implementa la interfaz *Object* y la interfaz *CCM_[NombreComponente]* correspondiente a la interfaz monolítica definida en el paquete *[NombreComponente]_Exec*. Esta clase incorpora como atributos:

- Una referencia al contexto, que le pasará el contenedor cuando la instancia sea creada a través del método *set_session_context* que se implementa como consecuencia de la herencia de la interfaz monolítica.
- La referencia a la clase agregada *[NombreComponente]_Impl*, del paquete de mismo nombre que será donde el implementador introduzca el código de negocio.

El código de casi todas las operaciones es una reinvocación del mismo método sobre la referencia a la clase agregada, excepto las operaciones *set_session_context* y *configuration_complete*, cuyo código es generado de forma automática directamente en esta clase.

En la figura que se muestra a continuación, se muestra el diagrama UML de esta clase para el componente GralComponent que estamos tomando como ejemplo durante este capítulo:

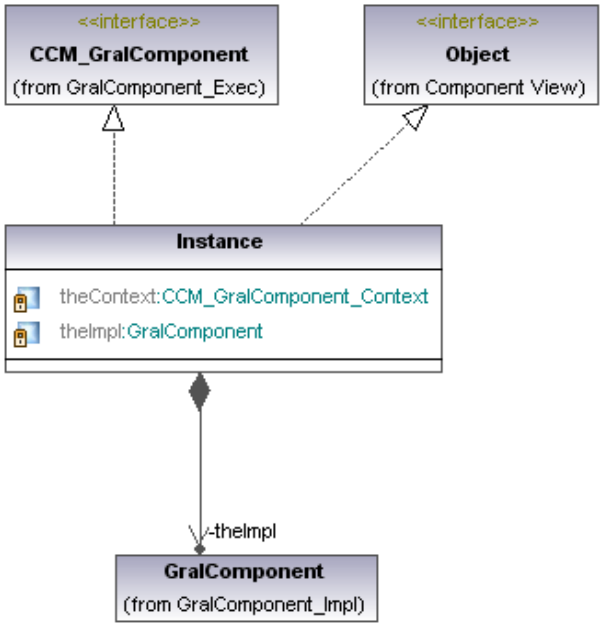

Figura 3.8: Diagrama UML de la clase Instante del paquete Exec_Imp.

La otra clase definida en este paquete es la clase *Home* que implementa la interfaz *CCM_[NombreHome]* definida en el paquete del ejecutor. No posee atributos. La operación *create* que se ha de implementar en esta clase (heredada de *CCM_[NombreHome]*) genera una instancia de la clase *Instance*, que a su vez instancia un objeto de la clase *[NombreComponente]_Impl*.

Se añade en el fichero la operación *create_home* para la creación externa de objetos de la clase *Home*, necesaria durante la fase de instanciación de los componentes.

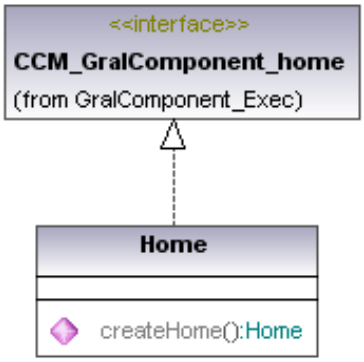

Figura 3.9: Diagrama UML de la clase Home del paquete Exec_Imp.

3.3.2 El paquete [NombreComponente]_Impl

Como hemos comentado al comienzo de este capítulo, éste será el paquete en el que el usuario introducirá el código de negocio de los componentes. Parte del código será generado de forma automática, como por ejemplo los esqueletos de todas las operaciones y la declaración de las clases.

Antes de comenzar a describir la estructura de este paquete, cabe destacar que como estrategia propia de la tecnología Ada-CCM, se ha optado por la creación de un componente denominado RTComponent, en el cual mediante la implementación de una interfaz denominada I_Component incluirá una serie de operaciones comunes a todos los componentes. De esta forma al heredar todos los componentes en esta tecnología de este componente, poseerán dichas operaciones comunes a todos ellos.

Figura 3.10: Diagrama UML de la clase Instance del paquete de implementación.

En este paquete como clase pública se definirá la clase Instance que heredará de la correspondiente RTComponent_Impl y que representará la implementación del

CAPÍTULO 4

ESTRUCTURA DE LOS CONECTORES ENTRE COMPONENTES

4.1 ESTRUCTURA GENÉRICA DE UN CONECTOR

En esta tecnología la idea del conector se basa en la de tratar al mismo como otro componente más, en el cual el código de negocio será generado automáticamente y cuya función es compatibilizar las invocación de las operaciones entre componentes que pueden estar implementados con diferente lenguaje de programación (lo que puede requerir operaciones de *marshalling* y *unmarshalling*), implementados en diferentes nudos de procesamiento (lo que hace que la invocación conlleve uso de los servicios de comunicaciones), o implementados en diferentes plataformas (lo que conlleva utilizar diferentes mecanismos de comunicación).

Las conexiones realizadas por los conectores se realizan entre los puertos de los componentes implicados, esto es, entre una faceta y un receptáculo. Por tanto, cada conexión vendrá asociada a un determinado tipo de interfaz.

Como hemos comentado, el código será generado automáticamente, de ahí que la herramienta que lo genere, en este caso la herramienta de despliegue, deba poseer la siguiente información:

- Interfaz asociada a los puertos que conecta. En esta tecnología, al tratarse los conectores como componentes, vendrá dada por la especificación IDL del conector.
- Servicio de comunicaciones empleado para la conexión. Este dato habría que aportarlo como propiedad de configuración de la conexión entre los

componentes en el plan de despliegue y en función de él, elegir la plantilla de conector que debemos emplear. En nuestro caso hemos utilizado GLADE.

- Nodo cliente y nodo servidor. En función de que sean el mismo o diferente nodo el tipo de conector será local o remoto.

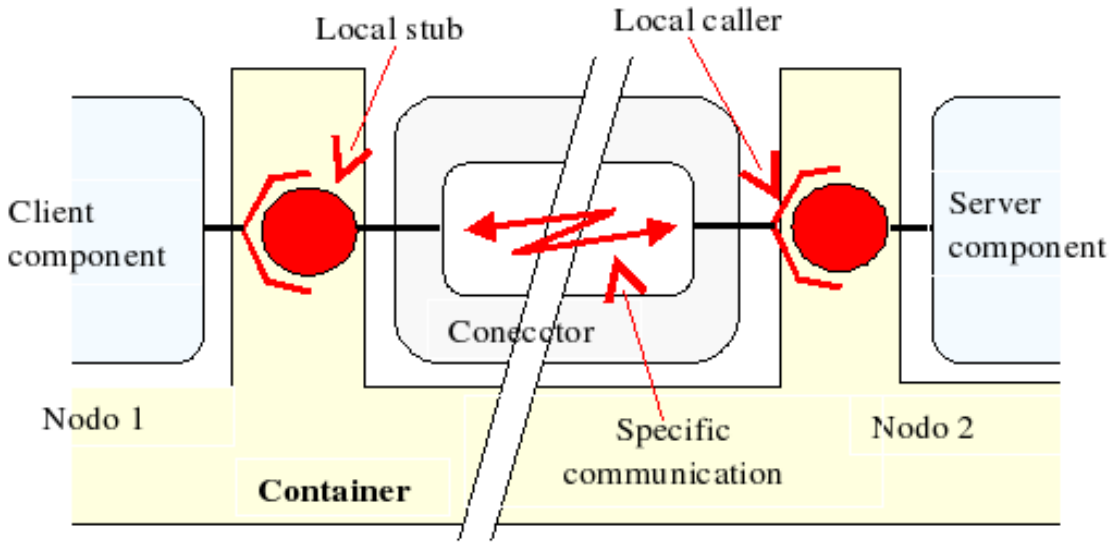

Figura 4.1: Esquema de conexiones de un conector con los componentes.

Como hemos comentado, el código será generado automáticamente, de ahí que la herramienta que lo genere, en este caso la herramienta de despliegue, deba poseer la siguiente información:

- Interfaz asociada a los puertos que conecta. En esta tecnología, al tratarse los conectores como componentes, vendrá dada por la especificación IDL del conector.

- Servicio de comunicaciones empleado para la conexión. Este dato habría que aportarlo como propiedad de configuración de la conexión entre los componentes en el plan de despliegue y en función de él, elegir la plantilla de conector que debemos emplear. En nuestro caso hemos utilizado GLADE.

- Nodo cliente y nodo servidor. En función de que sean el mismo o diferente nodo el tipo de conector será local o remoto.

En el caso de conectores distribuidos, el conector se divide en dos fragmentos, el *proxy* que se instancia en el nodo cliente y el *servant* que se instancia en el nodo servidor. Entre ellos se establecerá la comunicación por el medio que se haya especificado.

4.1.1 Fragmento Proxy del conector

El fragmento proxy del conector ofrecerá como faceta (como interfaz local) la interfaz requerida por el componente cliente. La descripción IDL necesaria para la generación del código por parte del generador sería:

```
Component connector_[NombreInterfaz]_proxy {
        Provides  [NombreInterfaz] [NombreFacet];
}
```

Su código se genera como si fuese un componente más, ya que las conexiones con el componente cliente se llevarán a cabo siguiendo la normativa CCM. Por cada operación ofrecida a través de la interfaz, el código del proxy realizará todas las actividades necesarias para realizar la invocación distribuida de la correspondiente operación.

4.1.2 Fragmento Servant del conector

Este fragmento del conector requiere la interfaz requerida por el cliente (y ofertada por el servidor). Por ello, poseerá un receptáculo correspondiente a la interfaz que implementa los puertos que conecta. Este fragmento del conector incluirá toda la infrastructura necesaria para gestionar la recepción de invocaciones distribuidas (según el servicio de comunicaciones utilizado) y a través de su receptáculo dirigirá dichas peticiones sobre el correspondiente componente servidor.

La descripción IDL de este fragmento será por tanto:

```
Component connector_[NombreInterfaz]_servant {
        uses  [NombreInterfaz] [NombreReceptaculo];
}
```

4.2 ESTRUCTURA DE UN CONECTOR QUE IMPLEMENTE EL DSA DE ADA

4.2.1 Introducción a GLADE

DSA (*Distributed Systems Annex E*) es el anexo E del manual de referencia de Ada, en el cual se trata sobre el soporte de implementaciones de sistemas distribuidos usando múltiples particiones trabajando juntas como parte de un único programa Ada. Existen varias implementaciones del DSA que soportan particionamiento y distribución de aplicaciones Ada. Una de estas implementaciones es GLADE (*GNAT Library for Ada Distributed Environments*), desarrollado en sus inicios por Pautet y Tardieu[13], y que se incluye actualmente en el proyecto GNAT desarrollado por *Ada Core Technologies* (ACT)[14].

La base de las aplicaciones distribuidas de Ada son las particiones. Básicamente una aplicación distribuida se compone de al menos un par de particiones. Cada una de

las particiones de una aplicación basada en GLADE, a la hora de la compilación se va a convertir en un ejecutable independiente. Cada uno de estos ejecutables serán los que se ejecuten por separado y se comuniquen entre ellos, por medio de invocaciones remotas (RPCs) y de la existencia de objetos distribuidos. Existe una herramienta que facilita todo este proceso: *gnatdist*.

Gnatdist lee un fichero de configuración en el que se especifica cómo queremos distribuir la aplicación y genera todos los ejecutables necesarios y toda la infraestructura que se requiere para soportar las invocaciones remotas entre particiones. De esta forma, es posible probar diferentes formas de distribuir una misma aplicación simplemente con lanzar *gnatdist* con un fichero de configuración distinto, sin necesidad de modificar el código de la aplicación.

4.2.2 Implementación del fragmento Servant con GLADE

La utilización de la implementación del DSA de Ada, implica la categorización de paquetes con los pragmas *RCI* y *Remote_types*. Esto tiene como consecuencia que las peticiones realizadas por el cliente sobre operaciones definidas en las facetas, al realizarse sobre GLADE, tendrán restringidos los tipos que podrán utilizarse.

Como queremos hacer al implementador independiente de la comunicación entre los componentes, abordaremos este problema en el conector mapeando los tipos que el cliente utilice (por ejemplo en los argumentos de las operaciones) a tipos permitidos por GLADE. Para realizar esto, hemos dividido los posibles tipos que se pueden emplear en la especificación de un componente en dos paquetes. El paquete *Remote_Common_Types* está categorizado con el pragma *Remote_Types* y contendrá la lista de tipos que pueden ser empleados en invocaciones remotas sin necesidad de mapeado. En el paquete *Common_Types* se han definido los posibles tipos que habrán de mapearse a tipos compatibles utilizando las operaciones contenidas en dicho paquete para su mapeo.

Para ser capaces de realizar invocaciones distribuidas utilizando GLADE, se requiere la categorización de paquetes con el pragma *RCI*, siendo los procedimientos declarados en estos paquetes sobre los que se pueden realizar invocaciones remotas. Como consecuencia de todo esto, en el lado del *servant*, además del fragmento *servant* del conector, se generarán de forma automática dos paquetes adicionales:

- *Connector_servant_RCI* : Paquete con pragma *RCI*. Ofrece todas las operaciones de la interfaz implementada por el conector pero mapeadas a tipos que sean soportados por el pragma *Remote_Types* de Ada. Todos los tipos que se incluyen en el fichero *Remote_Common_Types* son soportados directamente. Para el resto de los tipos hay que hacer mapeados, por ejemplo: Sequences => Arrays; Strings => Strings Ada, etc.

- *Connector_servant_Adapter*: Sirve para realizar la comunicación entre el *Connector_servant_RCI* y el componente conector *servant*. Es necesario introducirlo porque el componente conector tiene que ser instanciado de forma independiente, para poder ser accedido desde la herramienta de despliegue con el fin de conectarlo al componente servidor. Debido a ello,

no lo podemos conectar directamente con el RCI, ya que en ese caso solo podría hacer referencia a paquetes *Remote_Types*, y eso, según está diseñado el componente no es posible. El componente *adapter* ofrece hacia fuera las mismas operaciones que el *RCI* y mantiene una referencia al componente *Connector_Servant* sobre el que redirige las invocaciones recibidas, haciendo previamente el mapeado a los tipos propios de CCM.

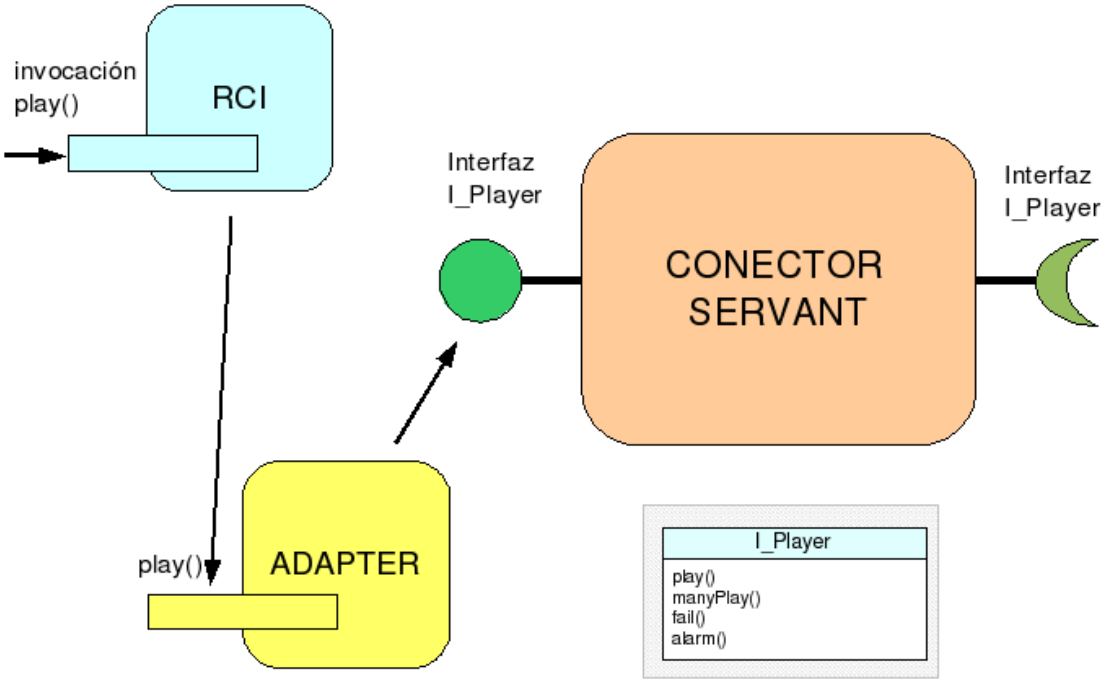

Figura 4.2: Esquema del proceso de reinvocación de peticiones en el servant.

El fragmento *servant* de un conector GLADE ofrecerá una faceta con la misma interfaz que el correspondiente receptáculo, de forma que cuando el *Adapter* recibe una invocación procedente del *RCI*, puede redirigirla a la correspondiente faceta del conector *servant*, que a su vez la dirigirá a través de su receptáculo, al componente servidor.

4.2.3 Implementación del fragmento Proxy con GLADE

La parte *proxy* del conector no requerirá el uso de paquetes adicionales. Cuando recibe una invocación por parte del componente cliente en cualquiera de las operaciones ofrecidas en la faceta del conector, se realiza el mapeado de aquellos tipos que lo requieran utilizando las operaciones destinadas a ese efecto y se invocará la correspondiente operación en el *RCI*. El mapeado y posterior invocación de la correspondiente *RCI* se realizará en el paquete de implementacion *[NombreConector]_proxy_impl* del componente correspondiente al conector *proxy*.

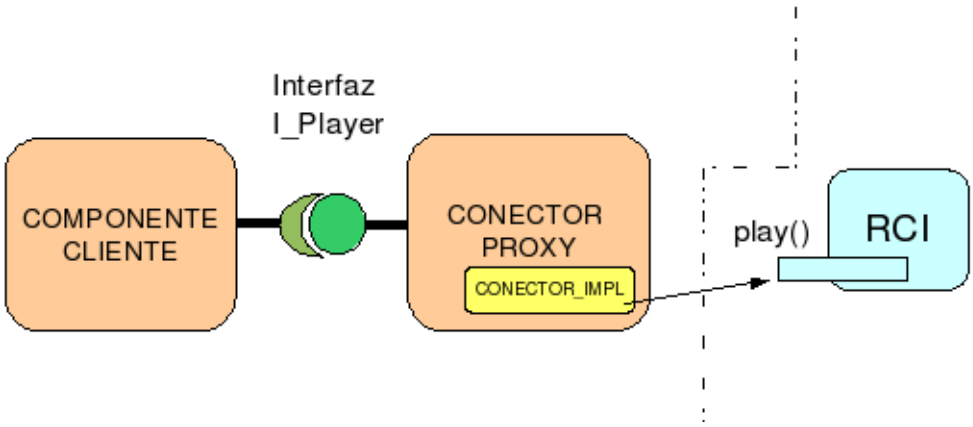

Figura 4.3: Esquema del proceso de reinvocación de peticiones en el proxy.

4.2.4 Proceso de despliegue de los conectores

Los conectores no se declaran de forma explícita, sino que basándonos en la información o propiedades asignadas a las conexiones, la herramienta de despliegue automática generará el código del correspondiente conector y los despliegues parciales necesarios para instanciar cada componente y cada fragmento de conector en las particiones correspondientes.

En la tecnología Ada-CCM se añade la posibilidad de asignar propiedades de configuración a las conexiones para poder configurar los conectores. Entre estas propiedades se incluyen:

- Tipo de conexión a emplear. En nuestro caso GLADE. Con esta información, la herramienta de generación de código elegirá las plantillas a emplear para desarrollar los componentes.
- Local o remota: Se puede incluir como una propiedad, o directamente obtenerla comparando los nodos en que están instanciados los componentes conectados. (También podría especificarse este aspecto como un tipo especial de conexión, en la propiedad anterior).
- En función del tipo de conexión empleada, habrá que asignarle propiedades de configuración especiales.

Aunque el concepto de conector implique una conexión entre puertos de componentes, podemos implementarlo como un conector único por faceta de un componente, a cuyo proxy estén conectados todos aquellos componentes que hagan peticiones sobre dicho faceta desde una misma partición. El proceso de despliegue en este caso lo vamos a realizar de la forma que se explica en el siguiente diagrama.

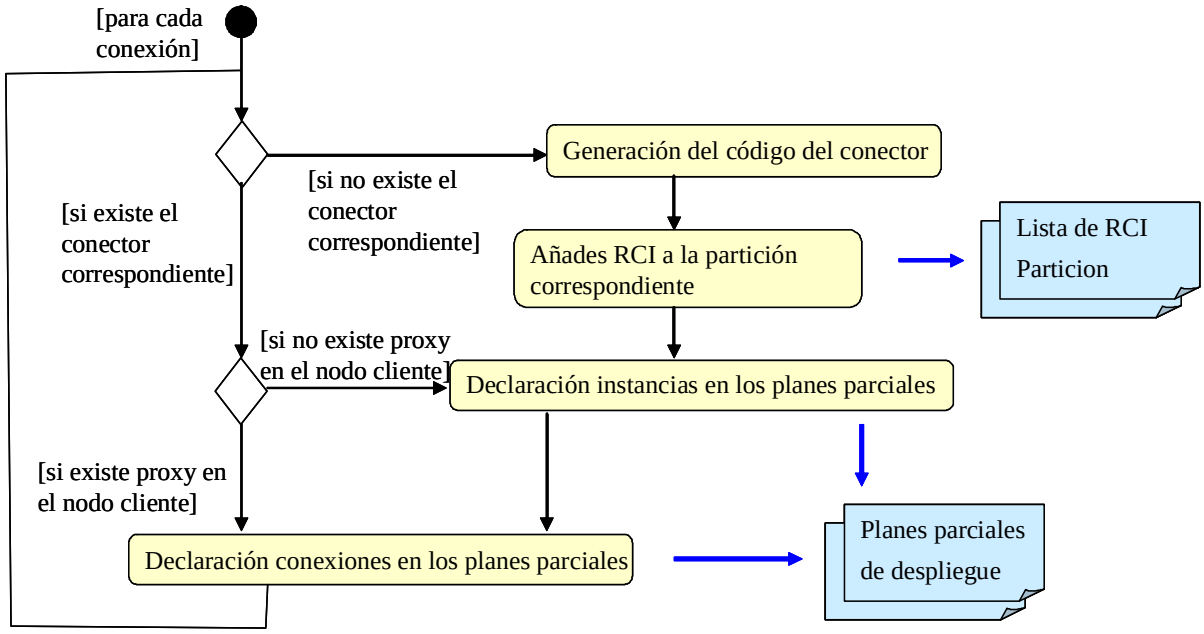

Figura 4.4: Esquema del despliegue de los conectores GLADE.

Por cada conexión declarada en el fichero de despliegue se evalúa si el conector asociado a dicha faceta de la instancia del componente servidor ya ha sido desarrollado. Si no existe, se genera su código:

- Si la conexión es local, se genera el *conector_proxy* y el *conector_servant* y se añaden sus correspondientes instancias, así como las conexiones entre ellas y las instancias cliente y servidor correspondientes al plan de despliegue.
- Si la conexión es de tipo GLADE, se genera la estructura que hemos explicado (los dos fragmentos del conector, el adapter y el RCI). Se añade una instancia del tipo servant a la partición servidor y una conexión entre el servant y el componente servidor. Se añade una instancia del tipo proxy a la partición cliente y la conexión entre ella y el componente cliente. El RCI generado se añade a la lista correspondiente a la partición servidor (se mantiene una lista de los RCI asociados a cada partición que será utilizada posteriormente para la generación del fichero de configuración GLADE).

Si el conector ya existía:

- Si la conexión es local, se añade una conexión entre el cliente y el proxy del conector.
- Si la conexión es distribuida (GLADE):
  - Si el fragmento *proxy* existe en el nodo cliente: Se declara una conexión entre el componente cliente y el proxy del conector.

- Si el fragmento *proxy* no existe en el nodo cliente: Se declara una instancia del proxy correspondiente en esa partición y una conexión entre ella y el componente cliente.

Además del código de los componentes y de los conectores, para desarrollar una aplicación utilizando GLADE, es necesario elaborar los procedimientos principales que se van a ejecutar en cada partición, que contendrán el código necesario para instanciar los conectores y componentes correspondientes a cada una de ellas. Para ello, durante el procesamiento del fichero de despliegue, se generarán despliegues parciales, en los que una herramienta posterior se basará para elaborar automáticamente el código de dichos procedimientos.

Una vez generado todo el código necesario para compilar y ejecutar la aplicación, es necesario generar el fichero de configuración de GLADE. Para ello necesitamos conocer el nombre de las particiones y para cada partición el nombre de host y de puerto por el que se quieren realizar las peticiones. Dichos datos se podrán extraer del fichero que contiene la descripción de la plataforma. Para la generación de este fichero se necesitará también la lista que mantiene los paquetes *RCI* que se han de asociar a cada partición y que se ha ido generando durante el proceso de generación del código de los conectores.

A continuación se muestra el contenido de un fichero de configuración sencillo que configura una aplicación para ejecutarse de forma distribuida:

```
configuration config is
pragma Starter (None);

particion1: partition;
procedure particion_cliente is in Particion1;
particion2: partition := (connector_servant_rci);
procedure particion_server;
for particion2'Main use particion_server;


pragma Boot_Location ("tcp","linux8a:12201");

for particion1'Self_Location use ("tcp","linux8a:12203");
for particion2'Self_Location use ("tcp","linux8:12205");

end config;
```

De este fichero podemos ver que el procedimiento principal que se ejecutará en la partición 1 será *particion_cliente*, por su parte en la partición 2 se ejecutará el main *particion_server*, y será la partición en que se encontrará el fichero *RCI* de nombre *connector_servant_rci*. Vemos también que el tipo de conexión que se establecerá será TCP, ejecutándose la partición 1 en la máquina linux8a usando el puerto 12203 y la partición 2 se ejecutará sobre la máquina linux8 utilizando el puerto 12205.

Este fichero lo utilizará *gnatdist* para compilar la aplicación y de esta forma generar los ejecutables correspondientes a los nodos que intervienen. Una vez generados la herramienta de lanzamiento los llevará a los nodos que corresponda y lanzará la ejecución de cada uno de ellos, poniendo la aplicación en funcionamiento.

CAPÍTULO 5

HERRAMIENTAS PARA LA GENERACIÓN AUTOMÁTICA DE CÓDIGO

5.1 ESTRUCTURA DEL REPOSITORIO

La información necesaria para la generación de código se encuentra en la base de datos que forma el repositorio. Toda la información asociada a los componentes disponibles, ya sean, como consecuencia de nuevos desarrollos o de adquisiciones de otros desarrollados por terceros, se registran en esta base de datos. Así mismo, en el repositorio existe almacenada información sobre las plataformas de ejecución que se encuentran especificadas, y sobre aplicaciones que se disponen desarrolladas y dispuestas para su ejecución. El repositorio es accedido con diferentes objetivos por las diferentes herramientas que se utilizan en el desarrollo de componentes y aplicaciones:

- Las herramientas de implementación acceden a la información de los componentes registrados a fin de analizar la información de los componentes cuyas instancias son integradas en el nuevo componente compuesto, y así mismo, hacen referencia a los componentes registrados en el repositorio cuando elaboran la descripción de un nuevo componente compuesto.

- Las herramientas de enpaquetamiento hacen referencia a la información almacenada sobre los componentes registrados en el repositorio, y almacenan en él la información relativa al nuevo componente que genera.

- Las herramientas de análisis del comportamiento y de ajuste de la configuración, accede al repositorio para obtener la descripción y los modelos de los componentes y de los elementos que constituyen la

plataforma de ejecución y que se encuentran descritos en el plan de despliegue.

- Las herramientas de despliegue acceden al repositorio para obtener la información que necesita para la gestión de los componentes incluidos en el plan de despliegue y para obtener información de la plataforma sobre la que ha de realizar el despliegue.

5.2 ESTRATEGIA DE GENERACIÓN DE CÓDIGO Y HERRAMIENTAS DE GENERACIÓN

Como hemos comentado anteriormente, la generación del código automático se realiza en base a la especificación IDL, ya sea de las interfaces que intervienen como de los componentes utilizados en la aplicación.

Los ficheros IDL, a pesar de poseer una estructura bien definida son ficheros de texto plano, cuyo procesado desde una herramienta de generación de código puede resultar compleja y pesada para la aplicación.

En esta tecnología se ha desarrollado una herramienta que genera ficheros XML en base a las descripciones IDL. XML(eXtensible Markup Language), es un metalenguaje apto para almacenar la información contenida en un fichero IDL de forma estructurada. XML es actualmente una Recomendación completa de la World Wide Web Consortium (W3C). Las razones que nos han llevado a utilizar este tipo de ficheros son fundamentalmente:

- Permite acotar la información contenida y definir la estructuración de la misma, mediante es uso de esquemas XML. Un fichero generado por esta herramienta deberá ser validado con el correspondiente esquema, de manera que a la hora de ser procesado la estructura generada sea la prevista por la herramienta que lo procesa.

- Se disponen de APIs en diferentes lenguajes para el procesado de este tipo de ficheros. La API DOM (Document Object Model) genera un árbol jerárquico en memoria del documento XML que procesa, haciendo que una vez creado este árbol la obtención de la información del mismo se realiza de forma intuitiva, y con un peso muy inferior al que tiene que soportar la máquina si el procesado fuese sobre texto plano. Java posee una potente implementación de esta API, lo que nos ha llevado a desarrollar la herramienta de generación de código en este lenguaje.

Como ejemplo de estos ficheros voy a poner el correspondiente a la especificación del componente GralComponent cuya descripción IDL fue vista en el capítulo 2.2.1:

```
<?xml version="1.0" encoding="UTF-8"?>
<IDL_Specification xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="C:\XML\IDL_XML\IDLSpecification.xsd"
       proyect="RT-CCM"
       institution="Computadores y Tiempo Real - Universidad de Cantabria"
       authors="Pablo Pacheco"
       version="02/08//07" >
       <include name="Interfaces/gral/Interface_A.idl"/>
       <include name="interfaces/gral/Interface_B.idl"/>
       <include name="interfaces/gral/Interface_X.idl"/>
       <include name="interfaces/gral/Interface_Y.idl"/>
       <include name="components/Gral/RTComponent.idl"/>
       <body>
              <simpleType name="Attr_1_Type" type="short"/>
              <simpleType name="Attr_2_Type" type="short"/>
              <component name="GralComponent" inheritanceBase="RTComponent">
                     <facet portName="thePort_A_1" interfaceType="Interface_A"/>
                     <facet portName="thePort_A_2" interfaceType="Interface_A"/>
                     <facet portName="thePort_B_1" interfaceType="Interface_B"/>
                     <facet portName="theOSAPort1" interfaceType="OneShotActivation"/>
                     <facet portName="theOSAPort2" interfaceType="OneShotActivation"/>
                     <facet portName="thePAPort1" interfaceType="PeriodicActivation"/>
                     <facet portName="thePAPort2" interfaceType="PeriodicActivation"/>
                     <receptacle portName="thePort_X_1" interfaceType="Interface_X"/>
                     <receptacle portName="thePort_X_2" interfaceType="Interface_X"/>
                     <receptacle portName="thePort_Y_1" interfaceType="Interface_Y"
                                 cardinality="MULTIPLE"/>
                     <attribute name="attr_1" type="Attr_1_Type"/>
                     <attribute name="attr_2" type="Attr_2_Type"/>
              </component>
              <home name="GralComponent_home" manages="GralComponent"
                    inheritanceBase="RTComponent_home"/>
       </body>
</IDL_Specification>
```

La herramienta de generación de código ha sido implementada haciendo uso de plantillas de código Ada que hemos tenido que crear. A modo de documentación se ha incluido en el Anexo el contenido completo de las mismas.

El procedimiento de generación de código en base a plantillas es muy sencillo. Tomando como fichero de partida el correspondiente a la descripción XML de la especificación IDL del componente, generaremos el código final conformando la estructura que hemos definido en el capitulo 3. La utilización de plantillas se basa en el uso de las mismas sustituyendo aquellas palabras clave marcadas de la forma *@Parametro@*, por las correspondientes según sea el caso.

Cada plantilla comenzará con el nombre del identificador de la misma, de la forma *<IdentificadorPlantilla>*. Tras esta etiqueta se encontrará el código Ada que será insertado al utilizar esta plantilla, con los parámetros a completar a partir de la información procesada del fichero XML. Y se finalizará el contenido de la plantilla indicando nuevamente el identificador de la plantilla pero ahora de la forma *</IdentificadorPlantilla>*.

Por ejemplo, si queremos declarar la interfaz monolítica del ejecutor querremos generar un código cuya plantilla podría ser:

```
<MonolithicExecutorInterfaceHead>
       type CCM_@ComponentName@ is interface
              and @FatherComponent@_Exec.CCM_@FatherComponent@;
       type CCM_@ComponentName@_Ref is access all CCM_@ComponentName@'Class;
</MonolithicExecutorInterfaceHead>
```

Esta plantilla requeriría como parámetros *@ComponentName@* y *@FatherComponent@*, cuya información la tendríamos que obtener de la especificación del componente.

5.2.1 Ejemplo de funcionamiento de la herramienta de generación de código

Para explicar el funcionamiento de esta herramienta vamos a poner como ejemplo la generación del código Ada correspondiente al ejecutor, el cual será generado a partir del fichero *GralComponent.idl.xml* que hemos mostrado.

La generación del ejecutor se realizará en el paquete Ada de nombre *GralComponent_Exec.ads*. La herramienta de generación de código cargará el fichero *Component_ExecTemplate.txt* definido en el Anexo, correspondiente a las plantillas necesarias para la generación de este paquete.

Lo primero que habrá que hacer en dicho paquete será la importación de los paquetes necesarios. Para realizar esta tarea haremos uso de las plantillas With y Use que se muestran a continuación:

```
<With>
with @Package@;</With>

<Use>
use @Package@;</Use>
```

Para importar determinados paquetes será necesario poseer la información procesada por DOM de la especificación IDL. Así por ejemplo por cada elemento *facet* o *receptacle* definido en el componente se requerirá el uso de la plantilla With introduciendo como parámetro el atributo *interfaceType* leído de dichos elementos, es decir, la interfaz que corresponde a la faceta o receptáculo.

Se muestra el código generado para el ejemplo del GralComponent correspondiente a los with necesarios:

```
with Components;
with Common_Types;
use Common_Types;
with Remote_Common_Types;
use Remote_Common_Types;
with Interface_A;
with Interface_B;
with Interface_X;
with Interface_Y;
with RTComponent_Exec;
with Ada.Containers.Vectors;
with gralcomponent_Types;
use gralcomponent_Types;
```

Una vez declarados los paquetes a importar siguiendo el procedimiento que se ha explicado, se pasa a la declaración de las interfaces *facet executor*. Por cada elemento *facet* que no sea puerto de activación (comprobable leyendo el atributo *interfaceType*) se utilizará la siguiente plantilla:

```
<FacetInterfaceHead>
        type CCM_@InterfaceName@ is interface and @InterfaceName@.Face;
        type CCM_@InterfaceName@_Ref is access all CCM_@InterfaceName@'Class;
</FacetInterfaceHead>
```

El código correspondiente a las *facet executor interfaces* de este componente que se ha generado en base a esta plantilla se muestra a continuación:

```
type CCM_Interface_A is interface and Interface_A.Face;
type CCM_Interface_A_Ref is access all CCM_Interface_A'Class;

type CCM_Interface_B is interface and Interface_B.Face;
type CCM_Interface_B_Ref is access all CCM_Interface_B'Class;
```

La generación de la interfaz monolítica se generará de una forma algo más compleja que como se ha contado a la hora de explicar el uso de las plantillas. En realidad son necesarias básicamente 3 plantillas. Esto es necesario porque es posible que un determinado componente soporte alguna interfaz. Las plantillas necesarias para realizar dicho código serían por tanto:

```
<MonolithicExecutorInterfaceHead>
        type CCM_@ComponentName@ is interface
                and @FatherComponent@_Exec.CCM_@FatherComponent@
</MonolithicExecutorInterfaceHead>

<MonolithicExecutorInterfaceWithSupport>
                and @SupportedInterface@.face</MonolithicExecutorInterfaceWithSupport>

<MonolithicExecutorInterfaceRef>;
        type CCM_@ComponentName@_Ref is access all CCM_@ComponentName@'Class;
</MonolithicExecutorInterfaceRef>
```

Y el código generado correspondiente a la declaración de la monolítica del componente GralComponent será:

```
type CCM_gralcomponent is interface
        and RTComponent_Exec.CCM_RTComponent;
type CCM_gralcomponent_Ref is access all CCM_gralcomponent'Class;
```

Esta interfaz que hemos declarado contendrá varias operaciones:

Por cada atributo (elemento de tag *attribute* en la estructura XML) se añaden las operaciones de acceso y escritura (si no es atributo de sólo lectura). Por tanto las plantillas necesarias serán:

```
<AttrOperationHead>
        function get_@Attr_Name@(Self: access @PropietaryClassName@)
                return @Attr_Type@ is abstract;
        procedure set_@Attr_Name@(Self: access @PropietaryClassName@;
                attribute: in @Attr_Type@) is abstract;
</AttrOperationHead>

<AttrReadOnlyOperationHead>
        function get_@Attr_Name@(Self: access @PropietaryClassName@)
                return @Attr_Type@ is abstract;
</AttrReadOnlyOperationHead>
```

Y el código generado será:

```
function get_attr_1(Self: access CCM_gralcomponent)
        return GralComponent_Types.attr_1_type is abstract;
procedure set_attr_1(Self: access CCM_gralcomponent;
        attribute: in GralComponent_Types.attr_1_type) is abstract;

function get_attr_2(Self: access CCM_gralcomponent)
        return GralComponent_Types.attr_2_type is abstract;
procedure set_attr_2(Self: access CCM_gralcomponent;
        attribute: in GralComponent_Types.attr_2_type) is abstract;
```

Otro tipo de operaciones de la interfaz monolítica serán las de acceso a las *facet executor interface*, las de configuración de las interfaces de activación y el método *configuration_complete*. Cuya generación se hace de forma similar y no merece la pena destacar.

Para la declaración de la interfaz *CCM_GralComponent_Context*, necesitaremos utilizar las siguientes plantillas:

```
<ContextInterfaceHead>
       type CCM_@ComponentName@_Context is interface
              and
@FatherComponent@_Exec.CCM_@FatherComponent@_Context;</ContextInterfaceHead>

<ContextInterfaceRef>
       type CCM_@ComponentName@_Context_Ref
              is access all CCM_@ComponentName@_Context'Class;
</ContextInterfaceRef>
```

Para implementar las operaciones *get_connection* de dicha clase utilizaremos las plantillas correspondientes. Podemos observar que el proceso que se lleva a cabo es bastante repetitivo y no merece la pena seguir detallándolo. Faltaría realizar la declaración de los *Home* así como las operaciones relacionadas con el mismo, utilizando las plantillas correspondientes que se encuentran definidas en el Anexo por si son de curiosidad para el lector.

5.3 GENERACIÓN DEL CÓDIGO ADA DEL CONTENEDOR DE UN COMPONENTE

Una vez comentado el proceso de generación de código, vamos a comentar los ficheros o paquetes que son generados por la herramienta para formar la estructura completa del mismo.

5.3.1 Paquetes generados correspondientes a un componente

Para cada componente, partiendo de su especificación IDL se generarán automáticamente los siguientes paquetes Ada:

- [NombreComponente]_Exec.ads: En este paquete se define el ejecutor del componente. En él se definen los esqueletos (clases raíces abstractas) y las interfaces que sirven de base para desarrollar las implementaciones de los componentes y de los home (siguiendo el estándar CCM). Este paquete es generado completamente de forma automática y es opaco por tanto al implementador.

- [NombreComponente]_Exec_Impl.ads/b: Paquete que representa la implemetación y que contiene como agregada la clase que realmente implementa la funcionalidad del componente. Su generación ha sido definida de forma estratégica para la separación del código de negocio a realizar por el implementador del código generado automáticamente. Este paquete es también generado automáticamente y el implementador no poseerá visibilidad sobre él.

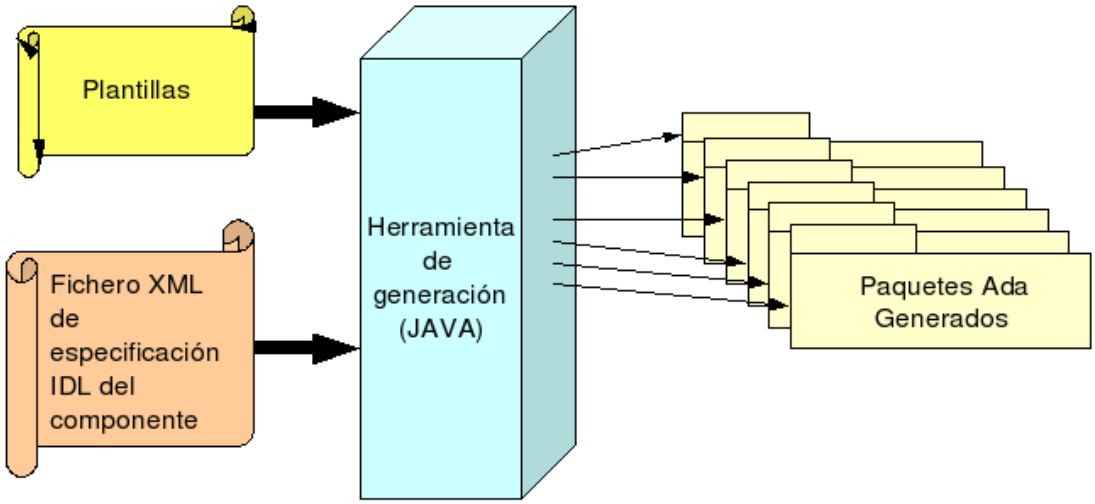

Figura 5.1: Proceso de generación de paquetes Ada.

- [NombreComponente]_Impl.ads/b: Paquete generado parcialmente de forma automática. En él se introduce el código de negocio que le da la funcionalidad deseada al componente, Entre el código de negocio que genera el implementador estará por ejemplo la implementación de las facets.

- [NombreComponente]_Wrapper.ads/b: Engloba todas las operaciones que el entorno de ejecución utiliza para gestionar los componentes. Este código es generado de forma completa automáticamente.

- [NombreComponente]_facet[NombreInterfaz].ads: Por cada facet ofertada por el componente se genera esta interfaz que heredará de la correspondiente *CCM_[NombreInterfaz]* definida en el ejecutor,

- [NombreComponente]_Types.ads/b: Todos los tipos definidos en el componente deberán ser almacenados en este paquete.

- [NombreComponente]_eqInt.ads: En este fichero se define la interfaz equivalente del componente, siguiendo la normativa CCM.

5.3.2 Paquetes generados correspondientes a los conectores

Los conectores como hemos explicado en el capítulo anterior están formados por dos fragmentos, que son implementados como componentes. Por tanto por cada fragmento se generarán los ficheros que hemos comentado correspondientes a un componente, salvo que en este caso el fichero *[NombreComponente]_Impl.adb* también será generado automáticamente. Esto requerirá una plantilla especial para este paquete, que podemos encontrar en el Anexo, siendo el resto de paquetes generado como si de un componente normal se tratara.

Por lo tanto, el fragmento *proxy* se generará como un componente normal, a excepción del paquete comentado, realizándose también a través de la especificación en formato XML del componente.

En cambio, el fragmento *servant*, además de la generación automática del código de negocio deberá incorporar los paquetes Ads/Adb correspondientes al *RCI* y al *Adapter* necesarios debido a la utilización de GLADE, que ya comentamos en el capítulo anterior. Por tanto, se requerirán las plantillas correspondientes para la generación de estos paquetes.

CAPÍTULO 6

JETFOLLOWER: EJEMPLO DE APLICACIÓN BASADA EN COMPONENTES

6.1 FUNCIONALIDAD Y ARQUITECTURA DE LA APLICACIÓN

Es un demostrador propuesto en el proyecto FRESCOR para poner de manifiesto la capacidad de la tecnología RT-CCM para dar soporte a aplicaciones basadas en componentes de tiempo real. Aprovechando de que se dispone de la especificación completa de los componentes que utiliza, así como la de la propia aplicación, se ha utilizado para probar la generación de código de las herramientas Ada-CCM que se han desarrollado.

En el proyecto, se ha utilizado sus especificaciones para generar el código Ada 2005 de los componentes y comprobar que este código es compilable. Aunque en algunos de los componentes como “DataBase” y “SoundGenerator” se han llegado a generar códigos ejecutables y se han probado, otros no ha sido posible llegar a ese nivel por falta de tiempo, y sobre todo por no estar disponible el dispositivo físico de *Tracker* y del robot de seguimiento.

La aplicación es un sistema que controla la posición de un conjunto de n servos que controlan la posición y orientación de un dispositivo mecánico hacia un objeto móvil cuya posición es determinada por un dispositivo externo al que denominamos *Tracker*. Ejemplos de aplicaciones a los que puede corresponder este sistema, es el de control de la orientación de una antena situado sobre un vehículo móvil hacia un satélite de comunicaciones fijo, o la orientación de un sistema antiaéreo hacia un avión que se mueva en el espacio.

La funcionalidad básica del sistema es el seguimiento continuo del objetivo, pero también bajo demanda del operador. El sistema registra en un *Logger* la trayectoria que realmente sigue el sistema (trayectoria de los servos), y así mismo genera señales acústicas cuando el sistema está operando o cuando los errores de posición de los servos superan unos ciertos umbrales preestablecidos.

En la figura 6.1 se muestra a través de un diagrama de clases UML la arquitectura software de la aplicación.

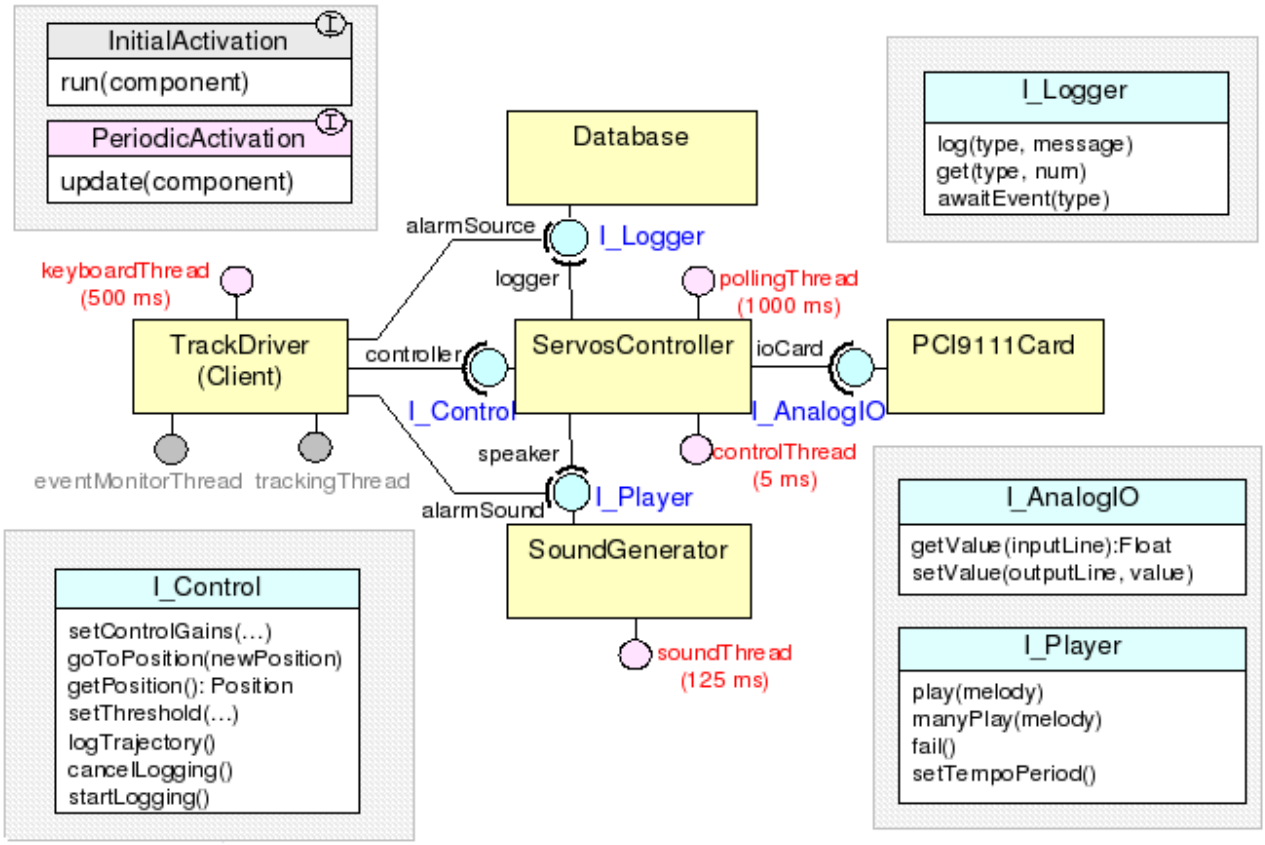

Figura 6.1: Arquitectura de la aplicación.

Para realizar esta aplicación, desde esta tecnología necesitamos de un componente cliente que será específico de esta aplicación al que hemos llamado *TrackDriver*. Necesitaremos un componente intermedio que implemente la interfaz *I_Control* al que hemos llamado *ServosController*, y tres componentes servidores finales *SoundGenerator*, *Database* y *PCI9111Card* que implementarán las interfaces *I_Player*, *I_Logger* e *I_AnalogIO* respectivamente.

El componente *TrackDriver* requiere utilizar operaciones definidas en las interfaces I_*Logger*, *I_Control* y *I_Player*, por lo tanto tendrá que conectarse a los componentes *Database*, *ServosController* y *SoundGenerator* respectivamente. Por su parte el componente *ServosController* deberá conectarse a los componentes *SoundGenerator*, *Database* y *PCI9111Card* para desarrollar su funcionalidad.

6.2 DESCRIPCIÓN DE LOS COMPONENTES QUE SE UTILIZAN

6.2.1 El componente TrackDriver

Este componente representa al cliente que controla la aplicación. Como corresponde a su naturaleza de cliente no ofrece ninguna interfaz pública, y la operaciones que se especifican encapsulan actividades internas (privadas) de respuesta a eventos del *timer* y de los periféricos que gestiona.

Es un componente con capacidad de mantener varias líneas de flujo de control concurrentemente:

- Atiende el teclado, y de acuerdo con el carácter que se pulsa realiza una de las siguientes operaciones:
  - 'T': Controla los servos siguiendo la posición del objetivo.
  - 'S': Para los servos a su posición segura.
  - 'M': Habilita sonidos mientras que exista movimiento.
  - 'Q': Elimina sonido mientras que exista movimiento.
  - 'F': Finaliza la aplicación.
  - 'A': Registra en el Logger y visualiza en el monitor la trayectoria de los servos.
  - 'C': Cancela la visualización de las trayectorias que se registran en el Logger.

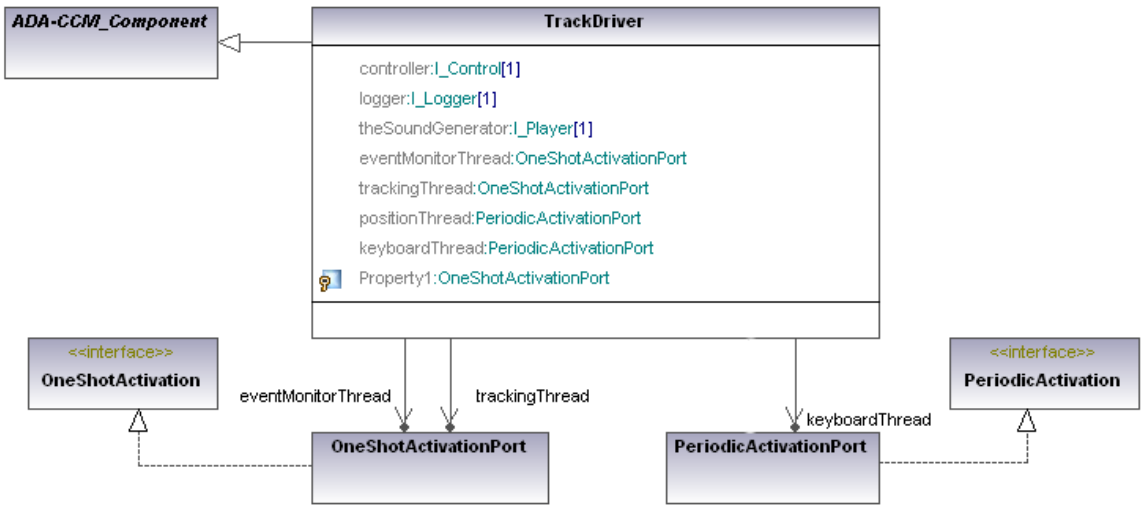

Figura 6.2: Descripción UML del componente TrackDriver.

Esta línea de control se construye utilizando el *thread* que se inicia a través del puerto *keyboardThread* y que en cada invocación lee el teclado, y en función de la tecla que encuentre pulsada, ejecuta una de las acciones que están especificadas. Como el sistema no puede quedarse a la espera de un dispositivo externo (ya que ello supondría el bloqueo de todos los *threads* de la aplicación), este *thread* se queda bloqueado en un recurso compartido, y utiliza la activación periódica que proporciona el puerto *keyboardThread* para supervisar cada 500 ms el estado del teclado.

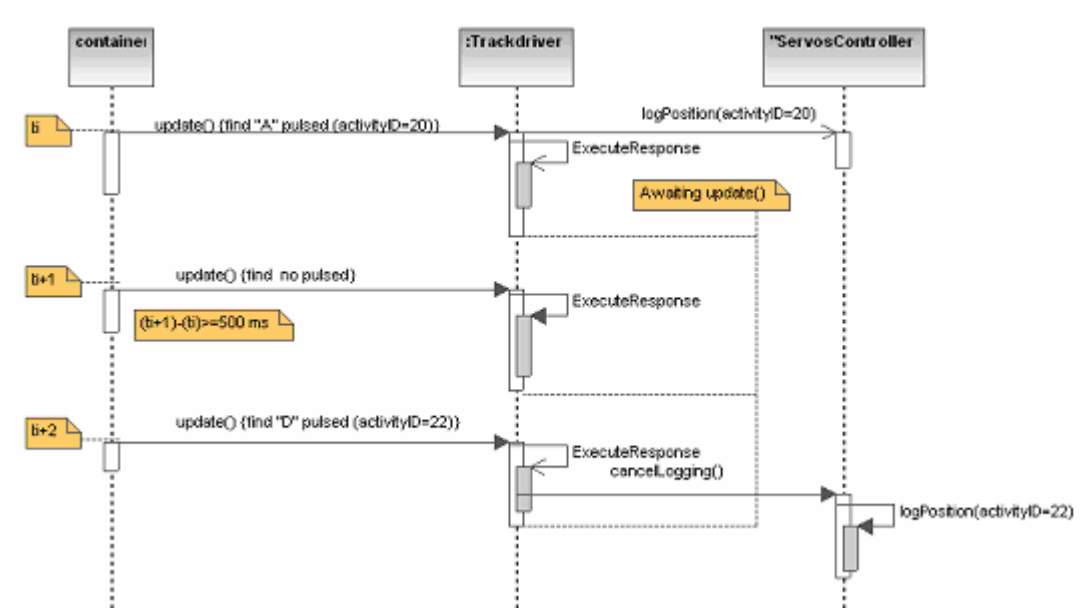

Figura 6.3: Diagrama de secuencias del componente TrackDriver.

- Recibe del dispositivo externo *trackingSystem* posiciones del objetivo que deben ser seguidas. Esta actividad se realiza haciendo uso del *thread* de activación *OneShot* que se obtiene del puerto de *trackingThread*. Este *thread* permanece a la espera de que el *trackingSystem* proporcione una posición del objetivo, y cuando la recibe planifica trayectoria que deben seguir los servos para su seguimiento mediante el procedimiento *planTrajectory*.

- Visualiza en el monitor las alarmas y errores que se registren en el *Logger DataBase*, y en el caso de que se encuentre habilitada, las coordenadas de la trayectoria de los servos que se registran. Esta actividad se implementa con el *thread* que proporciona el entono a través del puerto *OneShot eventMonitorThread*. Este *thread* se suspende en el *Logger* y a cada registro que se produzca en él, retorna el nuevo registro.

6.2.2 El componente ServosController

Este componente realiza las siguientes funciones:

- Implementa el algoritmo de control PID (Proporcional integral derivativo) para el conjunto de servos del sistema. Cuando al sistema se le propone una nueva posición de los servos mediante la operación *goToPosition*, genera una trayectoria por interpolación lineal y a la máxima velocidad entre la posición actual y la nueva, y controla los servos para que la sigan.
- Registra en el *Logger* los errores de posición de los servos que superan un cierto umbral establecido mediante el procedimiento *setThreshold*.
- Genera un sonido en el dispositivo *soundGenerator* cuando el sistema se está moviendo, y así mismo genera un sonido diferenciado cuando el error de posición de los servos supera el valor umbral establecido.
- Bajo demanda por el procedimiento *startLogging* se habilita un modo de monitorización que registra periódicamente en el *Logger* la posición de los servos, y bajo demanda *cancelLogging* lo cancela.
- El controlador almacena en una cola interna los puntos de la trayectoria que recibe mediante sucesivas invocaciones del procedimiento *goToPosition*. El controlador cada 5 ms obtiene la posición concreta de los servos obtenidas por interpolación lineal de la trayectoria almacenada.

Este componente ofrece la interfaz *I_Control* mediante la faceta *theControlPort*, a través de la cual recibirá las invocaciones de operaciones de control que le pida el componente *TrackDriver* conectado al mismo. Además de esto, este componente posee dos líneas de control que deben poder ejecutarse concurrentemente:

- Debido a que el componente cada 5 milisegundos debe calcular la consigna concreta, debe ejecutar el algoritmo PID de control y si el error de posición supera los umbrales, registrarlo en el *Logger*. Requerirá un *thread* periódico que recibe del entorno a través del puerto de activación periódica *controlThread*, para realizar esta actividad.
- Cuando a través de la invocación del procedimiento *startTrayectory* se demanda al controlador que pase al modo de monitorización del registro, requiere del entorno a través del puerto *pollingThread* un nuevo *thread*, que registra cada segundo las posiciones actuales en el *Logger*.

El estado del controlador se implementa mediante la clase *ServosControllerState*, donde se encontrará la información completa de que dispone el componente. Esta información es accesible desde cualquier puerto que implemente el controlador.

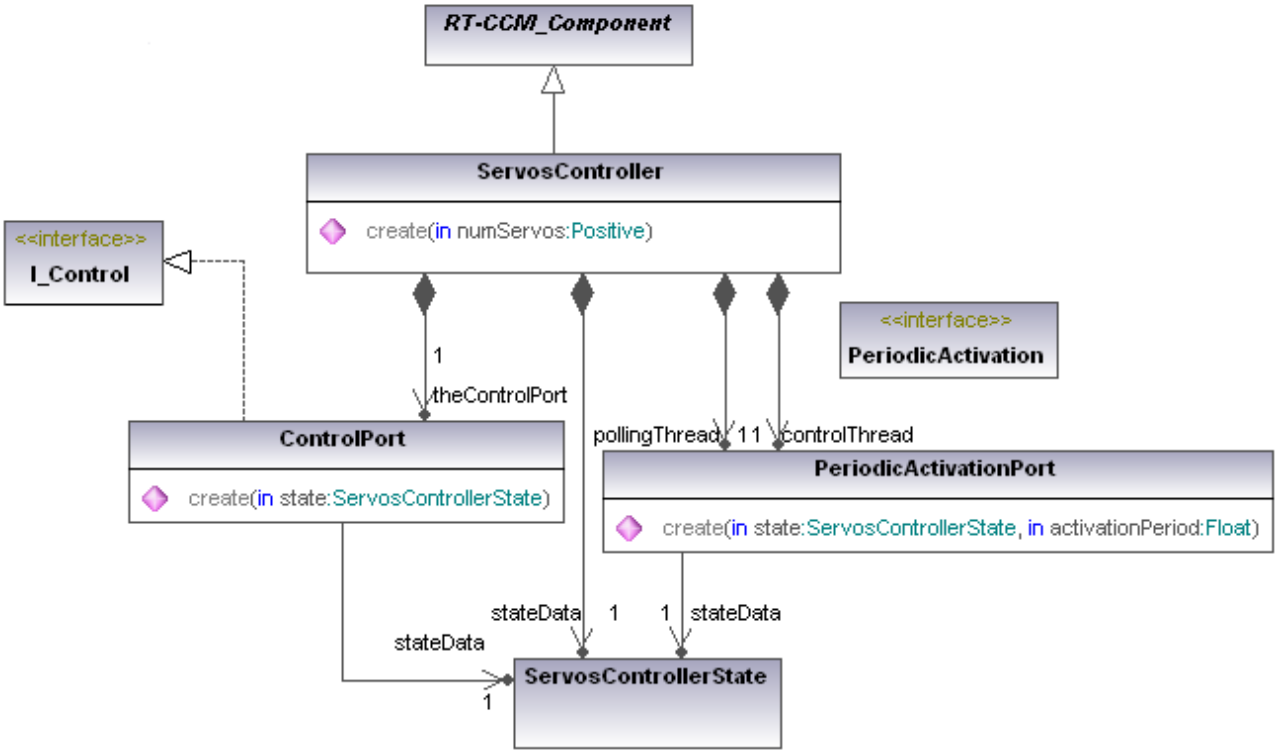

Figura 6.4: Diagrama UML de las clases que forman el componente ServosController.

6.2.2.1 La interfaz I_Control

Esta Interfaz define los servicios para el control de un conjunto de n servos. El componente ServosController implementa esta interfaz para desarrollar su funcionalidad.

Las operaciones que ofrece esta interfaz son:

- *setControlParamSet(paramSet: in ControlParamSet; servo: in Natural):* Establece los parámetros que se utilizan en el control de un servo. El parámetro servo indica el servo al que corresponden los parámetros.
- *setTrajectoryParamSet(paramSet: in TrayectoryParamSet; servo: in Natural):* Establece los parámetros con los que se planifica la trayectoria para pasar de la posición actual a una posición que se le establece como consigna.
- *putControlPeriod(thePeriod: in Float)*: Pasa información del periodo con el que debe operar el algoritmo de control. Esta operación no establece el periodo de control.
- *goToPosition(newPosition: in Position)*: Establece una nueva posición a la que debe conducirse los servos desde la situación actual. Para ello se realiza una interpolación lineal de acuerdo con los parámetros de trayectoria.

- *getCurrentPosition()*: Retorna la posición actual de los servos.
- *startLogging()*: Establece el modo de de registro de la trayectoria. Tras ser invocado este procedimiento el sistema registra en el *Logger* como *warning* la posición de los servos cada segundo.
- *cancelLogging()*: Deshabilita el modo de monitorización de trayectoria en el *Logger*.

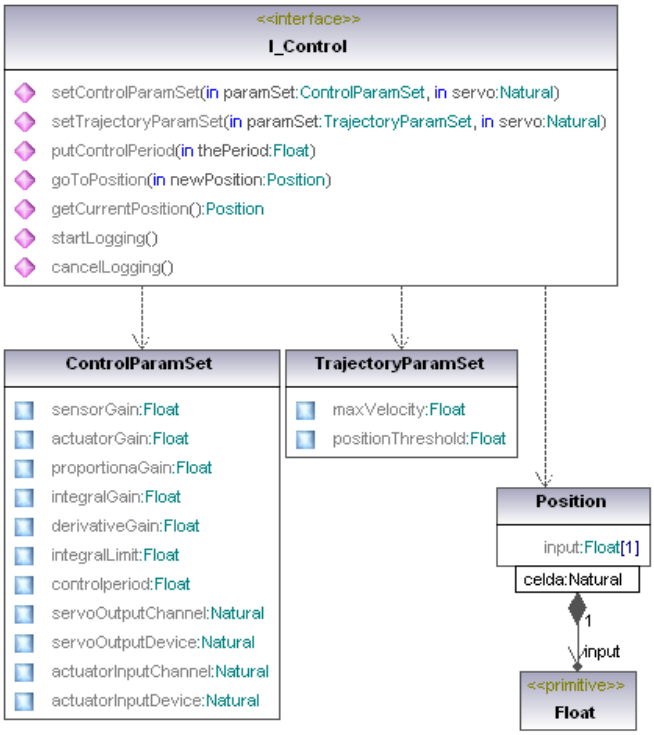

Figura 6.5: Diagrama UML de la interfaz I_Control.

La interfaz *I_Control* necesita manejar una estructura para introducir los parámetros para el control del servo (*ControlParamSet*) y otra para establecer los parámetros con los que se planificará la trayectoria (*TrayectoryParamSet*). A continuación se describen dichas estructuras:

- La estructura *ControlParamSet*:
  - *SensorGain*: Ganacia del sensor. La posición actual del sensor se obtiene multiplicando el valor a la salida del A/D por esta ganancia.
  - *actuatorGain*: Ganacia del actuador. Es el valor por el que hay que multiplicar la salida del conversor D/A para obtener la señal de entrada del servo.
  - *proportionalGain*: Ganancia proporcional del algoritmo PID.

- *integralGain*: Ganancia integral del algoritmo PID.
- *derivativeGain*: Ganancia de la derivada del algoritmo PID.
- *integralLimit*: Valor al que se limita la componente integral.
- *controlPeriod*: Valor del periodo de control.
- *servoOutputChannel*: ID del canal del A/D de lectura de la salida del servo.
- *servoOutputDevice*: ID de la Tarjeta IO con el A/D de lectura de la salida del servo.
- *actuatorInputChannel*: ID del canal del D/A que establece la entrada del actuador.
- *actuatorInputDevice*: ID de la Tarjeta IO con el D/A que establece la entrada del actuador.

- La estructura *TrajectoryParameterSet*:
  - *maxVelocity*: Máxima velocidad que se puede requerir en el movimiento de la salida del servo.
  - *positionThreshold*: Umbral en el error de posición de un servo que debe ser considerado como una alarma.

6.2.3 El componente SoundGenerator

Es un componente servidor que permite generar notas musicales o melodías utilizando el altavoz existente en el PC. Ofrece la interfaz *I_Player* que permite controlar los sonidos que genera. Es un componente activo que requiere ejecutar código en cada cambio de nota.

La actividad que ejecuta la obtiene de la activación periódica que se obtiene a través del puerto *changeNoteThread*.

Este componente ofrece la interfaz *I_Player* a través de la faceta *theSoundPlayer*. En esta aplicación dos componentes (*TrackDriver* y *ServosController*) requerirán de esta implementación a través de sus respectivos receptáculos, con el fin de generar sonidos.

6.2.3.1 La interfaz I_Player

Es implementada por este componente, y ofertada mediante la faceta *changeNoteThread* al exterior. Contiene las siguientes operaciones que le dan la funcionalidad al componente:

- *setTimeBase(newTempo: in Tempo)*: Establece el tiempo musical con el que se tocarán las notas. Por tanto, la elección del tiempo condicionará la velocidad en que se reproducirán las notas modificando el marcapasos.

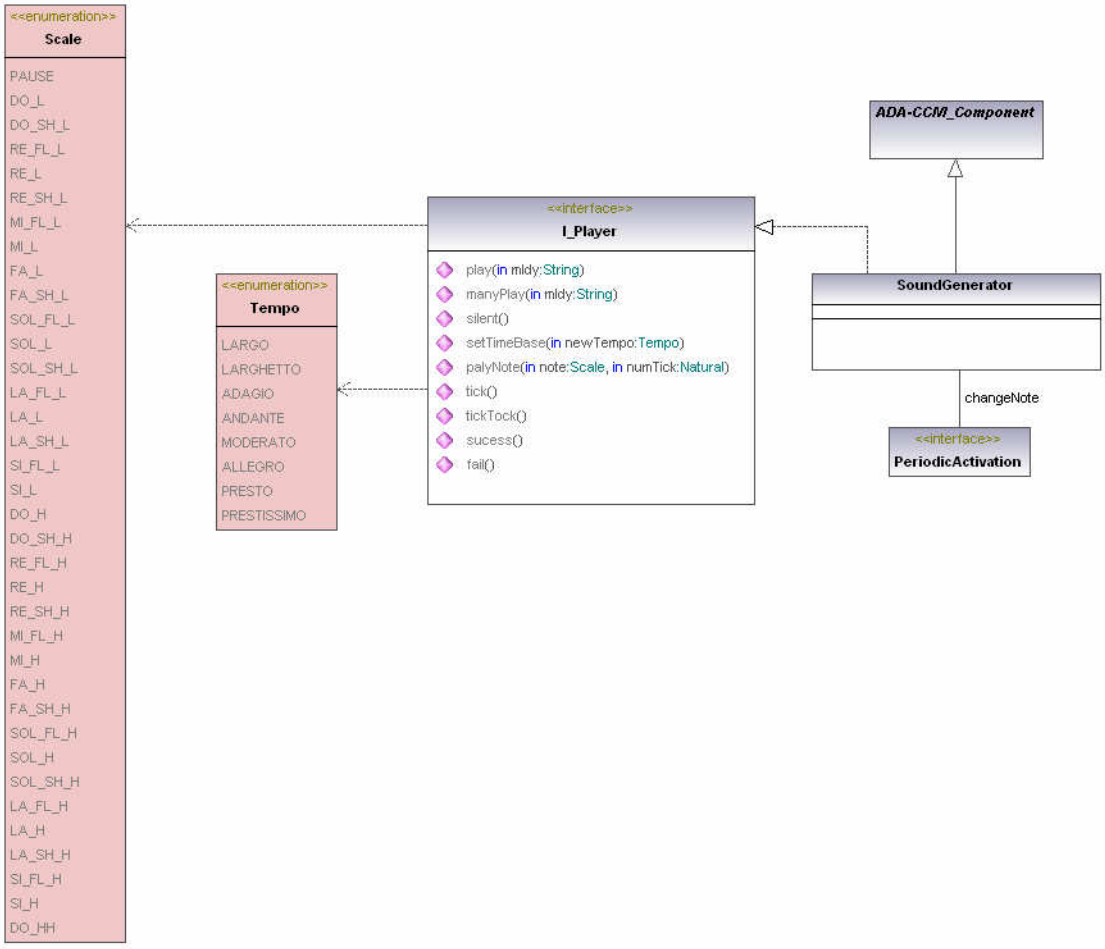

Figura 6.6: Diagrama UML de la interfaz I_Player.

- *playNote(note: in Scale; numTick: in Natural)*: Tocará una nota musical durante un número de *ticks* proporcionado.
- *play(mldy: in String)*: Interpretará la melodía o conjunto de notas musicales almacenada en el fichero XML cuyo path se le pasa por parámetro.
- *manyPlay(mldy: in String)*: Interpretará la melodía repetidamente hasta que se indique lo contrario.
- *silent()*: Parará la reproducción de sonidos que se estén reproduciendo en el momento de ejecutarse esta operación.
- *tick(), tickTock(), success() y fail()*: Son sonidos preconfigurados que pueden ser reproducidos ejecutando los procedimientos correspondientes.

Se definen en esta interfaz dos tipos enumerados. El tipo *Tempo* define los diferentes tiempos musicales con que puede reproducirse una melodia, y el tipo *Scale* define las diferentes notas musicales que pueden reproducirse con el SoundGenerator.

6.2.4 El componente Database

Este componente es un componente servidor que representa a un objeto pasivo que permite el acceso a un registro de eventos que implementa la interfaz *I_Logger* que sirve para registrar de forma persistente los eventos que se generan en la aplicación. Este componente por tanto ofrecerá una faceta de nombre *theLogger* a través de la cual se podrán acceder a los servicios que proporciona.

6.2.4.1 La interfaz I_Logger

Esta interfaz proporciona las operaciones de acceso y escritura en el *Logger*:

- *log(type: in EventType; mssg: in Unbounded_String):* almacena en el *Logger* el registro correspondiente. Los eventos se almacenan en una lista *EventList*, asociando a cada registro la fecha, el tipo de evento y el mensaje asociado al mismo.

- *get(option: in RequiringEventOption; numEvent : in Natural):* retorna una lista de *numEvent* eventos del tipo que se solicita registrados en el *Logger*.

- *awaitEvent(option: in RequirinEventOption)*: Operación que se queda a la espera de la llegada de un evento al *Logger*. Es ejecutada por el *thread* de un componente conectado al componente que implemente esta interfaz (por ejemplo el componente *TrackDriver* mediante *eventMonitorThread*).

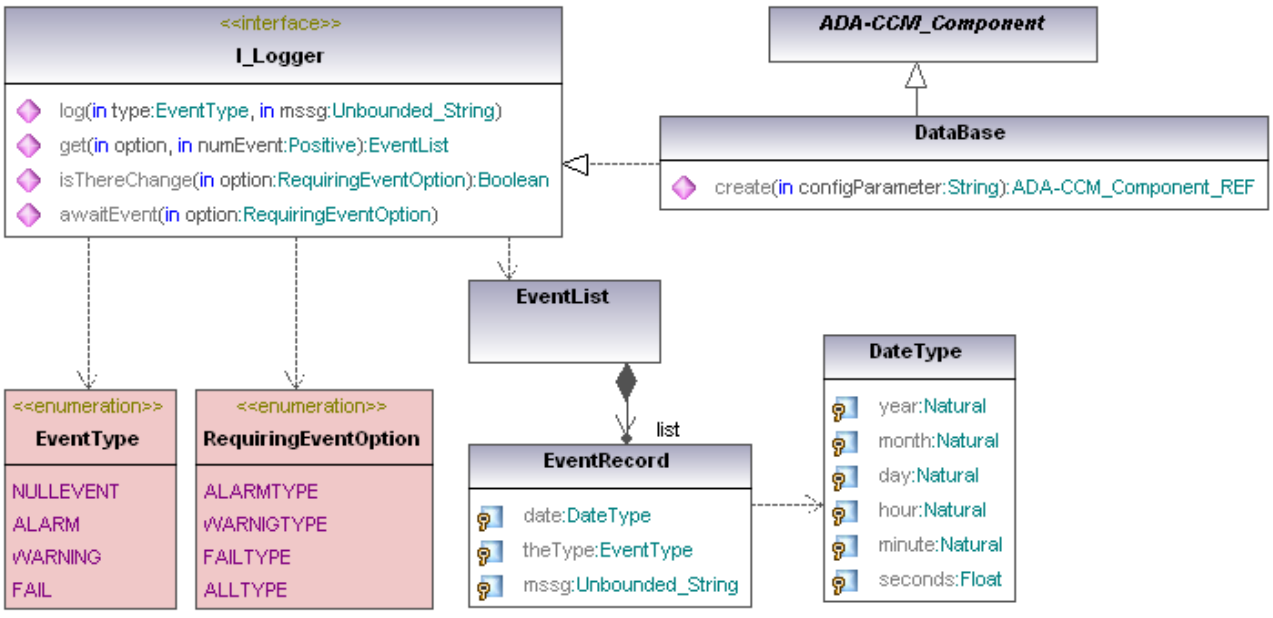

Figura 6.7: Diagrama UML de la interfaz I_Logger.

La interfaz *I_Logger* permite registrar tres tipos de eventos, que en el contexto de la aplicación tienen el siguiente significado:

- ALARM: Se ha producido un fallo funcional, ya que el error de posición supera los umbrales establecidos.

- WARNING: Se ha generado una nueva monitorización de la trayectoria.
- FAIL: En cualquier componente se ha producido una excepción no tratada en su lógica interna.

6.2.5 El componente PCI9111Card

Es un componente servidor que implementa el acceso a la tarjeta de IO analógica que contiene los convertidores D/A y A/D con los que respectivamente se controla los servos y se leen sus posiciones. Es un componente pasivo que simplemente sirven las invocaciones de lectura y establecimiento de las señales analógicas.

Ofrece por tanto la faceta *theIOCard* a través de la cual otros componentes podrán acceder a los servicios que proporciona.

Las operaciones que ofrece *I_AnalogIO* son para la lectura y establecimiento de las señales analógicas de control de los servos.

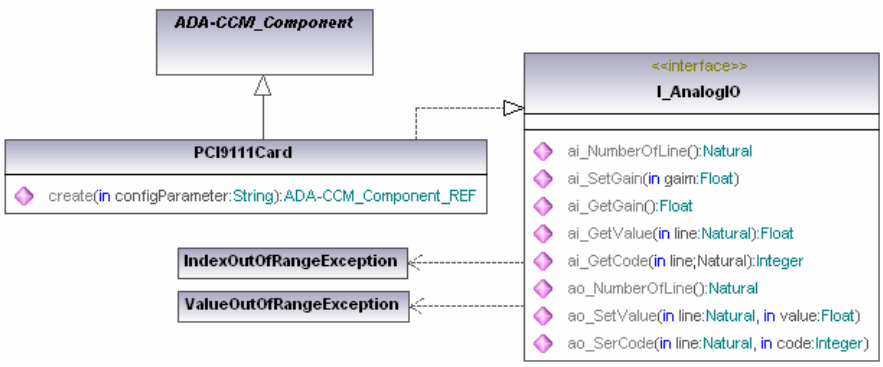

Figura 6.8: Diagrama UML de la interfaz I_AnalogIO.

6.3 DESCRIPCIÓN DE LA PLATAFORMA

Para realizar la aplicación *jetFollower* se necesita la descripción de la plataforma en la que se va a ejecutar. La descripción de la plataforma para esta aplicación presentará la siguiente información:

- Información de los nodos que se disponen para la ejecución de la aplicación. En nuestro caso tendremos dos nodos: central y remoto. Dentro de la información que concierne a cada uno de los nodos, tenemos los recursos asociados y disponibles que presenta. En el caso de la plataforma disponible para la ejecución del demostrador, se dispone de una plataforma con dos nodos y en cada uno de ellos con una partición. Como información de los recursos de los nodos estará el sistema operativo sobre el que trabajan, que en el caso de los dos nodos será MaRTE OS. También

formará parte de la información relativa al nodo la IP del mismo y la red a la que pertenece.

- Información de la redes de interconexión entre los nodos. Incluye el nombre de los nodos que interconecta y el protocolo que se utiliza. En este caso, el protocolo disponible será TCP.

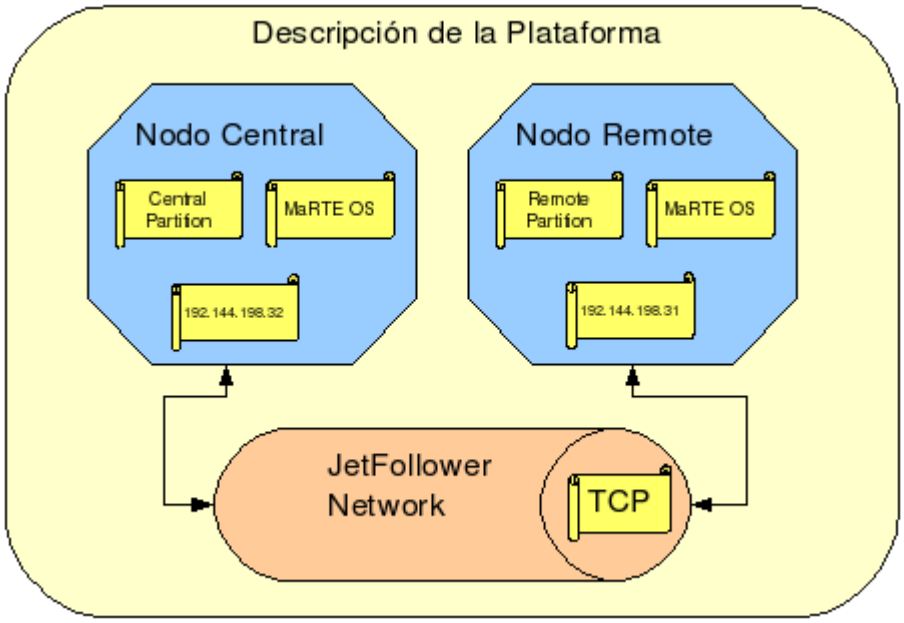

Figura 6.9: Esquema de la información contenida en la descripción de la plataforma.

En la siguiente tabla se muestra un fragmento del fichero que describe la plataforma. Incluye la información del nodo remoto y de la conexión entre el nodo remoto y el central:

```
<node name="remote" label="Operator Node">
        <resource resourceType="OS" name="remote_OS">
                <property name="type" kind="ATTRIBUTE">
                        <value>MaRTE_OS</value>
                </property>
        </resource>
        <resource resourceType="partition" name="remote_partition">
                <property name="name" kind="ATTRIBUTE">
                        <value>remote_partition</value>
                </property>
        </resource>
        <resource resourceType="machine" name="remote_ip">
                <property name="ip" kind="ATTRIBUTE">
                        <value>192.144.198.31</value>
                </property>
        </resource>
        <connection>jetfollower_network</connection>
</node>

<interconnect name="jetfollower_network" label="communications network">
        <connect>central</connect>
        <connect>remote</connect>
        <resource resourceType="protocol" name="theProtocol">
                <property name="type" kind="ATTRIBUTE">
                        <value>tcp</value>
                </property>
        </resource>
</interconnect>
```

6.4 PLAN DE DESPLIEGUE DE LA APLICACIÓN

A partir de la descripción de la plataforma, la especificación funcional y de tiempo real de la aplicación, y la información de los componentes disponibles en la base de datos, el agente de diseño generará un fichero con el plan de despliegue de la aplicación. La información contenida en el plan de despliegue se comenta a continuación:

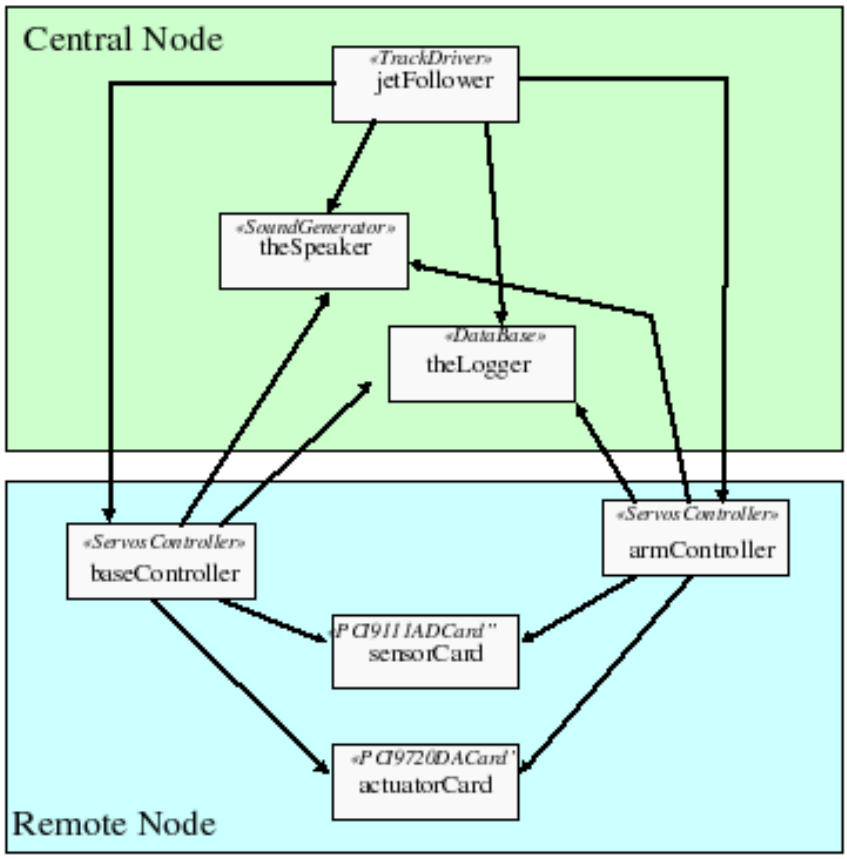

Figura 6.10: Ubicación de las instancias de los componentes y sus conexiones.

- Instancias que intervienen en la aplicación: Se requerirá una instancia (*jetFollower*) del componente *TrackDriver*, para la realizar la interfaz de usuario de la aplicación. Para generar los sonidos que se requieren en la aplicación se necesitará una instancia (*theSpeaker*) del componente *SoundGenerator*, y para registrar los eventos que se producen tenemos que crear una instancia (*theLogger*) del componente *Database*. Por otra parte se necesitará controlar dos servos, la base y el brazo móvil, por lo que se requerirán otras dos instancias (*baseController* y *armController*) del componente *ServosController*. Ambos controladores necesitan de dos instancias (*sensorCard* y *actuatorCard*) del componente *PCI9111Card* a través de las cuales podrán leer la posición (*sensorCard*) del servo correspondiente y posicionarlo (*actuatorCard*).

- Localización de los nodos en que se ejecutarán las instancias: Debido a la localización remota de los servos cuyo control realizamos, esta aplicación

estará distribuida en dos nodos. En el nodo central se ejecutarán las instancias de los componentes *TrackDriver*, *SoundGenerator* y *Database*. Y en el nodo remoto las instancias *baseController*, *armController*, *sensorCard* y *actuatorCard* correspondientes a los componentes *ServosController* y *PCI9111Card*.

- Conexiones a realizar entre los puertos de instancias que lo requieran: La instancia *JetFollower* requiere de la instancia *theSpeaker* para generar sonidos y de la instancia *theLogger* para monitorizar los eventos. A su vez cada uno de los controladores requerirán de la instancia *theSpeaker* para generar sonidos y de la instancia *theLogger* para almacenar los eventos que se vayan produciendo. Ambos controladores requieren también la conexión con las instancias *sensorCard* y *actuatorCard* para realizar su funcionalidad.

- Asignación de las propiedades de configuración: Se configurarán los puertos de activación que contengan los componentes con los parámetros de planificación con los que se establece la ejecución de los *threads* que se requieran, así como los periodos de las activaciones periódicas.

A continuación se muestra un pequeño fragmento del fichero del plan de despliegue:

```
<instance name="baseController" node="host" implementation="ServosController_impl"
              source="Components/ServosController">

        <configProperty name="controlThread">
               <value>
                      <periodicActivation_Property period ="0.005" schedAttrId ="2"/>
           </value>
         </configProperty>
         <configProperty name="pollingThread">
               <value>
                      <periodicActivation_Property period ="1" schedAttrId ="3"/>
           </value>
         </configProperty>

[..]

<connection name="baseControllerTotheLogger">
        <internalEndpoint instance="baseController" portName="logger"/>
        <internalEndpoint instance="theLogger" portName="iLogger"/>
</connection>
<connection name="baseControllerTotheSpeaker">
        <internalEndpoint instance="baseController" portName="speaker"/>
        <internalEndpoint instance="theSpeaker" portName="iPlayer"/>
</connection>
<connection name="baseControllerTosensorCard">
        <internalEndpoint instance="baseController" portName="ioCard"/>
        <internalEndpoint instance="sensorCard" portName="iAnalogIO"/>
</connection>
<connection name="baseControllerToactuatorCard">
        <internalEndpoint instance="baseController" portName="ioCard"/>
        <internalEndpoint instance="actuatorCard" portName="iAnalogIO"/>
</connection>
```

Primero encontramos un pequeño fragmento en el que se define la instancia *baseController* del componente *ServosController* y el nodo en que se localiza. En el fragmento que se muestra se pueden ver la definición de las propiedades de configuración de las activaciones periódicas (parámetro de planificación y periodo) relativas a dicha instancia. El segundo fragmento que vemos describe las conexiones de dicha instancia con instancias de otros componentes que requiere.

6.5 FICHEROS QUE CONSTITUYEN LA APLICACIÓN

En el proceso de desarrollo de los componentes previo al diseño de esta aplicación, se habrá generado código de diversos componentes mediante la herramienta de generación de código en base a su especificación IDL. Se completará el código de los mismos con el código de negocio que incorporará el implementador y se compilarán con GNAT, creándose los .o que se almacenarán en el repositorio para la utilización de los mismos en esta u otras aplicaciones.

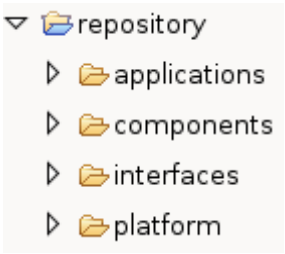

Figura 6.11: Estructura del repositorio.

Por tanto, cuando iniciamos el diseño de esta aplicación poseemos en el repositorio de una serie de componentes que podemos reutilizar. En nuestro caso, nos interesa reutilizar los componentes SoundGenerator, Database, ServosController y PCI9111Card que forman parte del repositorio.

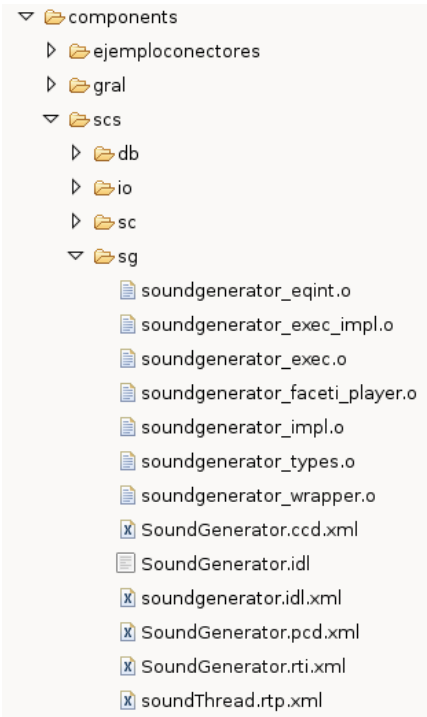

Figura 6.12: Ficheros en el repositorio del componente SoundGenerator.

Además de los ficheros .o del componente, encontraremos en el repositorio la siguiente información de los mismos:

- Ficheros de especificación IDL del componente: *[NombreComponente].idl* y *[NombreComponente].idl.xml*.
- Fichero de descripción del componente de acuerdo al D&C: *[NombreComponente].ccd.xml*. Es necesario ya debido a que la tecnología Ada-CCM sigue este estándar.
- Descripción de empaquetamiento del componente de acuerdo al D&C: *[NombreComponente].pcd.xml*. Describe el componente y toda la información que un usuario requiere para hacer uso de él.
- Modelo de tiempo real del componente: *[NombreComponente].rti.xml*. Éste fichero es propio de la tecnología Ada-CCM, realizando una extensión de D&C para el soporte de aplicaciones de tiempo real. En él se incluyen todos los parámetros temporales relacionados con el comportamiento del componente.

En el caso de esta aplicación, se requiere la creación de un componente específico de la aplicación. Nos referimos al componente *TrackDriver*. Por tanto, el diseñador debe generar su código a través de la herramienta de generación de código e implementar el código de negocio del mismo. Una vez desarrollado el componente se añade al repositorio.

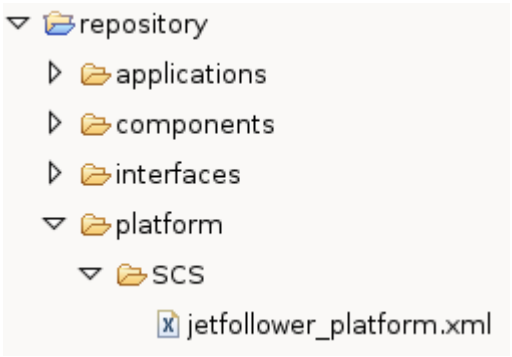

Figura 6.13: Localización del fichero que define la plataforma.

La descripción de la plataforma en que se ejecutará la aplicación *JetFollower* se encontrará en el repositorio en un fichero XML. Este fichero (*jetFollower_platform.xml*) contiene los nodos y recursos que posee en cada uno (Sistema operativo, nombre de las particiones, IP de la máquina y nombre de la red a la que pertenece), y la información relativa a las conexiones entre nodos (nodos que intervienen y protocolo utilizado).

En base a estos ficheros (Descripción de la plataforma, componentes disponibles e información acerca de ellos contenida en el repositorio), el Diseñador realiza el plan de despliegue de la aplicación, y en el caso de esta tecnología lo realizará también en un fichero XML.

Figura 6.14: Localización del plan de despliegue en el repositorio.

Una vez que se tiene el plan de despliegue, se realiza la generación del código de los conectores mediante la herramienta de despliegue. Como se comentó en el capitulo 4, los conectores se generan como si de componentes se trataran, con lo que la estructura de ficheros que se genera será similar.

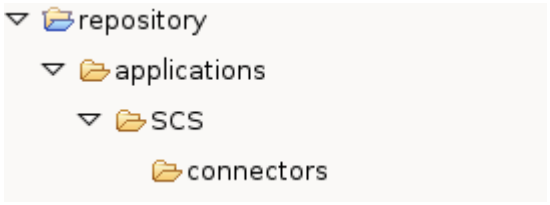

Figura 6.15: Localización del código de los conectores.

Como utilizamos GLADE como *middleware* de comunicaciones, por cada fragmento *servant* de conector de un componente se generará un *RCI* y se añadirá a la lista de RCIs que existen en la aplicación. En base a esta lista y el plan de despliegue se generará el fichero de configuración de GLADE que se requiere.

La herramienta de despliegue generará un plan de despliegue parcial por cada nodo en base al plan de despliegue que se generó. A partir de los cuales se generarán los códigos *main* de cada una de las particiones.

Finalmente, la herramienta de despliegue creará los ejecutables de las dos particiones mediante *gnatdist*, a partir del fichero de configuración de GLADE, del código *main* de cada partición y de los .o de los paquetes correspondientes a los componentes implicados. *Gnatdist* incluirá en los ejecutables toda la información necesaria para la comunicación entre nodos que se realiza a través de los conectores de los componentes.

Ya no se crearán más ficheros. La herramienta de lanzamiento leerá los ejecutables de los *main* de cada partición, y a partir de la información de despliegue que se tiene, los mandará a los nodos en que se ejecuta esta aplicación y lanzará la aplicación.

CAPÍTULO 7

CONCLUSIONES Y LÍNEAS FUTURAS DE TRABAJO

Se ha diseñado y desarrollado la herramienta de generación del código automático de los componentes y de los conectores para la tecnología Ada-CCM. Con ella, se ha completado la fase de desarrollo de los componentes prevista para sistemas embebidos que propone la tecnología Ada-CCM.

El código generado por esta herramienta ha sido implementado para la nueva especificación Ada 2005, cuyas nuevas características se han demostrado que son adecuadas para el desarrollo de aplicaciones basadas en componentes.

La tecnología Ada-CCM se encuentra actualmente en fase inicial, a la espera de infraestructura de soporte que pasamos a comentar:

- Las pruebas se han realizado en máquinas sobre plataforma Linux. El entorno de ejecución no es pues de tiempo real estricto. Esta tecnología necesita un sistema operativo de tiempo real, que soporte la ejecución de aplicaciones Ada 2005 y sus extensiones de tiempo real.

- Como hemos comentado en el capítulo destinado a los conectores, hemos utilizado GLADE para la realización de aplicaciones distribuidas. La utilización de GLADE no permite satisfacer los requisitos temporales propios de las tecnologías de tiempo real.

Actualmente en el grupo de computadores y tiempo real de la Universidad de Cantabria, se está trabajando para proporcionar el soporte necesario para el desarrollo de esta tecnología. Para ello:

- Se está realizando una nueva versión del sistema operativo MaRTE que proporcionará una plataforma de tiempo real con soporte de ejecución de aplicaciones Ada 2005.

- Se está desarrollando actualmente un *middleware* de tiempo real compatible con Ada 2005, que permitirá la ejecución de aplicaciones basadas en Ada 2005 sobre plataforma MaRTE distribuida.

Como línea de trabajo futuro está la implementación de un entorno integrado de desarrollo propio de esta tecnología, que facilitará la labor del diseñador, proporcionándole una herramienta mediante la cual podrá realizar de forma visual la elección de los componentes, su configuración, la composición de los mismos y la colocación en los diferentes nodos disponibles en la plataforma. El implementador por su parte no poseerá ninguna visibilidad ajena al código de negocio. Actualmente, se ha reducido la visibilidad al máximo del implementador, pero sigue teniendo acceso a código generado automáticamente que no requiere conocer.

REFERENCIAS BIBLIOGRÁFICAS

[1] *F. Bachmann, L. Bass, K. C. Wallnau y otros: "Volume I: Market Assessment of Component-Based Software Engineering", "Volume II: Technical Concepts of Component-Based Software Engineering", "Volume III: A Technology for Predictable Assembly from Certifiable Components", report CMU/SEI-2003-TR-009, Software Engineering Institute, Carnegie Mellon University, 2003.*

[2] *Ivica Crnkovic and Magnus Larsson: "Building reliable component–based Sosftware Systems" Artech House Publishers, 2002.*

[3] *Natis, Y.; Pezzini, M. & Schulte, R. "Middleware Deployment Trends: Survey of Real-World Enterprise Applications." Gartner Group Strategic Analysis Report. Stamford, CT: Gartner Group, April 1999.*

[4] *Williams, J. "Distributed Components and the Internet." Application Development Trends (December 1999): 79-80.*

[5] *Clements, P. & Northrop, L. Software Product Lines: Practices and Patterns. Boston, MA: Addison-Wesley, 2002.*

[6] *Stafford, J. & McGregor, J. "Issues in Predicting the Reliability of Composed Components." http://www.sei.cmu.edu/pacc/CBSE5/StaffordMcGregor-cbse5.pdf (2002).*

[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 projects: "FRESCOR (Framework for Real-time embedded Systems based on Contracts)".* http://www.frescor.org.

[10] *Lightweight CORBA Component Model - OMG draft adopted specification, Object Management Group, 2003.*

[11] *OMG: "Deployment and Configuration of Component-based Distributed Applications Specification" Draft Adopted Specification ptc/03-07-02. June 2003.* http://www.omg.org.

[12] *Ada Language Mapping Specification - Version 1.2: October 2001 by OMG.*

[13] *L. Pautet and S. Tardieu, "Inside the Distributed Systems Annex". Proceeding of the Intl. Conf. on Reliable Software Technologies, Ada-Europe'98, Uppsala, Sweden, in LNCS 1411, Springer, pp. 65–77, June 1998.*

[14] *Ada-Core Technologies, http://* www.gnat.com/

ANEXO: PLANTILLAS UTILIZADAS POR LAS HERRAMIENTAS DE GENERACIÓN DE CÓDIGO

Se ha querido incorporar este Anexo a modo de documentación de las plantillas que se han utilizado en las herramientas de generación de código automático o por si es de interés del lector para aclarar el funcionamiento de dichas herramientas.

PLANTILLA PARA LA GENERACIÓN DE LAS INTERFACES

Fichero *InterfaceTemplate.txt:*

```
<Head>
--
*******************************************************************************
******
-- Project: RT-CCM
-- Group: CTR -Universidad de Cantabria
--
-- Ada specification of the types belong to @ElemType@ : @ElemName@
-- Date: @Date@
--
*******************************************************************************
******
</Head>

<With>
with @Package@;</With>

<Use>
use @Package@;</Use>

<InterfaceHead>
package @InterfaceName@ is

        type Face is interface;
```

```
        type Ref is access all Face'Class;</InterfaceHead>

<InterfaceWithInheritanceHead>
package @InterfaceName@ is

        type Face is interface and </InterfaceWithInheritanceHead>

<InterfaceWithInheritanceBase>                      @BaseInterface@ and
</InterfaceWithInheritanceBase>

<InterfaceWithInheritanceBaseLast>                  @BaseInterface@;
        type Ref is access all Face'Class;</InterfaceWithInheritanceBaseLast>


<EnumTypeHead>
        type @TypeName@ is (
</EnumTypeHead>

<EnumValue>           @EnumValue@,
</EnumValue>

<LastEnumValue>              @EnumValue@);
</LastEnumValue>

<SimpleTypeHead>
        type @TypeName@ is new @IdlType@;</SimpleTypeHead>

<SequenceTypeHead>
        package @SequenceName@_pkg is new
                Ada.Containers.Vectors(Natural,@SequenceType@);
        subtype @SequenceName@ is @SequenceName@_pkg.Vector;</SequenceTypeHead>

<StringBoundedTypeHead>
        package @StringName@_pkg is new
                Ada.Strings.Bounded.Generic_Bounded_Length(@Length@);
        type @StringName@ is new
@StringName@_pkg.Bounded_String;</StringBoundedTypeHead>

<StringUnboundedTypeHead>
        type @StringName@ is new Common_Types.String;</StringUnboundedTypeHead>

<wStringBoundedTypeHead>
        package @WStringName@_pkg is new
                Ada.Strings.Wide_Bounded.Generic_Bounded_Length(@Length@);
        type @WStringName@ is new
@WStringName@_pkg.Bounded_Wide_String;</wStringBoundedTypeHead>

<wStringUnboundedTypeHead>
        type @WStringName@ is new Common_Types.Wstring;</wStringUnboundedTypeHead>

<FixedTypeHead>
        type @FixedName@ is delta @Delta@ digits @DigitNumber@;</FixedTypeHead>

<ArrayTypeHead>
        type @ArrayName@ is array(0 .. @Length_1@) of
@ArrayElementsType@;</ArrayTypeHead>

<ArrayTypeHead_D>
        type @ArrayName@ is array(0 .. @Length1_1@, 0 .. @Length2_1@) of
@ArrayElementsType@;</ArrayTypeHead_D>

<ArrayTypeHead_T>
        type @ArrayName@ is array(0 .. @Length1_1@, 0 .. @Length2_1@, 0 .. @Length3_1@)
of @ArrayElementsType@;</ArrayTypeHead_T>


<StructTypeHead>
        type @StructName@ is record
</StructTypeHead>

<StructTypeMember>           @MemberName@: @MemberType@;
</StructTypeMember>

<StructTypeLastMember>       @MemberName@: @MemberType@;
        end record;</StructTypeLastMember>

<UnionTypeHead>
        type @UnionName@ (Switch : @SwitchType@ := @SwitchType@'first) is
```

```
        record
                case Switch is</UnionTypeHead>

<UnionTypeCaseOption>
                        when @CaseCondition@ =></UnionTypeCaseOption>
<UnionTypeOthersOption>
                        when others =></UnionTypeOthersOption>

<UnionTypeCaseMember>
                                @ElementName@: @ElementType@;</UnionTypeCaseMember>

<UnionTypeCaseEnd>
                end case;
        end record;</UnionTypeCaseEnd>


<ConstantTypeHead>
        @ConstantName@ : constant @ConstantType@ := @ConstantValue@;</ConstantTypeHead>

<ExceptionTypeHead>
        @ExceptionName@ : exception;</ExceptionTypeHead>


<AttrOperationHead>
        procedure set_@Attr_Name@(Self: access Face; value: @Attr_Type@)is abstract;
        function get_@Attr_Name@(Self: access Face) return @Attr_Type@ is
abstract;</AttrOperationHead>

<AttrReadOnlyOperationHead>
        function get_@Attr_Name@(Self: access Face) return @Attr_Type@ is
abstract;</AttrReadOnlyOperationHead>

<ProcedureHead>
        procedure @OperationName@(Self: access Face</ProcedureHead>

<OperationParameter>; @ParameterName@ : @ParameterQualifier@
@ParameterType@</OperationParameter>
<FirstParameterOperation>; @ParameterName@ : @ParameterQualifier@
@ParameterType@</FirstParameterOperation>

<EndFunction>) return @ReturnedType@ is abstract;</EndFunction>
<EndProcedure>) is abstract;</EndProcedure>

<FunctionHead>
        function @OperationName@(Self: access Face</FunctionHead>

<EndInterface>
end @InterfaceName@;
</EndInterface>
```

PLANTILLAS PARA LA GENERACIÓN DE LOS PAQUETES ADS/ADB [NOMBRECOMPONENTE]_TYPES

Fichero *Component_TypesTemplate_ADS.txt*

```
<Head>
--
*****************************************************************************
******
-- Project: RT-CCM
-- Group: CTR -Universidad de Cantabria
--
-- Ada specification of the types belong to @ElemType@ : @ElemName@
-- Date: @Date@

--
*****************************************************************************
******
</Head>

<With>
with @Package@;</With>
```

```
<Use>
use @Package@;</Use>

<TypesFileHead>
package @TypesFileName@ is</TypesFileHead>

<EnumTypeHead>
        type @TypeName@ is (
</EnumTypeHead>

<EnumValue>           @EnumValue@,
</EnumValue>

<LastEnumValue>              @EnumValue@);
</LastEnumValue>

<SimpleTypeHead>
        type @TypeName@ is new @IdlType@;</SimpleTypeHead>

<SequenceTypeHead>
        package @SequenceName@_pkg is new
                Ada.Containers.Vectors(Natural,@SequenceType@);
        subtype @SequenceName@ is @SequenceName@_pkg.Vector;</SequenceTypeHead>

<StringBoundedTypeHead>
        package @StringName@_pkg is new
                Ada.Strings.Bounded.Generic_Bounded_Length(@Length@);
        type @StringName@ is new
@StringName@_pkg.Bounded_String;</StringBoundedTypeHead>

<StringUnboundedTypeHead>
        type @StringName@ is new Common_Types.String;</StringUnboundedTypeHead>

<wStringBoundedTypeHead>
        package @WStringName@_pkg is new
                Ada.Strings.Wide_Bounded.Generic_Bounded_Length(@Length@);
        type @WStringName@ is new
@WStringName@_pkg.Bounded_Wide_String;</wStringBoundedTypeHead>

<wStringUnboundedTypeHead>
        type @WStringName@ is new Common_Types.Wstring;</wStringUnboundedTypeHead>

<FixedTypeHead>
        type @FixedName@ is delta @Delta@ digits @DigitNumber@;</FixedTypeHead>

<ArrayTypeHead>
        type @ArrayName@ is array(0 .. @Length_1@) of
@ArrayElementsType@;</ArrayTypeHead>

<ArrayTypeHead_D>
        type @ArrayName@ is array(0 .. @Length1_1@, 0 .. @Length2_1@) of
@ArrayElementsType@;</ArrayTypeHead_D>

<ArrayTypeHead_T>
        type @ArrayName@ is array(0 .. @Length1_1@, 0 .. @Length2_1@, 0 .. @Length3_1@)
of @ArrayElementsType@;</ArrayTypeHead_T>


<StructTypeHead>
        type @StructName@ is record
</StructTypeHead>

<StructTypeMember>            @MemberName@: @MemberType@;
</StructTypeMember>

<StructTypeLastMember>        @MemberName@: @MemberType@;
        end record;</StructTypeLastMember>

<UnionTypeHead>
        type @UnionName@ (Switch : @SwitchType@ := @SwitchType@'first) is
        record
                case Switch is</UnionTypeHead>

<UnionTypeCaseOption>
                        when @CaseCondition@ =></UnionTypeCaseOption>
<UnionTypeOthersOption>
                        when others =></UnionTypeOthersOption>
```

```
<UnionTypeCaseMember>
                                @ElementName@: @ElementType@;</UnionTypeCaseMember>

<UnionTypeCaseEnd>
                end case;
        end record;</UnionTypeCaseEnd>


<ConstantTypeHead>
        @ConstantName@ : constant @ConstantType@ := @ConstantValue@;</ConstantTypeHead>

<ExceptionTypeHead>
        @ExceptionName@ : exception;</ExceptionTypeHead>

<AnyToAttrHead>
        function Any_To_@Attr_Type@(val:Common_Types.Any) return @Attr_Type@;
</AnyToAttrHead>

<EndTypes>
end @TypesFileName@;
</EndTypes>
```

Fichero *Component_TypesTemplate_ADB.txt*

```
<Head>
--
*******************************************************************************
******
-- Project: RT-CCM
-- Group: CTR -Universidad de Cantabria
--
-- Ada specification of the types belong to @ElemType@ : @ElemName@
-- Date: @Date@

--
*******************************************************************************
******
</Head>

<With>
with @Package@;</With>

<Use>
use @Package@;</Use>

<TypesFileHead>
package body @TypesFileName@ is</TypesFileHead>

<AnyToAttrHead>
        function Any_To_@Attr_Type@(val:Common_Types.Any) return @Attr_Type@ is
                toReturn : @Attr_Type@;
                aux : Integer;
        begin
                aux := Integer'Value(Common_types.Any_To_Standard_String(val));
                toReturn := @Attr_Type@(aux);
                return toReturn;
        end Any_To_@Attr_Type@;
</AnyToAttrHead>

<EndTypes>
end @TypesFileName@;
</EndTypes>
```

PLANTILLA PARA LA GENERACIÓN DEL PAQUETE ADS [NOMBRECOMPONENTE]_EXEC

Fichero *Component_ExecTemplate.txt*

```
<Head>
--
*******************************************************************************
******
```

```
-- Project: RT-CCM
-- Group: CTR -Universidad de Cantabria
--
-- Ada specification of the types belong to @ElemType@ : @ElemName@
-- Date: @Date@

--
*******************************************************************************
******
</Head>

<With>
with @Package@;</With>

<Use>
use @Package@;</Use>

<ComponentHead>

package @ComponentName@ is
</ComponentHead>

<FacetInterfaceHead>
        type CCM_@InterfaceName@ is interface and @InterfaceName@.Face;
        type CCM_@InterfaceName@_Ref is access all CCM_@InterfaceName@'Class;
</FacetInterfaceHead>

<MonolithicExecutorInterfaceHead_1>
        type CCM_@ComponentName@ is interface
                and
@FatherComponent@_Exec.CCM_@FatherComponent@</MonolithicExecutorInterfaceHead_1>

<MonolithicExecutorInterfaceHead_2>
        type CCM_@ComponentName@ is interface
                and Components.SessionComponent</MonolithicExecutorInterfaceHead_2>

<MonolithicExecutorInterfaceWithSupport>
                and @SupportedInterface@.face</MonolithicExecutorInterfaceWithSupport>

<MonolithicExecutorInterfaceRef>;
        type CCM_@ComponentName@_Ref is access all CCM_@ComponentName@'Class;
</MonolithicExecutorInterfaceRef>

<AttrOperationHead>
        function get_@Attr_Name@(Self: access @PropietaryClassName@)
                return @Attr_Type@ is abstract;
        procedure set_@Attr_Name@(Self: access @PropietaryClassName@;
                attribute: in @Attr_Type@) is abstract;
</AttrOperationHead>

<AttrReadOnlyOperationHead>
        function get_@Attr_Name@(Self: access @PropietaryClassName@)
                return @Attr_Type@ is abstract;
</AttrReadOnlyOperationHead>

<FacetAccessOperationHead>
        function get_@PortName@(Self: access CCM_@ComponentName@)
                return CCM_@InterfaceName@_Ref is abstract;</FacetAccessOperationHead>

<OSASettingOperationHead>
        procedure set_@PortName@_SchedParamID(self:access CCM_@ComponentName@;
                                id:SchedulingControlService.SchedControlID) is
abstract;</OSASettingOperationHead>
<PASettingOperationHead>
        procedure set_@PortName@_Period(self:access CCM_@ComponentName@;period:Duration)
is abstract;
        procedure set_@PortName@_SchedParamID(self:access CCM_@ComponentName@;
                                id:SchedulingControlService.SchedControlID) is
abstract;</PASettingOperationHead>

<ConfigCompleteHead>
        procedure configuration_complete(Self: access CCM_@ComponentName@) is abstract;
</ConfigCompleteHead>

<ContextInterfaceHead_1>
        type CCM_@ComponentName@_Context is interface
                and
@FatherComponent@_Exec.CCM_@FatherComponent@_Context;</ContextInterfaceHead_1>
```

```
<ContextInterfaceHead_2>
        type CCM_@ComponentName@_Context is interface
                and Components.SessionContext;</ContextInterfaceHead_2>

<ContextInterfaceRef>
        type CCM_@ComponentName@_Context_Ref
                is access all CCM_@ComponentName@_Context'Class;
</ContextInterfaceRef>

<ConnectionOperationHead>
        function get_connection_@ReceptacleName@(Self: access
CCM_@ComponentName@_Context)
                return @InterfaceName@.Ref is abstract;
</ConnectionOperationHead>

<ConnectionMultipleOperationHead>
        type @ReceptacleName@Connection is record
                objRef : @InterfaceName@.Ref;
                ck  :  Components.Cookie_Ref;
        end record;

-- Container Vector of @ReceptacleName@Connection's elements.
        package @ReceptacleName@Connections_pkg is new
                Ada.Containers.Vectors(Natural,@ReceptacleName@Connection);
        use @ReceptacleName@Connections_pkg;

        type @ReceptacleName@Connections is
                new @ReceptacleName@Connections_pkg.Vector with null record;
        type @ReceptacleName@Connections_Ref is access @ReceptacleName@Connections;

-- function that returns a list of multiple receptacles references.
        function get_connections_@ReceptacleName@(Self: access
CCM_@ComponentName@_Context)
                return @ReceptacleName@Connections_Ref is abstract;
</ConnectionMultipleOperationHead>

<HomeExplicitExecutorInterfaceHead_1>
        type CCM_@HomeName@Explicit is interface
                and
@FatherComponent@_Exec.CCM_@FatherHome@Explicit;</HomeExplicitExecutorInterfaceHead_1>

<HomeExplicitExecutorInterfaceHead_2>
        type CCM_@HomeName@Explicit
                is interface and
Components.HomeExecutorBase;</HomeExplicitExecutorInterfaceHead_2>

<HomeExplicitExecutorInterfaceRef>
        type CCM_@HomeName@Explicit_Ref
                is access all CCM_@HomeName@Explicit'Class;
</HomeExplicitExecutorInterfaceRef>

<HomeExplicitOperationHead>
        function @OperationName@(Self: access
CCM_@HomeName@Explicit</HomeExplicitOperationHead>

<HomeExplicitOperationParameter>;
                                @OperationParameter@ :
@ParameterType@</HomeExplicitOperationParameter>

<HomeExplicitOperationEnd>)
                return Components.EnterpriseComponent_Ref is abstract;
</HomeExplicitOperationEnd>

<HomeImplicitExecutorInterfaceHead>
        type CCM_@HomeName@Implicit is interface;
        type CCM_@HomeName@Implicit_Ref
                is access all CCM_@HomeName@Implicit'Class;

        function create(Self: access CCM_@HomeName@Implicit)
                return Components.EnterpriseComponent_Ref is abstract;
</HomeImplicitExecutorInterfaceHead>

<HomeMainExecutorInterfaceHead>
        type CCM_@HomeName@ is interface
                and CCM_@HomeName@Explicit and CCM_@HomeName@Implicit;
        type CCM_@HomeName@_Ref is access all CCM_@HomeName@'Class;
</HomeMainExecutorInterfaceHead>
```

```
<EndComponent>
end @ComponentName@;
</EndComponent>
```

PLANTILLAS PARA LA GENERACIÓN DE LOS PAQUETES ADS/ADB [NOMBRECOMPONENTE]_EXEC_IMPL

Fichero *Component_Exec_ImplTemplate_ADS.txt*

```
<Head>
--
*******************************************************************************
******
-- Project: RT-CCM
-- Group: CTR -Universidad de Cantabria
--
-- Ada ADS specification of executor implementation belong to @ElemType@ : @ElemName@
-- Date: @Date@

--
*******************************************************************************
******
</Head>

<With>
with @Package@;</With>

<Use>
use @Package@;</Use>


<ComponentHead>

package @ComponentName@ is
</ComponentHead>

<ExecutorImplementationClassHead>
        type Instance is new Object.Face and CCM_@ComponentName@ with private;
        type Ref is access all Instance'Class;
</ExecutorImplementationClassHead>

<AttrOperationHead>
        function get_@Attr_Name@(Self: access Instance)
                return @Attr_Type@;
        procedure set_@Attr_Name@(Self: access Instance;
                value: @Attr_Type@);
</AttrOperationHead>

<AttrReadOnlyOperationHead>
        function get_@Attr_Name@(Self: access Instance)
                return @Attr_Type@;
</AttrReadOnlyOperationHead>

<FacetAccessOperationHead>
        function get_@PortName@(Self: access Instance)
                return CCM_@InterfaceName@_Ref;</FacetAccessOperationHead>

<OSASettingOperationHead>
        procedure set_@PortName@_SchedParamID(self:access Instance;

        id:SchedulingControlService.SchedControlID);</OSASettingOperationHead>
<PASettingOperationHead>
        procedure set_@PortName@_Period(self:access Instance;period:Duration);
        procedure set_@PortName@_SchedParamID(self:access Instance;

        id:SchedulingControlService.SchedControlID);</PASettingOperationHead>

<ConfigCompleteHead>
        procedure configuration_complete(Self: access Instance);
</ConfigCompleteHead>

<ProcedureHead>
```

```
        procedure @OperationName@(Self: access Instance</ProcedureHead>

<OperationParameter>; @ParameterName@ : @ParameterQualifier@
@ParameterType@</OperationParameter>

<EndFunction>) return @ReturnedType@;</EndFunction>
<EndProcedure>);</EndProcedure>

<FunctionHead>
        function @OperationName@(Self: access Instance</FunctionHead>

<ObjectInheritanceOperationHead>
        function get_component(Self: access Instance) return Object.Ref;
</ObjectInheritanceOperationHead>

<SessionComponentInheritedOperations>
        procedure set_session_context ( Self: access Instance;
                ctx: in Components.SessionContext_Ref);
        procedure ccm_activate(Self: access Instance);
        procedure ccm_passivate(Self: access Instance);
        procedure ccm_remove (Self: access Instance);
</SessionComponentInheritedOperations>

<ClassHomeHead>
        type Home is new CCM_@HomeName@ with private;
        type Home_Ref is access all Home'Class;

        function create(Self: access Home)
                return Components.EnterpriseComponent_Ref;

        function create_home return Home_Ref;
</ClassHomeHead>

<HomeExplicitOperationHead>
        function @OperationName@(Self: access Home</HomeExplicitOperationHead>

<HomeExplicitOperationParameter>;
                                @OperationParameter@ :
@ParameterType@</HomeExplicitOperationParameter>

<HomeExplicitOperationEnd>)
                return Components.EnterpriseComponent_Ref;
</HomeExplicitOperationEnd>

<PrivateHead>
private
</PrivateHead>

<ExecutorImplementationPrivateHead>
        type Instance is new Object.Face and CCM_@ComponentName@ with record
                theContext: CCM_@ComponentName@_Context_Ref;
                theImpl: @ComponentName@_Impl.Ref;
        end record;
</ExecutorImplementationPrivateHead>
<PrivateHomeHead>
        type Home is new CCM_@HomeName@ with null record;
</PrivateHomeHead>

<EndComponent>
end @ComponentName@;
</EndComponent>
```

Fichero *Component_Exec_ImplTemplate_ADB.txt*

```
<Head>
--
*****************************************************************************
******
-- Project: RT-CCM
-- Group: CTR -Universidad de Cantabria
--
-- Ada ADB specification of executor implementation belong to @ElemType@ : @ElemName@
-- Date: @Date@

--
*****************************************************************************
******
</Head>
```

```
<With>
with @Package@;</With>

<Use>
use @Package@;</Use>


<ComponentHead>

package body @ComponentName@ is
</ComponentHead>

<AttrOperationHead>
        function get_@Attr_Name@(Self: access Instance)
                return @Attr_Type@ is
        begin
                return Self.theImpl.get_@Attr_Name@;
        end get_@Attr_Name@;
        procedure set_@Attr_Name@(Self: access Instance;
                value: in @Attr_Type@) is
        begin
                Self.theImpl.set_@Attr_Name@(value);
        end set_@Attr_Name@;
</AttrOperationHead>

<AttrReadOnlyOperationHead>
        function get_@Attr_Name@(Self: access Instance)
                return @Attr_Type@ is
        begin
                return Self.theImpl.get_@Attr_Name@;
        end get_@Attr_Name@;
</AttrReadOnlyOperationHead>

<FacetAccessOperationHead>
        function get_@PortName@(Self: access Instance)
                return CCM_@InterfaceName@_Ref is
        begin
                return CCM_@InterfaceName@_Ref(Self.theImpl.get_@PortName@);
        end get_@PortName@;</FacetAccessOperationHead>

<OSASettingOperationHead>
        procedure set_@PortName@_SchedParamID(self:access Instance;
                                id:SchedulingControlService.SchedControlID) is
        begin
                Self.theImpl.set_@PortName@_SchedParamID(id);
        end set_@PortName@_SchedParamID;
</OSASettingOperationHead>

<PASettingOperationHead>
        procedure set_@PortName@_Period(self:access Instance;period:Duration) is
        begin
                Self.theImpl.set_@PortName@_Period(period);
        end set_@PortName@_Period;
        procedure set_@PortName@_SchedParamID(self:access Instance;
                                id:SchedulingControlService.SchedControlID) is
        begin
                Self.theImpl.set_@PortName@_SchedParamID(id);
        end set_@PortName@_SchedParamID;
</PASettingOperationHead>

<ConfigCompleteHead>
        procedure configuration_complete(Self: access Instance) is</ConfigCompleteHead>

<ConfigCompleteReceptacle>
                theRecep_@ReceptacleNumber@:
@ReceptacleName@.Ref;</ConfigCompleteReceptacle>

<ConfigCompleteReceptacleMultiple>
                theReceps_@ReceptacleNumber@:
@PortName@Connections_Ref;</ConfigCompleteReceptacleMultiple>

<ConfigCompleteBegin>
                num : Natural;
        begin</ConfigCompleteBegin>

<ConfigCompleteReceptacleSimpleBody>
                theRecep_@ReceptacleNumber@:= Self.theContext.get_connection_@PortName@;
```

```
                Self.theImpl.@PortName@_Connect(theRecep_@ReceptacleNumber@);
</ConfigCompleteReceptacleSimpleBody>

<ConfigCompleteReceptacleMultipleBody>

        theReceps_@ReceptacleNumber@:=Self.theContext.get_connections_@PortName@;
                num := Natural(theReceps_@ReceptacleNumber@.length);
                Self.theImpl.@PortName@_Cardinality(num);
                for i in 1..num loop

        theRecep_@ReceptacleNumber@:=theReceps_@ReceptacleNumber@.element(i).objRef;
                        Self.theImpl.@PortName@_Connect(i,theRecep_@ReceptacleNumber@);
                end loop;
</ConfigCompleteReceptacleMultipleBody>

<ConfigCompleteReceptacleEnd>
                Self.theImpl.complete;
-- Invocamos desde aqui el activate
                Self.theImpl.activate;
        end configuration_complete;
</ConfigCompleteReceptacleEnd>

<FunctionHead>
        function @OperationName@(Self : access Instance</FunctionHead>

<OperationParameter>;
                                                        @ParameterName@ :
@ParameterQualifier@ @ParameterType@</OperationParameter>
<EndFunction>)
                return @ReturnedType@ is
        begin
                return Self.theImpl.@OperationName@;
        end @OperationName@;
</EndFunction>

<ProcedureHead>
        procedure @OperationName@(Self : access Instance</ProcedureHead>
<EndProcedure>)
        begin
                Self.theImpl.@OperationName@;
        end @OperationName@;
</EndProcedure>


<FirstParameterBodyProcedure>)
        begin

        Self.theImpl.@OperationName@(@ParameterName@</FirstParameterBodyProcedure>

<FirstParameterBodyFunction>)
        begin
                return
Self.theImpl.@OperationName@(@ParameterName@</FirstParameterBodyFunction>

<ParametersBodyOperation>; @ParameterName@</ParametersBodyOperation>

<EndOperationWithParameters>);
        end @OperationName@;
</EndOperationWithParameters>

<ObjectInheritanceOperationHead>
        function get_component(Self: access Instance) return Object.Ref is
        begin
                return Object.Ref(Self);
        end get_component;
</ObjectInheritanceOperationHead>

<SessionComponentInheritedOperations>
        procedure set_session_context (Self: access Instance;
                ctx: in Components.SessionContext_Ref) is
        begin
                if ctx.all in CCM_@ComponentName@_Context'Class then
                        Self.theContext := CCM_@ComponentName@_Context_Ref(ctx);
                else
                        raise Components.CCMException;
                end if;
        end set_session_context;
        procedure ccm_activate(Self: access Instance) is
```

```
        begin
                Self.theImpl.activate;
        end ccm_activate;
        procedure ccm_passivate(Self: access Instance) is
        begin
                Self.theImpl.passivate;
        end ccm_passivate;
        procedure ccm_remove (Self: access Instance) is
        begin
                Self.theImpl.remove;
        end ccm_remove;
</SessionComponentInheritedOperations>

<ClassHomeHead>
        function create(Self: access Home)
                return Components.EnterpriseComponent_Ref is
                theComp : Ref;
        begin
                theComp := new Instance;
                theComp.theImpl := @ComponentName@_Impl.create;
                return Components.EnterpriseComponent_Ref(theComp);
        end create;
        function create_Home return Home_Ref is
        begin
                return new Home;
        end create_Home;
</ClassHomeHead>

<HomeExplicitOperationHead>
        function @OperationName@(Self: access Home</HomeExplicitOperationHead>

<HomeExplicitOperationParameter>;
                                @OperationParameter@ :
@ParameterType@</HomeExplicitOperationParameter>

<HomeExplicitOperationEnd>)
                return Components.EnterpriseComponent_Ref is
                theComp : Ref;
        begin
                theComp := new Instance;
                theComp.theImpl := @ComponentName@_Impl.@OperationName@;
                return Components.EnterpriseComponent_Ref(theComp);
        end @OperationName@;
</HomeExplicitOperationEnd>

<EndComponent>
end @ComponentName@;
</EndComponent>
```

PLANTILLAS PARA LA GENERACIÓN DE LOS PAQUETES ADS/ADB [NOMBRECOMPONENTE]_IMPL

Fichero *Component_ImplTemplate_ADS.txt*

```
<Head>
--
*****************************************************************************
******
-- Project: RT-CCM
-- Group: CTR -Universidad de Cantabria
--
-- Ada ADS specification of implementation belong to @ElemType@ : @ElemName@
-- Date: @Date@
--
*****************************************************************************
******
</Head>

<With>
with @Package@;</With>

<Use>
use @Package@;</Use>
```

```
<ComponentHead>

package @ComponentName@ is
</ComponentHead>

<InstanceClassHead>
        type Instance is new @FatherComponent@_Impl.Instance with private;
        type Ref is access Instance'Class;
</InstanceClassHead>

<InstanceWithoutFatherClassHead>
        type Instance is tagged private;
        type Ref is access Instance'class;</InstanceWithoutFatherClassHead>

<AttrOperationHead>
        function get_@Attr_Name@(Self: access Instance)
                return @Attr_Type@;
        procedure set_@Attr_Name@(Self: access Instance;
                newValue: @Attr_Type@);
</AttrOperationHead>

<AttrReadOnlyOperationHead>
        function get_@Attr_Name@(Self: access Instance)
                return @Attr_Type@;
</AttrReadOnlyOperationHead>

<FacetAccessOperationHead>
        function get_@PortName@(Self: access Instance)
                return
@ComponentName@_Facet@InterfaceName@.Ref;</FacetAccessOperationHead>


<ConnectionOperationHead>
        procedure @ReceptacleName@_Connect(self:access Instance;
face:@InterfaceName@.Ref);
</ConnectionOperationHead>

<ConnectionMultipleOperationHead>
        procedure @ReceptacleName@_Cardinality(self:access Instance; numObj:Natural);
        procedure @ReceptacleName@_Connect(self:access Instance;
                                index:Positive; face:@InterfaceName@.Ref);
</ConnectionMultipleOperationHead>

<OSASettingOperationHead>
        procedure set_@PortName@_SchedParamID(self:access Instance;

        id:SchedulingControlService.SchedControlID);</OSASettingOperationHead>
<PASettingOperationHead>
        procedure set_@PortName@_Period(self:access Instance;period:Duration);
        procedure set_@PortName@_SchedParamID(self:access Instance;

        id:SchedulingControlService.SchedControlID);</PASettingOperationHead>

<HomeExplicitOperationHead>
        function @OperationName@</HomeExplicitOperationHead>

<HomeExplicitImplementationExtra>(
                                @OperationParameter@ : @ParameterType@
</HomeExplicitImplementationExtra>

<HomeExplicitOperationParameter>;
                                @OperationParameter@ :
@ParameterType@</HomeExplicitOperationParameter>

<HomeExplicitOperationEnd>@OptionalParameterParenthesis@
                return Ref;
</HomeExplicitOperationEnd>

<ConstructorsHead>
-- Is invoked before component and facets attributes setting
        function create return Ref;
-- Is invoked after component and facet attributed are set.
        procedure complete(self:access Instance);
-- Life control inherited from session
        procedure activate(self: access Instance);
        procedure passivate(self: access Instance);
        procedure remove (self: access Instance);
</ConstructorsHead>
```

```
<PrivateHead>
private
</PrivateHead>

<StateComponentHead>
        type @ComponentName@_State;
        type @ComponentName@_State_Ref is access @ComponentName@_State;
</StateComponentHead>

<SpecificElementRequiredByUser>
--  ********************************************************************
--  << Introduce the specific element required to the component state>>
--  ********************************************************************
</SpecificElementRequiredByUser>

<ActivationElementHead>
-- Activation element
        activationEnd: exception;
        type ActivationOper is access procedure(state:@ComponentName@_State_Ref);
</ActivationElementHead>

<CreateComponentStateHead>
        function create_@ComponentName@_State return @ComponentName@_State_Ref;
</CreateComponentStateHead>

<CompleteComponentStateHead>
        procedure complete_@ComponentName@_State(state:@ComponentName@_State_Ref);
</CompleteComponentStateHead>

<BussinessFacetPortHead>
        type @PortName@_Port is new @ComponentName@_Facet@InterfaceName@.Face with
record
                state:@ComponentName@_State_Ref;</BussinessFacetPortHead>
<BussinessFacetPortAttribute_1>
                @AttrName@ : @InterfaceName@.@AttrType@;</BussinessFacetPortAttribute_1>
<BussinessFacetPortAttribute_2>
                @AttrName@ : @AttrType@;</BussinessFacetPortAttribute_2>
<BussinessFacetPortEnd>
        end record;
        type @PortName@_Port_Ref is access @PortName@_Port'class;
</BussinessFacetPortEnd>
<BussinessFacetPortOperations>
        function create_@PortName@_Port(state:@ComponentName@_State_Ref)
                        return @PortName@_Port_Ref;
        procedure complete_@PortName@_Port(port:@PortName@_Port_Ref);
</BussinessFacetPortOperations>

<AttrOperationHead_2>
        function get_@Attr_Name@(Self: access @PropietaryClassName@_Port)
                return @Attr_Type@;
        procedure set_@Attr_Name@(Self: access @PropietaryClassName@_Port;
                newValue: @Attr_Type@);
</AttrOperationHead_2>

<AttrReadOnlyOperationHead_2>
        function get_@Attr_Name@(Self: access @PropietaryClassName@_Port)
                return @Attr_Type@;
</AttrReadOnlyOperationHead_2>

<AttrOperationHead_3>
        function get_@Attr_Name@(Self: access @PropietaryClassName@_Port)
                return @Attr_Type@;
        procedure set_@Attr_Name@(Self: access @PropietaryClassName@_Port;
                newValue: @Attr_Type@);
</AttrOperationHead_3>

<AttrReadOnlyOperationHead_3>
        function get_@Attr_Name@(Self: access @PropietaryClassName@_Port)
                return @Attr_Type@;
</AttrReadOnlyOperationHead_3>

<ProcedureHead>
        procedure @OperationName@(Self: access
@PropietaryClassName@_Port</ProcedureHead>

<OperationParameter>;
                                @ParameterName@ : @ParameterQualifier@
```

```
@ParameterType@</OperationParameter>
<FirstParameterOperation>;
                                @ParameterName@ : @ParameterQualifier@
@ParameterType@</FirstParameterOperation>
<EndFunction>)
                                return @ReturnedType@;</EndFunction>
<EndProcedure>);</EndProcedure>

<FunctionHead>
        function @OperationName@(Self: access @PropietaryClassName@_Port</FunctionHead>


<OSActivationElementHead>
        protected type OneShotActivationController is
                procedure activate;
                procedure passivate;
                procedure remove;
                entry taskControl;
        private
                isActivate:Boolean:=False;
                isRemoved:Boolean:=False;
        end OneShotActivationController;

        type OneShotActivationController_Ref is access OneShotActivationController;

        type OneShotActivationBlock is record
                state: @ComponentName@_State_Ref;
                schedulingId:SchedulingControlService.SchedControlID;
                run: ActivationOper;
                controller:OneShotActivationController_Ref;
        end record;

        type OneShotActivationBlock_Ref is access OneShotActivationBlock;

        task type OneShotActivationTask (block: OneShotActivationBlock_Ref);
        type OneShotActivationTask_Ref is access OneShotActivationTask;
</OSActivationElementHead>

<PActivationElementHead>
        protected type PeriodicActivationController is
                procedure activate;
                procedure passivate;
                procedure remove;
                entry taskControl;
        private
                isActivate:Boolean:=False;
                isRemoved:Boolean:=False;
        end PeriodicActivationController;

        type PeriodicActivationController_Ref is access PeriodicActivationController;

        type PeriodicActivationBlock is record
                state: @ComponentName@_State_Ref;
                period: Duration:=5.0; -- Esto es provisional. Este atributo deberÃ¡ ser
inicializado en el despliegue
                schedulingId:SchedulingControlService.SchedControlID;
                update: ActivationOper;
                controller:PeriodicActivationController_Ref;
        end record;

        type PeriodicActivationBlock_Ref is access PeriodicActivationBlock;

        task type PeriodicActivationTask (block: PeriodicActivationBlock_Ref);
        type PeriodicActivationTask_Ref is access PeriodicActivationTask;
</PActivationElementHead>

<OSARunHead>
        procedure @PortName@_Run(state: @ComponentName@_State_Ref);
</OSARunHead>

<PAUpdateHead>
        procedure @PortName@_Update(state: @ComponentName@_State_Ref);
</PAUpdateHead>

<InstanceClassPrivateHead>
        type Instance is new @FatherComponent@_Impl.Instance with record
--      Component state
                state:@ComponentName@_State_Ref;</InstanceClassPrivateHead>
```

```
<InstanceWithoutFatherClassPrivateHead>
        type Instance is tagged record
--      Component state
                state:@ComponentName@_State_Ref;</InstanceWithoutFatherClassPrivateHead>

<InstanceFacetImplementation>
                @PortName@: @PortName@_Port_Ref:=null;</InstanceFacetImplementation>

<InstanceOSAPortImplementation>
                @PortName@:
OneShotActivationTask_Ref:=null;</InstanceOSAPortImplementation>

<InstancePAPortImplementation>
                @PortName@:
PeriodicActivationTask_Ref:=null;</InstancePAPortImplementation>

<InstanceClassPrivateEnd>
        end record;
</InstanceClassPrivateEnd>

<ArrayReceptacleContainerHead>
        type @PortName@_Container is array (Positive range <>) of @InterfaceName@.Ref;
        type @PortName@_Container_Ref is access @PortName@_Container;
</ArrayReceptacleContainerHead>

<UserStateInformationHead>

--
        ********************************************************************************
********
--      << Introduce the required resources for describing the @ComponentName@ component
state>>
--
        ********************************************************************************
********
</UserStateInformationHead>

<ClassStateComponentHead>
        type @ComponentName@_State is tagged record
-- Structural element of the state</ClassStateComponentHead>

<ClassStateSimpleReceptacleReference>
                @PortName@: @InterfaceName@.Ref;</ClassStateSimpleReceptacleReference>

<ClassStateMultipleReceptacleReference>
                @PortName@_num:Natural;

        @PortName@:@PortName@_Container_Ref;</ClassStateMultipleReceptacleReference>

<ClassStateComponentAttribute>
                @Attr_Name@ : @Attr_Type@;</ClassStateComponentAttribute>

<ClassStateOSAPortBlockData>
                @PortName@:OneShotActivationBlock_Ref;</ClassStateOSAPortBlockData>

<ClassStatePAPortBlockData>
                @PortName@:PeriodicActivationBlock_Ref;</ClassStatePAPortBlockData>

<ClassStateSpecificUserAttributesAndEndClass>
-- Specific attributes of the component @ComponentName@

                -- ******************************************
                -- << introduce specific elements of the state>>
                -- ******************************************

        end record;
</ClassStateSpecificUserAttributesAndEndClass>

<ClassStateEnd>
        end record;
</ClassStateEnd>

<ClassStateWithoutRecordsEnd>
                null;
        end record;
</ClassStateWithoutRecordsEnd>
```

```
<ClassStateSpecificUserAttributesAndEndWithoutRecordsClass>
-- Specific attributes of the component @ComponentName@
               null;
               -- ******************************************
               -- << introduce specific elements of the state>>
               -- ******************************************

       end record;
</ClassStateSpecificUserAttributesAndEndWithoutRecordsClass>

<EndComponent>
end @ComponentName@;
</EndComponent>
```

Fichero *Component_ImplTemplate_ADB.txt*

```
<Head>
--
*****************************************************************************************
******
-- Project: RT-CCM
-- Group: CTR -Universidad de Cantabria
--
-- Ada ADB specification of implementation belong to @ElemType@ : @ElemName@
-- Date: @Date@

--
*****************************************************************************************
******
</Head>

<With>
with @Package@;</With>

<Use>
use @Package@;</Use>


<ComponentHead>

package body @ComponentName@ is
</ComponentHead>

<AttrOperationHead>
       function get_@Attr_Name@(Self: access Instance)
               return @Attr_Type@ is
       begin
               return Self.state.@Attr_Name@;
       end get_@Attr_Name@;
       procedure set_@Attr_Name@(Self: access Instance;
               newValue: @Attr_Type@) is
       begin
               self.state.@Attr_Name@:=newValue;
       end set_@Attr_Name@;
</AttrOperationHead>

<AttrReadOnlyOperationHead>
       function get_@Attr_Name@(Self: access Instance)
               return @Attr_Type@ is
       begin
               return Self.state.@Attr_Name@;
       end get_@Attr_Name@;
</AttrReadOnlyOperationHead>

<FacetAccessOperationHead>
       function get_@PortName@(Self: access Instance)
               return @ComponentName@_Facet@InterfaceName@.Ref is
       begin
               return @ComponentName@_Facet@InterfaceName@.Ref(Self.@PortName@);
       end get_@PortName@;</FacetAccessOperationHead>

<ConnectionOperationHead>
       procedure @ReceptacleName@_Connect(self:access Instance;
face:@InterfaceName@.Ref) is
       begin
               Self.state.@ReceptacleName@:=face;
       end @ReceptacleName@_Connect;
```

```
</ConnectionOperationHead>

<ConnectionMultipleOperationHead>
        procedure @ReceptacleName@_Cardinality(self:access Instance; numObj:Natural) is
        begin
                self.state.@ReceptacleName@_num:=numObj;
                self.state.@ReceptacleName@:=new @ReceptacleName@_Container(1..numObj);
        end @ReceptacleName@_Cardinality;

        procedure @ReceptacleName@_Connect(self:access Instance;
                                index:Positive; face:@InterfaceName@.Ref) is
        begin
                self.state.@ReceptacleName@(index):=face;
        end @ReceptacleName@_Connect;
</ConnectionMultipleOperationHead>

<OSASettingOperationHead>
        procedure set_@PortName@_SchedParamID(self:access Instance;
                                id:SchedulingControlService.SchedControlID) is
        begin
                self.state.@PortName@.schedulingId:=id;
        end set_@PortName@_SchedParamID;
</OSASettingOperationHead>

<PASettingOperationHead>
        procedure set_@PortName@_Period(self:access Instance;period:Duration) is
        begin
                self.state.@PortName@.period:=period;
        end set_@PortName@_Period;
        procedure set_@PortName@_SchedParamID(self:access Instance;
                                id:SchedulingControlService.SchedControlID) is
        begin
                self.state.@PortName@.schedulingId:=id;
        end set_@PortName@_SchedParamID;
</PASettingOperationHead>

<HomeExplicitOperationHead>
        function @OperationName@</HomeExplicitOperationHead>

<HomeExplicitImplementationExtra>(
                                @OperationParameter@ : @ParameterType@
</HomeExplicitImplementationExtra>

<HomeExplicitOperationParameter>;
                                @OperationParameter@ :
@ParameterType@</HomeExplicitOperationParameter>

<HomeExplicitOperationEnd>@OptionalParameterParenthesis@
                return Ref is
        begin

                -- **********************
                -- <<Introduce user code>>
                -- **********************

                return null;
        end @OperationName@;
</HomeExplicitOperationEnd>

<CreateHead>
-- Is invoked before component and facets attributes setting
        function create return Ref is
                newInstance:Ref;
        begin
-- Create the component element
                newInstance:=new Instance;
-- Create the state element
                newInstance.state:=create_@ComponentName@_State;
-- Create facet elements</CreateHead>
<CreateFacetElement>
                newInstance.@PortName@:=

        create_@PortName@_Port(newInstance.state);</CreateFacetElement>
<CreateEnd>
-- Return instance
                return newInstance;
        end create;
</CreateEnd>
```

```
<CompleteHead>
-- Is invoked after component and facet attributed are set.
        procedure complete(self:access Instance) is
        begin
-- Complete component state
                complete_@ComponentName@_State(self.state);
-- complete facet ports</CompleteHead>
<CompleteFacetElement>
                complete_@PortName@_Port(self.@PortName@);</CompleteFacetElement>

<CompleteEnd>
        end complete;
</CompleteEnd>
<ActivateHead>
-- Life control inherited from session
        procedure activate(self: access Instance) is
        begin
-- if the first time create them</ActivateHead>

<ActivateOSAElement>
                if (self.@PortName@=null)
                then
                        self.state.@PortName@:=new OneShotActivationBlock;
                        self.state.@PortName@.state:=self.State;
                        self.state.@PortName@.run:=@PortName@_Run'access;
                        self.state.@PortName@.controller:= new
OneShotActivationController;
                        self.@PortName@:=new
OneShotActivationTask(self.state.@PortName@);
                end if;
-- Anyway activates the controller
                self.state.@PortName@.controller.activate;
</ActivateOSAElement>

<ActivatePAElement>
                if (self.@PortName@=null)
                then
                        self.state.@PortName@:=new PeriodicActivationBlock;
                        self.state.@PortName@.state:=self.State;
                        self.state.@PortName@.update:=@PortName@_Update'access;
                        self.state.@PortName@.controller:= new
PeriodicActivationController;
                        self.@PortName@:=new
PeriodicActivationTask(self.state.@PortName@);
                end if;
-- Anyway activates the controller
                self.state.@PortName@.controller.activate;
</ActivatePAElement>

<ActivateEnd>
        end activate;
</ActivateEnd>

<ActivateWithoutActivationPortsEnd>
                null;
        end activate;
</ActivateWithoutActivationPortsEnd>

<PassivateHead>
        procedure passivate(self: access Instance) is
        begin</PassivateHead>
<PassivateElement>
                self.state.@PortName@.controller.passivate;</PassivateElement>
<PassivateEnd>
        end passivate;
</PassivateEnd>

<PassivateWithoutActivationPortsEnd>
                null;
        end passivate;
</PassivateWithoutActivationPortsEnd>

<RemoveHead>
        procedure remove (self: access Instance) is
        begin</RemoveHead>

<RemoveElement>
                self.state.@PortName@.controller.remove;</RemoveElement>
```

```
<RemoveEnd>
        end remove;
</RemoveEnd>

<RemoveWithoutActivationPortsEnd>
                null;
        end remove;
</RemoveWithoutActivationPortsEnd>

<SpecificElementRequiredByUser>

        -- ********************************************************************
        -- << Introduce the specific element required to the component state>>
        -- ********************************************************************
</SpecificElementRequiredByUser>

<CreateComponentStateHead>
        function create_@ComponentName@_State return @ComponentName@_State_Ref is
                newState:@ComponentName@_State_Ref;
        begin
                newState:=new @ComponentName@_State;</CreateComponentStateHead>

<CreateComponentStateReceptacles>
                newState.@PortName@:=null;</CreateComponentStateReceptacles>

<CreateComponentStatePeriodicActivationPorts>
                newState.@PortName@:=new
PeriodicActivationBlock;</CreateComponentStatePeriodicActivationPorts>

<CreateComponentStateOneShotActivationPorts>
                newState.@PortName@:=new
OneShotActivationBlock;</CreateComponentStateOneShotActivationPorts>

<CreateComponentStateComplexReceptacles>
                newState.@PortName@_num:=0;
                newState.@PortName@:=null;</CreateComponentStateComplexReceptacles>

<CreateComponentSpecificUserAttrs>

                -- ********************************************************************
                -- << Introduce the creation of specific attributes of the state >>
                -- ********************************************************************
</CreateComponentSpecificUserAttrs>

<CreateComponentStateEnd>
                return newState;
        end create_@ComponentName@_State;
</CreateComponentStateEnd>

<CompleteComponentStateHead>
        procedure complete_@ComponentName@_State(state:@ComponentName@_State_Ref)is
        begin

                -- ************************************************
                -- <<Introduce code to complete the component state >>
                -- ************************************************

                null; -- ONLY FOR COMPILING IF THERE ARE NOT CODE HERE
        end complete_@ComponentName@_State;
</CompleteComponentStateHead>

<CompleteConnectorStateHead>
        procedure complete_@ComponentName@_State(state:@ComponentName@_State_Ref)is
        begin
                null; -- ONLY FOR COMPILING IF THERE ARE NOT CODE HERE
        end complete_@ComponentName@_State;
</CompleteConnectorStateHead>

<BussinessConnectorFacetPortOperations>
        function create_@PortName@_Port(state:@ComponentName@_State_Ref)
                                return @PortName@_Port_Ref is
                newPort:@PortName@_Port_Ref;
        begin
                newPort:=new @PortName@_Port;
                newPort.state:=state;

                return newPort;
```

```
        end create_@PortName@_Port;

        procedure complete_@PortName@_Port(port:@PortName@_Port_Ref)is
        begin
                null;
        end complete_@PortName@_Port;
</BussinessConnectorFacetPortOperations>

<BussinessFacetPortOperations>
        function create_@PortName@_Port(state:@ComponentName@_State_Ref)
                                return @PortName@_Port_Ref is
                newPort:@PortName@_Port_Ref;
        begin
                newPort:=new @PortName@_Port;
                newPort.state:=state;

                -- ***********************************
                -- <<create specific port structure >>
                -- ***********************************

                return newPort;
        end create_@PortName@_Port;

        procedure complete_@PortName@_Port(port:@PortName@_Port_Ref)is
        begin

                -- ************************************************
                -- <<introduce code to complete the port structure >>
                -- ************************************************

                null;
        end complete_@PortName@_Port;
</BussinessFacetPortOperations>

<AttrOperationHead_2>
        function get_@Attr_Name@(Self: access @PropietaryClassName@_Port)
                return @Attr_Type@ is
        begin
            return self.@Attr_Name@;
        end get_@Attr_Name@;
        procedure set_@Attr_Name@(Self: access @PropietaryClassName@_Port;
                newValue: in @Attr_Type@) is
        begin
                self.@Attr_Name@:=newValue;
        end set_@Attr_Name@;
</AttrOperationHead_2>

<AttrReadOnlyOperationHead_2>
        function get_@Attr_Name@(Self: access @PropietaryClassName@_Port)
                return @Attr_Type@ is
        begin
            return self.@Attr_Name@;
        end get_@Attr_Name@;
</AttrReadOnlyOperationHead_2>



<FunctionHead>
        function @OperationName@(Self : access @PropietaryClassName@_Port</FunctionHead>
<OperationParameter>;
                                                        @ParameterName@ :
@ParameterQualifier@ @ParameterType@</OperationParameter>
<FirstParameterOperation>;
                                                        @ParameterName@ :
@ParameterQualifier@ @ParameterType@</FirstParameterOperation>

<EndFunction>)
                return @ReturnedType@ is
        begin

                --
*********************************************************************
                -- <<Introduce user code: this function must return type:
@ReturnedType@>>
                --
*********************************************************************
                return --a @ReturnedType@ value for compile!.
        end @OperationName@;
```

```
</EndFunction>

<ProcedureHead>
        procedure @OperationName@(Self : access
@PropietaryClassName@_Port</ProcedureHead>
<EndProcedure>) is
        begin

                -- ************************
                -- <<Introduce user code>>
                -- ************************

                null;
        end @OperationName@;
</EndProcedure>

<FirstParameterBodyProcedure>) is
        begin

                -- ************************
                -- <<Introduce user code>>
                -- ************************

                null;
        end @OperationName@;
</FirstParameterBodyProcedure>

<FirstParameterBodyFunction>)
                return @ReturnedType@ is
        begin

                --
**********************************************************************
                -- <<Introduce user code: this function must return type:
@ReturnedType@>>
                --
**********************************************************************
                return --a @ReturnedType@ value for compile!.
        end @OperationName@;</FirstParameterBodyFunction>

<ParametersBodyOperation> </ParametersBodyOperation>

<EndOperationWithParameters> </EndOperationWithParameters>

<OSActivationElementHead>
-- One shot activation controller
        protected body OneShotActivationController is
                procedure activate is
                begin
                        isActivate:=True;
                end activate;

                procedure passivate is
                begin
                        isActivate:=False;
                end passivate;

                procedure remove is
                begin
                        isRemoved:=True;
                        isActivate:=True;
                end remove;

                entry taskControl when (isActivate or isRemoved) is
                begin
                        if isRemoved then raise activationEnd;end if;
                end taskControl;
        end OneShotActivationController;

        task body OneShotActivationTask  is
        begin
                block.controller.taskControl;
                SchedulingControlService.bind(block.schedulingId);
                select
                        block.controller.taskControl;
                then abort
                        block.Run(block.state);
                        raise activationEnd;
```

```
                end select;
        exception
                when activationEnd
=>SchedulingControlService.unbind(block.schedulingId);
        end OneShotActivationTask;
</OSActivationElementHead>

<PActivationElementHead>
--  Periodic activation controller:
        protected body PeriodicActivationController is
                procedure activate is
                begin
                        isActivate:=True;
                end activate;

                procedure passivate is
                begin
                        isActivate:=False;
                end passivate;

                procedure remove is
                begin
                        isRemoved:=True;
                        isActivate:=True;
                end remove;

                entry taskControl when isActivate is
                begin
                        if isRemoved then raise activationEnd;end if;
                end taskControl;
        end PeriodicActivationController;

        task body PeriodicActivationTask  is
                nextUpdate:Time:=Clock;
        begin
                SchedulingControlService.bind(block.schedulingId);
                loop
                        block.controller.taskControl;
                        block.update(block.state);
                        delay until nextUpdate;
                        nextUpdate:= nextUpdate + block.period;
                end loop;
        exception
                when activationEnd
=>SchedulingControlService.unbind(block.schedulingId);
        end PeriodicActivationTask;
</PActivationElementHead>

<OSARunHead>
        procedure @PortName@_Run(state: @ComponentName@_State_Ref)is
        begin

                -- ************************************
                -- <<Introduce code for periodic update >>
                -- ************************************

                null;  -- Delete it!, only for initial compilation
   end @PortName@_Run;
</OSARunHead>

<PAUpdateHead>
        procedure @PortName@_Update(state: @ComponentName@_State_Ref)is
        begin

                -- ************************************
                -- <<Introduce code for periodic update >>
                -- ************************************

                null;  -- Delete it!, only for initial compilation
        end @PortName@_Update;
</PAUpdateHead>

<EndComponent>
end @ComponentName@;
</EndComponent>
```

PLANTILLAS PARA LA GENERACIÓN DE LOS PAQUETES ADS/ADB [NOMBRECOMPONENTE]_WRAPPER

Fichero *Component_WrapperTemplate_ADS.txt*

```
<Head>
--
*******************************************************************************
******
-- Project: RT-CCM
-- Group: CTR -Universidad de Cantabria
--
-- Ada ADS specification of the Wrapper belong to @ElemType@ : @ElemName@
-- Date: @Date@

--
*******************************************************************************
******
</Head>

<With>
with @Package@;</With>

<Use>
use @Package@;</Use>


<ComponentHead>

package @ComponentName@ is
</ComponentHead>

<ContextImplementationInterfaceHead>
        type @ComponentName@_Context
                is new @ComponentName@_Exec.CCM_@ComponentName@_Context with private;
        type @ComponentName@_Context_Ref is access all @ComponentName@_Context'Class;
</ContextImplementationInterfaceHead>

<ConnectionOperationHead>
        function get_connection_@ReceptacleName@(Self: access @ComponentName@_Context)
                return @InterfaceName@.Ref;
</ConnectionOperationHead>

<ConnectionMultipleOperationHead>
        function get_connections_@ReceptacleName@(Self: access @ComponentName@_Context)
                return @ReceptacleName@Connections_Ref;
</ConnectionMultipleOperationHead>

<ContextInterfacesInheritedHead>
--      Inherited from Components.CCMContext
        function get_CCM_home(Self: access @ComponentName@_Context)
                                return Components.CCMHome_Ref;
--      Inherited from Components.SessionContext
        function get_CCM_object(Self: access @ComponentName@_Context)
                                return Object.Ref;
</ContextInterfacesInheritedHead>

<WrapperImplementationInterfaceHead>
        type @ComponentName@_Wrapper is new Object.Face
                and Components.CCMObject with private;
        type @ComponentName@_wrapper_Ref is access all @ComponentName@_wrapper'Class;
</WrapperImplementationInterfaceHead>

<WrapperInterfacesInheritedHead>
--      Inherited from Object
        function get_component(Self: access @ComponentName@_Wrapper)
                                return Object.Ref;

--      Inherited from CCMObject
        function get_ccm_home(Self: access @ComponentName@_Wrapper)
                                return Components.CCMHome_Ref;
        procedure configuration_complete(Self: access @ComponentName@_Wrapper);
        procedure remove(Self: access @ComponentName@_Wrapper);

--      Inherited from Navigation interface
        function provide_facet(Self: access @ComponentName@_Wrapper;
```

```
                              name: in FeatureName)return Object.Ref;

--      Inherited from Receptacles interface
        function connect(Self: access @ComponentName@_Wrapper;
                              name: in FeatureName;
                              connection:in Object.Ref) return Cookie_Ref;
        procedure disconnect(Self: access @ComponentName@_Wrapper;
                              name: in FeatureName;
                              ck: in Cookie_Ref);
</WrapperInterfacesInheritedHead>

<WrapperHomeImplementationHead>
        type @ComponentName@_Home_Wrapper is new
                Components.HomeConfiguration and Components.KeylessCCMHome
                and I_Set_Home_executor.Face with private;
        type @ComponentName@_home_Wrapper_Ref is access all
                @ComponentName@_home_wrapper'Class;
</WrapperHomeImplementationHead>

<WrapperHomeHead>
        type @HomeName@_Wrapper is new Components.HomeConfiguration
                and Components.KeylessCCMHome and I_Set_Home_executor.Face with private;

        type @HomeName@_Wrapper_Ref is access all @HomeName@_Wrapper'Class;
        function create(Self: access @HomeName@_Wrapper)
                              return @ComponentName@_EqInt.@ComponentName@_Ref;
</WrapperHomeHead>

        function @OperationName@(Self: access @HomeName@ExplicitWrapper
<HomeExplicitOperationHead>
        function @OperationName@(Self: access
@HomeName@_Wrapper</HomeExplicitOperationHead>

<HomeExplicitOperationParameter>;
                              @OperationParameter@ :
@ParameterType@</HomeExplicitOperationParameter>

<HomeExplicitOperationEnd>)
                return @ComponentName@_EqInt.@ComponentName@_Ref;
</HomeExplicitOperationEnd>

<WrapperHomeInheritedHead>
--      Inherited from CCMHome
        procedure remove_component(Self: access @HomeName@_wrapper;
                              comp: CCMObject_Ref);

--      Inherited from Homeconfiguration
        procedure set_configuration_values(Self: access @HomeName@_wrapper;
                              config: ConfigValues_Ref);
        procedure complete_component_configuration
                              (Self: access @HomeName@_wrapper; b: in boolean);
        procedure disable_home_configuration
                              (Self: access @HomeName@_wrapper );

--      Inherited from KeylessCCMHome; Raises CreateFailure
        function create_component(Self: access @HomeName@_wrapper)
                              return Components.CCMObject_Ref;

--      Inherited from I_Set_Home_Executor.Face Interface
        procedure set_home_executor(Self: access @HomeName@_Wrapper;
                              theHome: in Components.HomeExecutorBase_Ref);

--      for creating the home
        function create return Components.CCMHome_Ref;
</WrapperHomeInheritedHead>

<PrivateHead>
private
</PrivateHead>

<ProxyInterfaceHead>
        type wrapper_@InterfaceName@ is new Object.Face
                and @InterfaceName@.Face with record
                        delegated: @ComponentName@_Exec.CCM_@InterfaceName@_Ref;
        end record;
        type wrapper_@InterfaceName@_Ref is access all wrapper_@InterfaceName@'Class;

--      Inherited from Object.Face
```

```
        function get_component(Self: access wrapper_@InterfaceName@)
                                return Object.Ref;
</ProxyInterfaceHead>

<AttrOperationHead>
        function get_@Attr_Name@(Self: access wrapper_@PropietaryClassName@)
                return @Attr_Type@;
        procedure set_@Attr_Name@(Self: access wrapper_@PropietaryClassName@;
                value: @Attr_Type@);
</AttrOperationHead>

<AttrReadOnlyOperationHead>
        function get_@Attr_Name@(Self: access wrapper_@PropietaryClassName@)
                return @Attr_Type@;
</AttrReadOnlyOperationHead>

<ProcedureHead>
        procedure @OperationName@(Self: access
wrapper_@PropietaryClassName@</ProcedureHead>

<OperationParameter>;
                                @ParameterName@ : @ParameterQualifier@
@ParameterType@</OperationParameter>

<FirstParameterOperation>;
                                @ParameterName@ : @ParameterQualifier@
@ParameterType@</FirstParameterOperation>


<EndFunction>)
                                return @ReturnedType@;</EndFunction>
<EndProcedure>);</EndProcedure>

<FunctionHead>
        function @OperationName@(Self: access
wrapper_@PropietaryClassName@</FunctionHead>

<CreateProxyHead>
        function create(theFacet : @ComponentName@_Exec.CCM_@InterfaceName@_Ref)
                                return wrapper_@InterfaceName@_Ref;
</CreateProxyHead>

<ContextPrivateImplementationInterfaceHead>
        type @ComponentName@_Context is new
                        @ComponentName@_Exec.CCM_@ComponentName@_Context with
record</ContextPrivateImplementationInterfaceHead>

<ContextPrivateSimpleReceptacle>
                @PortName@_Recep : @InterfaceName@.Ref;</ContextPrivateSimpleReceptacle>

<ContextPrivateMultipleReceptacle>
                @PortName@_Receps :
@PortName@Connections_Ref;</ContextPrivateMultipleReceptacle>

<ContextPrivatePointerToHomeAndEnd>
--      Pointer to home for being returned in get_ccm_Home
                theHome: Components.CCMHome_Ref;
        end record;
</ContextPrivatePointerToHomeAndEnd>


<WrapperPrivateImplementationInterfaceHead>
        type @ComponentName@_wrapper is new Object.Face
                        and Components.CCMObject with
record</WrapperPrivateImplementationInterfaceHead>

<WrapperPrivateProxyPort>
                @PortName@_Facet: wrapper_@InterfaceName@_Ref;</WrapperPrivateProxyPort>

<WrapperPrivateReferencesAndEnd>
--      Executor reference:
                theExecutor: @ComponentName@_Exec.CCM_@ComponentName@_Ref;
--      Context reference:
                theContext: @ComponentName@_Context_Ref;
--      Home reference:
                theHome: Components.CCMHome_Ref;
   end record;
</WrapperPrivateReferencesAndEnd>
```

```
<WrapperHomePrivateImplementationHead>
        type @HomeName@_Wrapper is new Components.HomeConfiguration
                and Components.KeylessCCMHome and I_Set_Home_executor.Face with record
                        theHomeExecutor : @ComponentName@_Exec.CCM_@HomeName@_Ref;
                        theConfigValues: Components.ConfigValues_Ref;
        end record;
</WrapperHomePrivateImplementationHead>

<EndComponent>
end @ComponentName@;
</EndComponent>
```

Fichero *Component_WrapperTemplate_ADB.txt*

```
<Head>
--
*****************************************************************************
******
-- Project: RT-CCM
-- Group: CTR -Universidad de Cantabria
--
-- Ada ADB specification of the wrapper belong to @ElemType@ : @ElemName@
-- Date: @Date@

--
*****************************************************************************
******
</Head>

<With>
with @Package@;</With>

<Use>
use @Package@;</Use>


<ComponentHead>

package body @ComponentName@ is
</ComponentHead>

<ConnectionOperationHead>
        function get_connection_@ReceptacleName@(Self: access @ComponentName@_Context)
                return @InterfaceName@.Ref is
        begin
                return Self.@ReceptacleName@_Recep;
        end get_connection_@ReceptacleName@;
</ConnectionOperationHead>

<ConnectionMultipleOperationHead>
        function get_connections_@ReceptacleName@(Self: access @ComponentName@_Context)
                return @ReceptacleName@Connections_Ref is
        begin
                return Self.@ReceptacleName@_Receps;
        end get_connections_@ReceptacleName@;
</ConnectionMultipleOperationHead>

<ContextInterfacesInheritedHead>
--      Inherited from Components.CCMContext
        function get_CCM_home(Self: access @ComponentName@_Context)
                                return Components.CCMHome_Ref is
        begin
                return Self.theHome;
        end get_CCM_home;

--      Inherited from Components.SessionContext
        function get_CCM_object(Self: access @ComponentName@_Context)
                                return Object.Ref is
        begin
                return null;
        end get_CCM_Object;
</ContextInterfacesInheritedHead>

<SimpleTemplateOfSomeWrapperOperations>
        function get_component(Self: access @ComponentName@_Wrapper)
                                return Object.Ref is
        begin
```

```
                return Object.Ref(Self);
        end get_component;

        function get_ccm_home(Self: access @ComponentName@_Wrapper)
                return CCMHome_Ref is
        begin
                return Self.theHome;
        end get_ccm_home;

--      Raises InvalidConfiguration
        procedure configuration_complete(Self: access @ComponentName@_Wrapper) is
        begin
                Self.theExecutor.configuration_complete;
        end configuration_complete;

--      Raises  RemoveFailure
        procedure remove(Self: access @ComponentName@_Wrapper) is
        begin
                null;
        end remove;
</SimpleTemplateOfSomeWrapperOperations>

<ProvideFacetWrapperOperationHead>
        function provide_facet(Self: access @ComponentName@_Wrapper;
                               name: in FeatureName)
                               return Object.Ref is
        begin</ProvideFacetWrapperOperationHead>

<ProvideFacetIf>
                if Common_Types.To_Standard_String(Common_Types.String(name)) =
"@PortName@" then
                        return Object.Ref(Self.@PortName@_Facet);</ProvideFacetIf>
<ProvideFacetElseIf>
                elsif Common_Types.To_Standard_String(Common_Types.String(name)) =
"@PortName@" then
                        return Object.Ref(Self.@PortName@_Facet);</ProvideFacetElseIf>

<ProvideFacetEndWithoutFacets>
                return null;
        end provide_facet;
</ProvideFacetEndWithoutFacets>

<ProvideFacetEnd>
                end if;
                raise InvalidName;
        end provide_facet;
</ProvideFacetEnd>

<ConnectReceptacleHead>
        function connect(Self: access @ComponentName@_Wrapper;
                               name: in FeatureName;
                               connection:in Object.Ref) return Cookie_Ref
is</ConnectReceptacleHead>

<ConnectMultiplePortReturningStructure>
                aPort@PortName@_Connection:
@PortName@Connection;</ConnectMultiplePortReturningStructure>

<ConnectSimplePortFirstIf>
                ckValue: Integer;
        begin
                if Common_Types.To_Standard_String(Common_Types.String(name)) =
"@PortName@" then
                        if Self.theContext.@PortName@_Recep = null then
                     if connection.all in @InterfaceName@.Face'Class then
                                        Self.theContext.@PortName@_Recep :=
@InterfaceName@.Ref(connection);
                               else
                                        raise InvalidConnection;
                               end if;
                        else
                               raise AlreadyConnected;
                        end if;
                        return null;</ConnectSimplePortFirstIf>

<ConnectSimplePortElseIf>
                elsif Common_Types.To_Standard_String(Common_Types.String(name)) =
"@PortName@" then
```

```
                        if Self.theContext.@PortName@_Recep = null then
                                if connection.all in @InterfaceName@.Face'Class then
                                        Self.theContext.@PortName@_Recep :=
@InterfaceName@.Ref(connection);
                                else
                                        raise InvalidConnection;
                                end if;
                        else
                                raise AlreadyConnected;
                        end if;
                        return null;</ConnectSimplePortElseIf>

<ConnectMultiplePortFirstIf>
                ckValue: Integer;
        begin
                if Common_Types.To_Standard_String(Common_Types.String(name)) =
"@PortName@" then
                        if connection.all in @InterfaceName@.Face'Class then
                                ckValue :=
Integer(Self.theContext.@PortName@_Receps.length)+1;
                                aPort@PortName@_Connection:=
                                                (objref => @InterfaceName@.Ref(connection),
                                                ck => new
Cookie'(cookieValue=>Remote_Common_Types.octet(ckValue)));

        Self.theContext.@PortName@_Receps.Append(aPort@PortName@_Connection);
                                return aPort@PortName@_Connection.ck;
                        else
                                raise InvalidConnection;
                        end if;</ConnectMultiplePortFirstIf>

<ConnectMultiplePortElseIf>
                elsif Common_Types.To_Standard_String(Common_Types.String(name)) =
"@PortName@" then
                        if connection.all in @InterfaceName@.Face'Class then
                                ckValue :=
Integer(Self.theContext.@PortName@_Receps.length)+1;
                                aPort@PortName@_Connection:=
                                                (objref => @InterfaceName@.Ref(connection),
                                                ck => new
Cookie'(cookieValue=>Remote_Common_Types.octet(ckValue)));

        Self.theContext.@PortName@_Receps.Append(aPort@PortName@_Connection);
                                return aPort@PortName@_Connection.ck;
                        else
                                raise InvalidConnection;
                        end if;</ConnectMultiplePortElseIf>

<ConnectReceptacleEnd>
                end if;
                raise InvalidName;
        end connect;
</ConnectReceptacleEnd>

<ConnectWithoutReceptacleEnd>
        begin
                return null;
        end connect;
</ConnectWithoutReceptacleEnd>

<DisconnectReceptacleHead>
        procedure disconnect(Self: access @ComponentName@_Wrapper;
                                name: in FeatureName;
                                ck: in Cookie_Ref) is</DisconnectReceptacleHead>

<DisconnectSimplePortFirstIf>
        begin
                if Common_Types.To_Standard_String(Common_Types.String(name)) =
"@PortName@" then
                        if Self.theContext.@PortName@_Recep = null then
                                raise NoConnection;
                        else
                                Self.theContext.@PortName@_Recep := null;
                        end if;</DisconnectSimplePortFirstIf>

<DisconnectSimplePortElseIf>
                elsif Common_Types.To_Standard_String(Common_Types.String(name)) =
"@PortName@" then
```

```
                        if Self.theContext.@PortName@_Recep = null then
                                raise NoConnection;
                        else
                                Self.theContext.@PortName@_Recep := null;
                        end if;</DisconnectSimplePortElseIf>

<DisconnectMultiplePortFirstIf>
        begin
                if Common_Types.To_Standard_String(Common_Types.String(name)) =
"@PortName@" then
                        if ck = null then
                                raise CookieRequired;
                        else
                                for i in 1..Self.theContext.@PortName@_Receps.length loop
                                        if
Self.theContext.@PortName@_Receps.element(Positive(i)).ck = ck then

        Self.theContext.@PortName@_Receps.delete(Positive(i));
                                                return;
                                        end if;
                                end loop;
                                raise InvalidConnection;
                        end if;</DisconnectMultiplePortFirstIf>

<DisconnectMultiplePortElseIf>
                elsif Common_Types.To_Standard_String(Common_Types.String(name)) =
"@PortName@" then
                        if ck = null then
                                raise CookieRequired;
                        else
                                for i in 1..Self.theContext.@PortName@_Receps.length loop
                                        if
Self.theContext.@PortName@_Receps.element(Positive(i)).ck = ck then

        Self.theContext.@PortName@_Receps.delete(Positive(i));
                                                return;
                                        end if;
                                end loop;
                                raise InvalidConnection;
                        end if;</DisconnectMultiplePortElseIf>

<DisconnectReceptacleEnd>
                end if;
                raise InvalidName;
        end disconnect;
</DisconnectReceptacleEnd>

<DisconnectWithoutReceptacleEnd>
        begin
                null;
        end disconnect;
</DisconnectWithoutReceptacleEnd>

<ConfigureComponentHead>
        procedure configure_component (Self:@ComponentName@_Wrapper_Ref;
                                theValues: Components.ConfigValues_Ref) is
                aConfigValue: ConfigValue;
        begin
                for i in 1..theValues.length loop
                        aConfigValue:=theValues.element(Positive(i));
                        if </ConfigureComponentHead>

<ConfigureComponentAttrFirstIf>Common_Types.To_Standard_String
                                        (Common_Types.String(aConfigValue.name)) =
"@Attr_Name@" then
                                Self.theExecutor.set_@Attr_Name@

        (@PackageTypes@.Any_To_@Attr_Type@(aconfigValue.value))@CloseParenthesisValue@;<
/ConfigureComponentAttrFirstIf>

<ConfigureComponentActivationElementsFirstIf>
        procedure configure_component (Self:@ComponentName@_Wrapper_Ref;
                                theValues: Components.ConfigValues_Ref) is
                aConfigValue: ConfigValue;
        begin
                for i in 1..theValues.length loop
                        aConfigValue:=theValues.element(Positive(i));
                        if Common_Types.To_Standard_String
```

```
                                        (Common_Types.String(aConfigValue.name)) =
"@PortName@_SchedParamId" then
                                Self.theExecutor.set_@PortName@_SchedParamID

        (SchedulingControlService.SchedControlID(Natural'Value(Common_Types.Any_To_Stand
ard_String(aconfigValue.value))));</ConfigureComponentActivationElementsFirstIf>

<ConfigureComponentActivationElementsElseIf>
                        elsif Common_Types.To_Standard_String
                                        (Common_Types.String(aConfigValue.name)) =
"@PortName@_SchedParamId" then
                                Self.theExecutor.set_@PortName@_SchedParamID

        (SchedulingControlService.SchedControlID(Natural'Value(Common_Types.Any_To_Stand
ard_String(aconfigValue.value))));</ConfigureComponentActivationElementsElseIf>

<ConfigureComponentPeriodicActivation>
                        elsif Common_Types.To_Standard_String
                                        (Common_Types.String(aConfigValue.name)) =
"@PortName@_Period" then
                                Self.theExecutor.set_@PortName@_Period

        (Duration'Value(Common_Types.Any_To_Standard_String(aconfigValue.value)));</Conf
igureComponentPeriodicActivation>

<ConfigureComponentAttrElseIf>
                        elsif Common_Types.To_Standard_String
                                        (Common_Types.String(aConfigValue.name)) =
"@Attr_Name@" then
                                Self.theExecutor.set_@Attr_Name@

        (@PackageTypes@.Any_To_@Attr_Type@(aconfigValue.value))@CloseParenthesisValue@;<
/ConfigureComponentAttrElseIf>

<ConfigureComponentEndIfThereAreNotAttributes>
        procedure configure_component (Self:@ComponentName@_Wrapper_Ref;
                                theValues: Components.ConfigValues_Ref) is
        begin
                null;
        end configure_component;
</ConfigureComponentEndIfThereAreNotAttributes>

<ConfigureComponentEnd>
                        end if;
                end loop;
        end configure_component;
</ConfigureComponentEnd>

<HomeExplicitOperationHead>
        function @OperationName@(Self: access
@HomeName@_Wrapper</HomeExplicitOperationHead>

<HomeExplicitOperationParameter>;
                                @OperationParameter@ :
@ParameterType@</HomeExplicitOperationParameter>

<HomeExplicitOperationEnd>)
                        return @ComponentName@_EqInt.@ComponentName@_Ref is
                theComp : @ComponentName@_Wrapper_Ref;
                theExec: Components.EnterpriseComponent_Ref;
                ctx: @ComponentName@_Context_Ref;
        begin
                theComp := new @ComponentName@_Wrapper;
--      Executor creation
                theExec :=
Self.theHomeExecutor.@OperationName@</HomeExplicitOperationEnd>

<HomeExplicitOperationFirstParameterBody>(@OperationParameter@</HomeExplicitOperationFi
rstParameterBody>

<HomeExplicitOperationParameterBody>,@OperationParameter@</HomeExplicitOperationParamet
erBody>

<HomeExplicitHead>@ParametersParenthesis@;
                theComp.theExecutor := CCM_@ComponentName@_Ref(theExec);
--      Context creation
                ctx:= new @ComponentName@_Context;</HomeExplicitHead>
```

```
<HomeExplicitReceptacleList>
                ctx.@PortName@_Receps := new
@PortName@Connections;</HomeExplicitReceptacleList>

<HomeExplicitBody>
                theComp.theContext:=ctx;

        theComp.theExecutor.set_session_context(SessionContext_Ref(ctx));</HomeExplicitB
ody>

<HomeExplicitProxyFacet>
                theComp.@PortName@_Facet:=
create(theComp.theExecutor.get_@PortName@);</HomeExplicitProxyFacet>

<HomeExplicitEnd>
                configure_component(theComp,Self.theConfigValues);
                return @ComponentName@_EqInt.@ComponentName@_Ref(theComp);
        end @OperationName@;
</HomeExplicitEnd>


<CreateHead>
        function create(Self: access @HomeName@_Wrapper)
                    return @ComponentName@_EqInt.@ComponentName@_Ref is
                theComp : @ComponentName@_Wrapper_Ref;
                theExec: Components.EnterpriseComponent_Ref;
                ctx: @ComponentName@_Context_Ref;
        begin
                theComp := new @ComponentName@_Wrapper;
--      Executor creation
                theExec := Self.theHomeExecutor.create;
                theComp.theExecutor := CCM_@ComponentName@_Ref(theExec);
--      Context creation
                ctx:= new @ComponentName@_Context;</CreateHead>

<CreateReceptacleList>
                ctx.@PortName@_Receps := new
@PortName@Connections;</CreateReceptacleList>

<CreateBody>
                theComp.theContext:=ctx;

        theComp.theExecutor.set_session_context(SessionContext_Ref(ctx));</CreateBody>

<CreateProxyFacet>
                theComp.@PortName@_Facet:=
create(theComp.theExecutor.get_@PortName@);</CreateProxyFacet>

<CreateEnd>
                configure_component(theComp,Self.theConfigValues);
                return @ComponentName@_EqInt.@ComponentName@_Ref(theComp);
        end create;
</CreateEnd>

<CreateComponentHead>
        function create_component(Self: access @HomeName@_Wrapper)
                    return CCMObject_Ref is
                theComp : @ComponentName@_Wrapper_Ref;
                theExec: Components.EnterpriseComponent_Ref;
                ctx: @ComponentName@_Context_Ref;
        begin
                theComp := new @ComponentName@_Wrapper;
--      Executor creation
                theExec := Self.theHomeExecutor.create;
                theComp.theExecutor := CCM_@ComponentName@_Ref(theExec);
--      Context creation
                ctx:= new @ComponentName@_Context;</CreateComponentHead>


<CreateComponentEnd>
                configure_component(theComp,Self.theConfigValues);
                return CCMObject_Ref(theComp);
        end create_component;
</CreateComponentEnd>

<SpecialConnectorWrapperLine>
                @ConnectorName@_Adapter.Set_Connector_Impl(TheComp);
</SpecialConnectorWrapperLine>
```

```
<SimpleTemplateOfSomeHomeWrapperOperations>
        procedure remove_component(Self: access @HomeName@_Wrapper;
                                   comp: CCMObject_Ref) is
        begin
                null;
        end remove_component;

        procedure set_configuration_values(Self: access @HomeName@_Wrapper;
                                   config: ConfigValues_Ref) is
        begin
                Self.theConfigValues := config;
        end set_configuration_values;

        procedure complete_component_configuration(Self: access @HomeName@_Wrapper;
                                        b: in boolean) is
        begin
                null;
        end complete_component_configuration;

        procedure disable_home_configuration(Self: access @HomeName@_Wrapper) is
        begin
                null;
        end disable_home_configuration;

        procedure set_home_executor(Self: access @HomeName@_Wrapper;
                                   theHome: in Components.HomeExecutorBase_Ref) is
        begin
                Self.theHomeExecutor := CCM_@HomeName@_Ref(theHome);
        end set_home_executor;

        function create return Components.CCMHome_Ref is
        begin
                return new @HomeName@_Wrapper;
        end create;
</SimpleTemplateOfSomeHomeWrapperOperations>

<GetComponentHead>
        function get_component(Self: access wrapper_@InterfaceName@) return Object.Ref
is
        begin
                return null;
        end get_component;
</GetComponentHead>

<AttrOperationHead>
        function get_@Attr_Name@(Self: access wrapper_@PropietaryClassName@)
                return @Attr_Type@ is
        begin
            return self.delegated.get_@Attr_Name@;
        end get_@Attr_Name@;
        procedure set_@Attr_Name@(Self: access wrapper_@PropietaryClassName@;
                value: in @Attr_Type@) is
        begin
                Self.delegated.set_@Attr_Name@(value);
        end set_@Attr_Name@;
</AttrOperationHead>

<AttrReadOnlyOperationHead>
        function get_@Attr_Name@(Self: access wrapper_@PropietaryClassName@)
                return @Attr_Type@ is
        begin
            return self.delegated.get_@Attr_Name@;
        end get_@Attr_Name@;
</AttrReadOnlyOperationHead>

<FunctionHead>
        function @OperationName@(Self : access
wrapper_@PropietaryClassName@</FunctionHead>
<OperationParameter>;
                                                @ParameterName@ :
@ParameterQualifier@ @ParameterType@</OperationParameter>

<FirstParameterOperation>;
                                                @ParameterName@ :
@ParameterQualifier@ @ParameterType@</FirstParameterOperation>
```

```
<EndFunction>)
                return @ReturnedType@ is
        begin
                return Self.delegated.@OperationName@;
        end @OperationName@;
</EndFunction>

<ProcedureHead>
        procedure @OperationName@(Self : access
wrapper_@PropietaryClassName@</ProcedureHead>
<EndProcedure>) is
        begin
                Self.delegated.@OperationName@;
        end @OperationName@;
</EndProcedure>

<FirstParameterBodyProcedure>) is
        begin

        Self.delegated.@OperationName@(@ParameterName@</FirstParameterBodyProcedure>

<FirstParameterBodyFunction>)
                return @ReturnedType@ is
        begin
                return
Self.delegated.@OperationName@(@ParameterName@</FirstParameterBodyFunction>


<ParametersBodyOperation>, @ParameterName@</ParametersBodyOperation>

<EndOperationWithParameters>);
        end @OperationName@;
</EndOperationWithParameters>

<CreateProxyHead>
        function create(theFacet : @ComponentName@_Exec.CCM_@InterfaceName@_Ref)
                                return wrapper_@InterfaceName@_Ref is
                toReturn : wrapper_@InterfaceName@_Ref;
        begin
                toReturn := new wrapper_@InterfaceName@;
                toReturn.delegated := theFacet;
                return toReturn;
        end create;
</CreateProxyHead>

<EndComponent>
end @ComponentName@;
</EndComponent>
```

PLANTILLA PARA LA GENERACIÓN DE LA INTERFAZ EQUIVALENTE DEL COMPONENTE

Fichero *Component_EqIntTemplate.txt*

```
<Head>
--
*******************************************************************************
******
-- Project: RT-CCM
-- Group: CTR -Universidad de Cantabria
--
-- Ada specification of the equivalent interface of component @ElemName@
-- Date: @Date@

--
*******************************************************************************
******
</Head>

<With>
with @Package@;</With>
```

```
<Use>
use @Package@;</Use>

<InterfaceHead>
package @ComponentName@_EqInt is

        type @ComponentName@ is interface and Components.CCMObject;
        type @ComponentName@_Ref is access all @ComponentName@'Class;
</InterfaceHead>

<HomeExplicitHead>
        type @HomeName@Explicit is interface and
                        Components.HomeConfiguration;
</HomeExplicitHead>

<HomeImplicitHead>
        type @HomeName@Implicit is interface and Components.KeylessCCMHome;
        function create(Self: access @HomeName@Implicit)
                                return @ComponentName@_Ref is abstract;
</HomeImplicitHead>

<HomeHead>
        type @HomeName@ is interface
                        and @HomeName@Explicit
                        and @HomeName@Implicit;
        type @HomeName@_Ref is access all @HomeName@'Class;
</HomeHead>

<HomeExplicitOperationHead>
        function @OperationName@(Self: access
@HomeName@Explicit</HomeExplicitOperationHead>

<HomeExplicitOperationParameter>;
                                @OperationParameter@ :
@ParameterType@</HomeExplicitOperationParameter>

<HomeExplicitOperationEnd>)
                return @ComponentName@_Ref is abstract;
</HomeExplicitOperationEnd>

<EndComponent>
end @ComponentName@;
</EndComponent>
```

PLANTILLA PARA LA GENERACIÓN DE LOS PAQUETES ADS DE LAS FACETAS

Fichero *Component_FacetInterfacesTemplate.txt*

```
<Head>
--
*******************************************************************************
******
-- Project: RT-CCM
-- Group: CTR -Universidad de Cantabria
--
-- Ada specification of the facet interface @ElemType@ of component @ElemName@
-- Date: @Date@

--
*******************************************************************************
******
</Head>

<With>
with @Package@;</With>

<Use>
use @Package@;</Use>

<InterfaceHead>
package @ComponentName@_facet@InterfaceName@ is
```

```
        type Face is interface and @ComponentName@_Exec.CCM_@InterfaceName@;
        type Ref is access all Face'Class;

end @ComponentName@_facet@InterfaceName@;
</InterfaceHead>
```

PLANTILLAS PARA LA GENERACIÓN DEL LOS PAQUETES ADS/ADB DE LOS ADAPTER DE LOS CONECTORES

Fichero *Connector_Servant_AdapterTemplate_ADS.txt*

```
<Head>
--
*******************************************************************************
******
-- Project: RT-CCM
-- Group: CTR -Universidad de Cantabria
--
-- Ada ADS specification of connector RCI belong to @ElemType@ : @ElemName@
-- Date: @Date@

--
*******************************************************************************
******
</Head>

<With>
with @Package@;</With>

<Use>
use @Package@;</Use>

<ComponentHead>

package @ComponentName@ is
</ComponentHead>

<AttrOperationHead>
        function get_@Attr_Name@ return @Attr_Type@;
        procedure set_@Attr_Name@(newValue: @Attr_Type@);
</AttrOperationHead>

<AttrReadOnlyOperationHead>
        function get_@Attr_Name@ return @Attr_Type@;
</AttrReadOnlyOperationHead>

<FunctionHead>
        function @OperationName@</FunctionHead>
<OperationParameter></OperationParameter>
<EndFunction>@PosibleAdsEnd@
                return @ReturnedType@;
</EndFunction>

<ProcedureHead>
        procedure @OperationName@</ProcedureHead>

<EndProcedure>@PosibleAdsEnd@;
</EndProcedure>

<FirstParameterOperation>(@ParameterName@ : @ParameterQualifier@
@ParameterType@</FirstParameterOperation>

<OperationParameter>;
                                @ParameterName@ : @ParameterQualifier@
@ParameterType@</OperationParameter>


<EndOperationWithParameters>)@PosibleFunctionEnd@;
</EndOperationWithParameters>

<SetConnectorImplHead>
        procedure
```

```
Set_Connector_Impl(TheConnector:@ConnectorName@_Wrapper.@ConnectorName@_Wrapper_Ref);

private

        connectorWrapper : @ConnectorName@_Wrapper.@ConnectorName@_Wrapper_Ref;
</SetConnectorImplHead>

<EndComponent>
end @ComponentName@;
</EndComponent>
```

Fichero *Connector_Servant_AdapterTemplate_ADB.txt*

```
<Head>
--
*******************************************************************************
******
-- Project: RT-CCM
-- Group: CTR -Universidad de Cantabria
--
-- Ada ADB specification of implementation belong to @ElemType@ : @ElemName@
-- Date: @Date@

--
*******************************************************************************
******
</Head>

<With>
with @Package@;</With>

<Use>
use @Package@;</Use>


<ComponentHead>

package body @ComponentName@ is
</ComponentHead>

<AttrOperationHead>
        function get_@Attr_Name@ return @Attr_Type@ is
                facetObject : Object.Ref;
                facet : @InterfaceName@.Ref;
        begin
                if connectorWrapper /= null then
                        facetObject := connectorWrapper.provide_facet

        (Components.FeatureName(Common_Types.To_String("@FacetPortName@")));
                        facet:= @InterfaceName@.Ref(facetObject);
                return @ConversionToUserType@;
                else
                        raise Program_Error; -- this is temporal
                end if;
        end get_@Attr_Name@;
        procedure set_@Attr_Name@(newValue: in @Attr_Type@) is
                facetObject : Object.Ref;
                facet : @InterfaceName@.Ref;
        begin
                if connectorWrapper /= null then
                        facetObject := connectorWrapper.provide_facet

        (Components.FeatureName(Common_Types.To_String("@FacetPortName@")));
                        facet:= @InterfaceName@.Ref(facetObject);
                        facet.set_@Attr_Name@(@ConversionToGladeType@);
                else
                        raise Program_Error; -- this is temporal
                end if;
        end set_@Attr_Name@;
</AttrOperationHead>

<AttrReadOnlyOperationHead>
        function get_@Attr_Name@ return @Attr_Type@ is
                facetObject : Object.Ref;
                facet : @InterfaceName@.Ref;
        begin
                if connectorWrapper /= null then
                        facetObject := connectorWrapper.provide_facet
```

```
        (Components.FeatureName(Common_Types.To_String("@FacetPortName@")));
                        facet:= @InterfaceName@.Ref(facetObject);
                        return @ConversionToUserType@;
                else
                        raise Program_Error; -- this is temporal
                end if;
        end get_@Attr_Name@;
</AttrReadOnlyOperationHead>

<FunctionHead>
        function @OperationName@</FunctionHead>
<OperationParameter>;
                                                        @ParameterName@ :
@ParameterQualifier@ @ParameterType@</OperationParameter>

<FirstParameterOperation>(@ParameterName@ : @ParameterQualifier@
@ParameterType@</FirstParameterOperation>


<EndFunction>
                        return @ReturnedType@ is
                facetObject : Object.Ref;
                facet : @InterfaceName@.Ref;
        begin
                if connectorWrapper /= null then
                        facetObject := connectorWrapper.provide_facet

        (Components.FeatureName(Common_Types.To_String("@FacetPortName@")));
                        facet:= @InterfaceName@.Ref(facetObject);
                        return
@GladeReturnedTypeConversion@facet.@OperationName@@GladeParenthesisEnd@;
                else
                        raise Program_Error; -- this is temporal
                end if;
        end @OperationName@;
</EndFunction>

<ProcedureHead>
        procedure @OperationName@</ProcedureHead>
<EndProcedure> is
                facetObject : Object.Ref;
                facet : @InterfaceName@.Ref;
        begin
                if connectorWrapper /= null then
                        facetObject := connectorWrapper.provide_facet

        (Components.FeatureName(Common_Types.To_String("@FacetPortName@")));
                        facet:= @InterfaceName@.Ref(facetObject);
                        facet.@OperationName@;
                else
                        raise Program_Error; -- this is temporal
                end if;
        end @OperationName@;
</EndProcedure>

<FirstParameterBodyProcedure>) is
                facetObject : Object.Ref;
                facet : @InterfaceName@.Ref;
        begin
                if connectorWrapper /= null then
                        facetObject := connectorWrapper.provide_facet

        (Components.FeatureName(Common_Types.To_String("@FacetPortName@")));
                        facet:= @InterfaceName@.Ref(facetObject);

        facet.@OperationName@(@GladeParameterTypeConversion@</FirstParameterBodyProcedur
e>

<FirstParameterBodyFunction>)
                        return @ReturnedType@ is
                facetObject : Object.Ref;
                facet : @InterfaceName@.Ref;
        begin
                if connectorWrapper /= null then
                        facetObject := connectorWrapper.provide_facet

        (Components.FeatureName(Common_Types.To_String("@FacetPortName@")));
```

```
                        facet:= @InterfaceName@.Ref(facetObject);
                        return
@GladeReturnedTypeConversion@facet.@OperationName@(@GladeParameterTypeConversion@</Firs
tParameterBodyFunction>

<ParametersBodyOperation>; @GladeParameterTypeConversion@</ParametersBodyOperation>

<EndOperationWithParameters>@GladeParenthesisEnd@);
                else
                        raise Program_Error; -- this is temporal
                end if;
        end @OperationName@;
</EndOperationWithParameters>

<UseConnectorWrapperRefHead>
        use type @ConnectorName@_Wrapper.@ConnectorName@_Wrapper_Ref;
</UseConnectorWrapperRefHead>

<SetConnectorImplHead>
        procedure set_connector_impl(theConnector:
@ConnectorName@_Wrapper.@ConnectorName@_Wrapper_Ref) is
        begin
                connectorWrapper :=theConnector;
        end set_connector_impl;
</SetConnectorImplHead>

<EndComponent>
end @ComponentName@;
</EndComponent>
```

PLANTILLAS PARA LA GENERACIÓN DEL LOS PAQUETES ADS/ADB DE LOS RCI DE LOS CONECTORES

Fichero *Connector_Servant_RCITemplate_ADS.txt*

```
<Head>
--
*******************************************************************************
******
-- Project: RT-CCM
-- Group: CTR -Universidad de Cantabria
--
-- Ada ADS specification of connector RCI belong to @ElemType@ : @ElemName@
-- Date: @Date@

--
*******************************************************************************
******
</Head>

<With>
with @Package@;</With>

<Use>
use @Package@;</Use>

<ComponentHead>

package @ComponentName@ is
</ComponentHead>

<PragmaRCI>
        pragma Remote_Call_Interface;
</PragmaRCI>

<AttrOperationHead>
        function get_@Attr_Name@ return @Attr_Type@;
        procedure set_@Attr_Name@(newValue: @Attr_Type@);
</AttrOperationHead>

<AttrReadOnlyOperationHead>
        function get_@Attr_Name@ return @Attr_Type@;
</AttrReadOnlyOperationHead>
```

```
<FunctionHead>
        function @OperationName@</FunctionHead>
<OperationParameter></OperationParameter>
<EndFunction>@PosibleAdsEnd@
               return @ReturnedType@;
</EndFunction>

<ProcedureHead>
        procedure @OperationName@</ProcedureHead>

<EndProcedure>@PosibleAdsEnd@;
</EndProcedure>

<FirstParameterOperation>(@ParameterName@ : @ParameterQualifier@
@ParameterType@</FirstParameterOperation>

<OperationParameter>;
                                @ParameterName@ : @ParameterQualifier@
@ParameterType@</OperationParameter>


<EndOperationWithParameters>)@PosibleFunctionEnd@;
</EndOperationWithParameters>

<EndComponent>
end @ComponentName@;
</EndComponent>
```

Fichero *Connector_Servant_RCITemplate_ADB.txt*

```
<Head>
--
*******************************************************************************
******
-- Project: RT-CCM
-- Group: CTR -Universidad de Cantabria
--
-- Ada ADB specification of implementation belong to @ElemType@ : @ElemName@
-- Date: @Date@

--
*******************************************************************************
******
</Head>

<With>
with @Package@;</With>

<Use>
use @Package@;</Use>


<ComponentHead>

package body @ComponentName@ is
</ComponentHead>

<AttrOperationHead>
        function get_@Attr_Name@ return @Attr_Type@ is
        begin
            return @ConnectorID@.get_@Attr_Name@;
        end get_@Attr_Name@;
        procedure set_@Attr_Name@(newValue: in @Attr_Type@) is
        begin
                @ConnectorID@.set_@Attr_Name@(newValue);
        end set_@Attr_Name@;
</AttrOperationHead>

<AttrReadOnlyOperationHead>
        function get_@Attr_Name@ return @Attr_Type@ is
        begin
            return @ConnectorID@.get_@Attr_Name@;
        end get_@Attr_Name@;
</AttrReadOnlyOperationHead>
```

```
<FunctionHead>
        function @OperationName@</FunctionHead>
<OperationParameter>;
                                                        @ParameterName@ :
@ParameterQualifier@ @ParameterType@</OperationParameter>

<FirstParameterOperation>(@ParameterName@ : @ParameterQualifier@
@ParameterType@</FirstParameterOperation>


<EndFunction>
                return @ReturnedType@ is
        begin
                return @ConnectorID@.@OperationName@;
        end @OperationName@;
</EndFunction>

<ProcedureHead>
        procedure @OperationName@</ProcedureHead>
<EndProcedure> is
        begin
                @ConnectorID@.@OperationName@;
        end @OperationName@;
</EndProcedure>

<FirstParameterBodyProcedure>) is
        begin

        @ConnectorID@.@OperationName@(@ParameterName@</FirstParameterBodyProcedure>

<FirstParameterBodyFunction>)
                return @ReturnedType@ is
        begin
                return
@ConnectorID@.@OperationName@(@ParameterName@</FirstParameterBodyFunction>


<ParametersBodyOperation>; @ParameterName@</ParametersBodyOperation>

<EndOperationWithParameters>);
        end @OperationName@;
</EndOperationWithParameters>

<EndComponent>
end @ComponentName@;
</EndComponent>
```

PLANTILLA PARA LA GENERACIÓN DEL LOS PAQUETES [NOMBRECONECTOR]_IMPL DE LOS CONECTORES

Fichero *Connector_ImplTemplate_ADB.txt*

```
<Head>
--
*******************************************************************************
******
-- Project: RT-CCM
-- Group: CTR -Universidad de Cantabria
--
-- Ada ADB specification of implementation belong to @ElemType@ : @ElemName@
-- Date: @Date@

--
*******************************************************************************
******
</Head>

<With>
with @Package@;</With>

<Use>
use @Package@;</Use>
```

```
<ComponentHead>

package body @ComponentName@ is
</ComponentHead>

<AttrOperationHead>
        function get_@Attr_Name@(Self: access Instance)
                return @Attr_Type@ is
        begin
                return Self.state.@Attr_Name@;
        end get_@Attr_Name@;
        procedure set_@Attr_Name@(Self: access Instance;
                newValue: @Attr_Type@) is
        begin
                self.state.@Attr_Name@:=newValue;
        end set_@Attr_Name@;
</AttrOperationHead>

<AttrReadOnlyOperationHead>
        function get_@Attr_Name@(Self: access Instance)
                return @Attr_Type@ is
        begin
                return Self.state.@Attr_Name@;
        end get_@Attr_Name@;
</AttrReadOnlyOperationHead>

<FacetAccessOperationHead>
        function get_@PortName@(Self: access Instance)
                return @ComponentName@_Facet@InterfaceName@.Ref is
        begin
                return @ComponentName@_Facet@InterfaceName@.Ref(Self.@PortName@);
        end get_@PortName@;</FacetAccessOperationHead>

<ConnectionOperationHead>
        procedure @ReceptacleName@_Connect(self:access Instance;
face:@InterfaceName@.Ref) is
        begin
                Self.state.@ReceptacleName@:=face;
        end @ReceptacleName@_Connect;
</ConnectionOperationHead>

<ConnectionMultipleOperationHead>
        procedure @ReceptacleName@_Cardinality(self:access Instance; numObj:Natural) is
        begin
                self.state.@ReceptacleName@_num:=numObj;
                self.state.@ReceptacleName@:=new @ReceptacleName@_Container(1..numObj);
        end @ReceptacleName@_Cardinality;

        procedure @ReceptacleName@_Connect(self:access Instance;
                                index:Positive; face:@InterfaceName@.Ref) is
        begin
                self.state.@ReceptacleName@(index):=face;
        end @ReceptacleName@_Connect;
</ConnectionMultipleOperationHead>

<OSASettingOperationHead>
        procedure set_@PortName@_SchedParamID(self:access Instance;
                                id:SchedulingControlService.SchedControlID) is
        begin
                self.state.@PortName@.schedulingId:=id;
        end set_@PortName@_SchedParamID;
</OSASettingOperationHead>

<PASettingOperationHead>
        procedure set_@PortName@_Period(self:access Instance;period:Duration) is
        begin
                self.state.@PortName@.period:=period;
        end set_@PortName@_Period;
        procedure set_@PortName@_SchedParamID(self:access Instance;
                                id:SchedulingControlService.SchedControlID) is
        begin
                self.state.@PortName@.schedulingId:=id;
        end set_@PortName@_SchedParamID;
</PASettingOperationHead>

<CreateHead>
-- Is invoked before component and facets attributes setting
```

```
        function create return Ref is
                newInstance:Ref;
        begin
-- Create the component element
                newInstance:=new Instance;
-- Create the state element
                newInstance.state:=create_@ComponentName@_State;
-- Create facet elements</CreateHead>
<CreateFacetElement>
                newInstance.@PortName@:=

        create_@PortName@_Port(newInstance.state);</CreateFacetElement>
<CreateEnd>
-- Return instance
                return newInstance;
        end create;
</CreateEnd>
<CompleteHead>
-- Is invoked after component and facet attributed are set.
        procedure complete(self:access Instance) is
        begin
-- Complete component state
                complete_@ComponentName@_State(self.state);
-- complete facet ports</CompleteHead>
<CompleteFacetElement>
                complete_@PortName@_Port(self.@PortName@);</CompleteFacetElement>

<CompleteEnd>
        end complete;
</CompleteEnd>

<ActivateHead>
-- Life control inherited from session
        procedure activate(self: access Instance) is
        begin
-- if the first time create them</ActivateHead>

<ActivateOSAElement>
                if (self.@PortName@=null)
                then
                        self.state.@PortName@:=new OneShotActivationBlock;
                        self.state.@PortName@.state:=self.State;
                        self.state.@PortName@.run:=@PortName@_Run'access;
                        self.state.@PortName@.controller:= new
OneShotActivationController;
                        self.@PortName@:=new
OneShotActivationTask(self.state.@PortName@);
                end if;
-- Anyway activates the controller
                self.state.@PortName@.controller.activate;
</ActivateOSAElement>

<ActivatePAElement>
                if (self.@PortName@=null)
                then
                        self.state.@PortName@:=new PeriodicActivationBlock;
                        self.state.@PortName@.state:=self.State;
                        self.state.@PortName@.update:=@PortName@_Update'access;
                        self.state.@PortName@.controller:= new
PeriodicActivationController;
                        self.@PortName@:=new
PeriodicActivationTask(self.state.@PortName@);
                end if;
-- Anyway activates the controller
                self.state.@PortName@.controller.activate;
</ActivatePAElement>

<ActivateEnd>
        end activate;
</ActivateEnd>

<ActivateWithoutActivationPortsEnd>
                null;
        end activate;
</ActivateWithoutActivationPortsEnd>

<PassivateHead>
        procedure passivate(self: access Instance) is
```

```
        begin</PassivateHead>
<PassivateElement>
                self.state.@PortName@.controller.passivate;</PassivateElement>
<PassivateEnd>
        end passivate;
</PassivateEnd>

<PassivateWithoutActivationPortsEnd>
                null;
        end passivate;
</PassivateWithoutActivationPortsEnd>

<RemoveHead>
        procedure remove (self: access Instance) is
        begin</RemoveHead>

<RemoveElement>
                self.state.@PortName@.controller.remove;</RemoveElement>

<RemoveEnd>
        end remove;
</RemoveEnd>

<RemoveWithoutActivationPortsEnd>
                null;
        end remove;
</RemoveWithoutActivationPortsEnd>


<CreateComponentStateHead>
        function create_@ComponentName@_State return @ComponentName@_State_Ref is
                newState:@ComponentName@_State_Ref;
        begin
                newState:=new @ComponentName@_State;</CreateComponentStateHead>

<CreateComponentStateReceptaclesAndActivationPorts>

        newState.@PortName@:=null;</CreateComponentStateReceptaclesAndActivationPorts>

<CreateComponentStateComplexReceptacles>
                newState.@PortName@_num:=0;
                newState.@PortName@:=null;</CreateComponentStateComplexReceptacles>


<CreateComponentStateEnd>
                return newState;
        end create_@ComponentName@_State;
</CreateComponentStateEnd>

<CompleteConnectorStateHead>
        procedure complete_@ComponentName@_State(state:@ComponentName@_State_Ref)is
        begin
                null; -- ONLY FOR COMPILING IF THERE ARE NOT CODE HERE
        end complete_@ComponentName@_State;
</CompleteConnectorStateHead>

<BussinessConnectorFacetPortOperations>
        function create_@PortName@_Port(state:@ComponentName@_State_Ref)
                                return @PortName@_Port_Ref is
                newPort:@PortName@_Port_Ref;
        begin
                newPort:=new @PortName@_Port;
                newPort.state:=state;

                return newPort;
        end create_@PortName@_Port;

        procedure complete_@PortName@_Port(port:@PortName@_Port_Ref)is
        begin
                null;
        end complete_@PortName@_Port;
</BussinessConnectorFacetPortOperations>


<AttrOperationHead_2>
        function get_@Attr_Name@(Self: access @PropietaryClassName@_Port)
                return @Attr_Type@ is
        begin
```

```
            return @ConversionToUserType@;
        end get_@Attr_Name@;
        procedure set_@Attr_Name@(Self: access @PropietaryClassName@_Port;
                newValue: in @Attr_Type@) is
        begin
                @ConnectorID@.set_@Attr_Name@(@ConversionToGladeType@);
        end set_@Attr_Name@;
</AttrOperationHead_2>

<AttrReadOnlyOperationHead_2>
        function get_@Attr_Name@(Self: access @PropietaryClassName@_Port)
                return @Attr_Type@ is
        begin
            return @ConversionToUserType@;
        end get_@Attr_Name@;
</AttrReadOnlyOperationHead_2>

<AttrOperationHead_3>
        function get_@Attr_Name@(Self: access @PropietaryClassName@_Port)
                return @Attr_Type@ is
        begin
            return @ConnectorID@.get_@Attr_Name@;
        end get_@Attr_Name@;
        procedure set_@Attr_Name@(Self: access @PropietaryClassName@_Port;
                newValue: in @Attr_Type@) is
        begin
                @ConnectorID@.set_@Attr_Name@(newValue);
        end set_@Attr_Name@;
</AttrOperationHead_3>

<AttrReadOnlyOperationHead_3>
        function get_@Attr_Name@(Self: access @PropietaryClassName@_Port)
                return @Attr_Type@ is
        begin
            return @ConnectorID@.get_@Attr_Name@;
        end get_@Attr_Name@;
</AttrReadOnlyOperationHead_3>

<FunctionHead>
        function @OperationName@(Self : access @PropietaryClassName@_Port</FunctionHead>
<OperationParameter>;
                                                        @ParameterName@ :
@ParameterQualifier@ @ParameterType@</OperationParameter>

<FirstParameterOperation>;
                                                        @ParameterName@ :
@ParameterQualifier@ @ParameterType@</FirstParameterOperation>


<EndFunction>)
                return @ReturnedType@ is
        begin
                return @ConnectorID@.@OperationName@@GladeParenthesisEnd@;
        end @OperationName@;
</EndFunction>

<ProcedureHead>
        procedure @OperationName@(Self : access
@PropietaryClassName@_Port</ProcedureHead>
<EndProcedure>) is
        begin
                @ConnectorID@.@OperationName@;
        end @OperationName@;
</EndProcedure>

<FirstParameterBodyProcedure>) is
        begin

        @ConnectorID@.@OperationName@(@ParameterName@</FirstParameterBodyProcedure>

<FirstParameterBodyFunction>)
                return @ReturnedType@ is
        begin
                return
@ConnectorID@.@OperationName@(@ParameterName@</FirstParameterBodyFunction>


<ParametersBodyOperation>;
```

```
                                        @ParameterName@</ParametersBodyOperation>

<EndOperationWithParameters>);
       end @OperationName@;
</EndOperationWithParameters>

<OSActivationElementHead>
-- One shot activation controller
       protected body OneShotActivationController is
              procedure activate is
              begin
                     isActivate:=True;
              end activate;

              procedure passivate is
              begin
                     isActivate:=False;
              end passivate;

              procedure remove is
              begin
                     isRemoved:=True;
                     isActivate:=True;
              end remove;

              entry taskControl when (isActivate or isRemoved) is
              begin
                     if isRemoved then raise activationEnd;end if;
              end taskControl;
       end OneShotActivationController;

       task body OneShotActivationTask  is
       begin
              block.controller.taskControl;
              SchedulingControlService.bind(block.schedulingId);
              select
                     block.controller.taskControl;
              then abort
                     block.Run(block.state);
                     raise activationEnd;
              end select;
       exception
              when activationEnd
=>SchedulingControlService.unbind(block.schedulingId);
       end OneShotActivationTask;
</OSActivationElementHead>

<PActivationElementHead>
--  Periodic activation controller:
       protected body PeriodicActivationController is
              procedure activate is
              begin
                     isActivate:=True;
              end activate;

              procedure passivate is
              begin
                     isActivate:=False;
              end passivate;

              procedure remove is
              begin
                     isRemoved:=True;
                     isActivate:=True;
              end remove;

              entry taskControl when isActivate is
              begin
                     if isRemoved then raise activationEnd;end if;
              end taskControl;
       end PeriodicActivationController;

       task body PeriodicActivationTask  is
              nextUpdate:Time:=Clock;
       begin
              SchedulingControlService.bind(block.schedulingId);
              loop
                     block.controller.taskControl;
```

```
                        block.update(block.state);
                        delay until nextUpdate;
                        nextUpdate:= nextUpdate + block.period;
                end loop;
        exception
                when activationEnd
=>SchedulingControlService.unbind(block.schedulingId);
        end PeriodicActivationTask;
</PActivationElementHead>

<EndComponent>
end @ComponentName@;
</EndComponent>
```