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


# Herramientas para el despliegue de aplicaciones basadas en componentes.

**César Cuevas Cuesta**
cuevasce@unican.es


**Director:**
**José María Drake Moyano.**
**Grupo de Computadores y Tiempo Real**
**Departamento de Electrónica y Computadores.**

**Santander, Septiembre de 2008**
**Curso 2007/2008**

**Agradecimientos:**

- En primer lugar, agradecer a mi mentor José María Drake Moyano, por haberme guiado en la elaboración de esta tesis y sobre todo por haberme dado la oportunidad de pertenecer a CTR, para mi mucho más que un puesto laboral.

- En segundo lugar a mis compañeros del despacho-12, Ángela, Iñaki, Laura, quien diseñó las reglas de generación mediante las cuales pude desarrollar las herramientas que se presentan en este trabajo, y Álvaro, quién lo único que no saben él y su intuición es que lo saben todo, y en general a todos mis compañeros de CTR.

*A mis padres.*

**Siglas y acrónimos**

- CASE — ***C**omputer **A**ided **S**oftware **E**ngineering.*
- GUI — ***G**raphic **U**ser **I**nterface.*
- OMG — ***O**bject **M**anagement **G**roup*.
- CCM — ***C**ORBA **C**omponent **M**odel*.
- CCM — ***C**ontainer **C**omponent **M**odel.*
- CORBA — ***C**ommon **O**bject **R**equest **B**roker **A**rchitecture.*
- LwCCM — ***L**ight **w**eight* **CCM**.
- RT-EP — ***R**eal **T**ime **E**thernet **P**rotocol*.
- ICE — ***I**nternet **C**ommunication **E**ngine.*
- PIM — ***P**latform **I**ndependent **M**odel*.
- PSM — ***P**latform **S**pecific **M**odel.*
- BIM — ***B**usiness **I**nterface **M**anagement.*
- SLICE — ***S**pecification **L**anguage for **ICE.***
- IDL — ***I**nterface **D**escription **L**anguage.*
- UML — ***U**nified **M**odelling **L**anguage*.
- XML — ***E**xtensible **M**arkup **L**anguage.*
- IDE — ***I**ntegrated **D**evelopment **E**nvironment.*
- SWT — ***S**tandard **W**idget **T**oolkit*.
- AWT — ***A**bstract **W**indowing **T**oolkit*.
- WSAD — ***W**ebSphere **S**tudio **A**pplication **D**eveloper*.
- MVC — ***M**odel **V**iewer **C**ontroller.*
- MDA — ***M**odel **D**riven **A**rchitecture*.
- EMF — ***E**clipse **M**odelling **F**ramework.*
- MOF — ***M**eta-**O**bject Facility.*
- CWM — ***C**ommon **W**arehouse **M**etamodeling.*
- XMI — XML ***M**odel **I**nterchange*.
- W3C — ***W**orld **W**ide **W**eb **C**onsortium.*

**Índice:**

## Índice de figuras:

# 1. PROGRAMACIÓN GENERATIVA.

## *1.1 Programación generativa y automatización del desarrollo de las aplicaciones informáticas.*

El lugar tan relevante que los sistemas informáticos ocupan hoy en día en nuestra sociedad, contrasta con el bajo nivel de evolución que actualmente tiene la ingeniería software que se utiliza para desarrollarlos. De la ingeniería software se espera que sea capaz de gestionar tanto la complejidad que año a año se duplica como la productividad que se necesita para cubrir la expansión de los objetivos a los que se aplica y la calidad que se requiere para ser aplicada a sistemas que son críticos desde el punto de vista social o económico. Desafortunadamente, la ingeniería software que actualmente se usa no es capaz de satisfacer estas expectativas. Vista con la perspectiva de un siglo de experiencia de revolución industrial en otras ingenierías, la ingeniería software se encuentra en las primeras fases de su evolución. En ella domina la producción a medida, que hace "a mano" cada producto, frente a la producción en cadena que utilizan otras ingenierías más maduras y con la que consiguen garantizar niveles superiores de productividad y calidad. Como se muestra en la figura 1.1, que describe los grandes hitos en la evolución de la revolución industrial, ésta no ha sido una tarea sencilla sino que ha requeridos casi dos siglos para alcanzar su madurez.

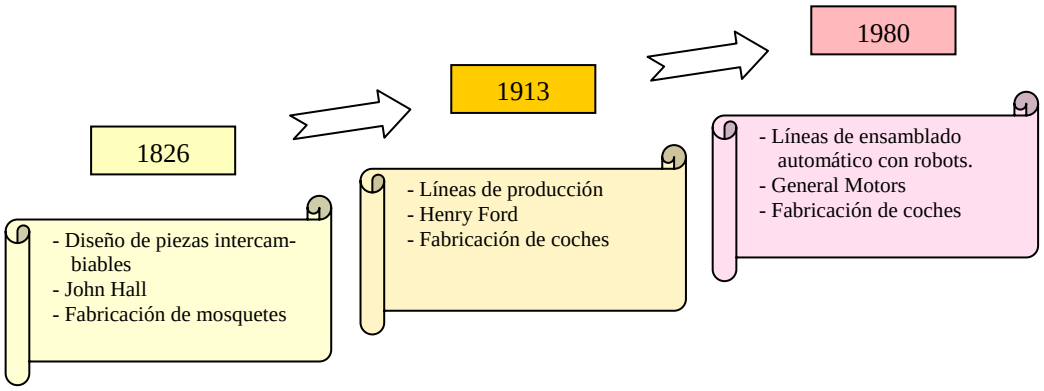



**Figura 1.1 - Principales hitos en la revolución industrial.**

En contraposición a esta evolución propia de las ingenierías mecánicas, eléctrica, electrónica, etc., en la figura 1.2 se muestra la evolución de la ingeniería software.

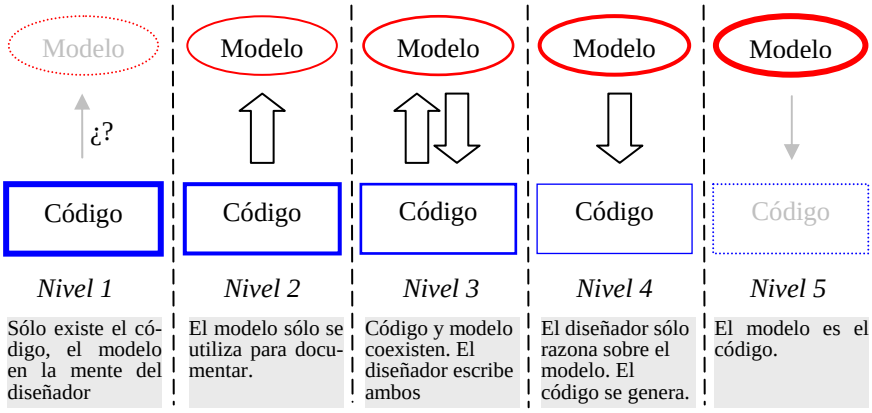



**Figura 1.2 - Fases en la automatización de la generación del código.**

Veamos a continuación una breve descripción de cada nivel:

1. Se utiliza únicamente el código, que, simultáneamente, es el medio de definir la funcionalidad del programa, de documentar su contenido y de transmitir los conceptos de diseño incluidos en él. Apoyado por el hecho de que requiere un único documento y sólo un editor de texto es aún hoy en día muy frecuentemente utilizado.
2. El diseñador diseña y escribe el programa utilizando el código como elemento principal. Únicamente utiliza el modelo como forma de comunicar a otros diseñadores humanos las ideas de diseño que se incluyen en él.
3. El diseñador utiliza de forma combinada el modelo y el código como medios de diseñar y codificar el programa. En las fases iniciales en que domina el diseño, utiliza el modelo y, haciendo uso de generadores de código, traduce el modelo a código. Posteriormente, en las fases de codificación y verificación trabaja directamente sobre el código, y con herramientas de ingeniería inversa mantiene el modelo sincronizado con el código que escribe. Es la metodología que corresponde a los actuales entornos **CASE** (***C**omputer **A**ided **S**oftware **E**ngineering)* de programación.
4. El diseñador sólo piensa sobre el modelo, el cual, a través de un generador automático de código, es codificado para su posterior interpretación de forma convencional por un computador. El código se genera porque lo requiere la máquina, pero no se requiere de programador que lo sepa interpretar. Es la metodología que utilizan los entornos basados en componentes, como los entornos de diseño de GUIs (***G**raphic **U**ser **I**nterface)*. Es el nivel al que se destina este trabajo.
5. Sólo existe el modelo, el cual representa el medio con el que el diseñador diseña el programa, y así mismo, es lo que interpreta el computador para implementar la funcionalidad que se le requiere. Actualmente sólo es utilizado para dominios de aplicación muy específicos.

La programación generativa (PG, en adelante) [1] constituye una metodología básica para ser aplicada en el nivel 4 y representa un paso más en la evolución de la ingeniería software en busca de la automatización de los procesos de desarrollo y ensamblado de los sistemas informáticos. Se basa en dos principios:

i. Se traslada el objetivo del diseño al desarrollo de familias de productos o dominios de aplicación compuestos por componentes estandarizados. Estos a su vez se utilizan para diseñar aplicaciones concretas que se construyen ensamblando los componentes ya disponibles.
ii. Se diseñan herramientas que automatizan la generación del código de los componentes y su ensamblado en sistemas informáticos.

Así pues, la PG es un paradigma de la ingeniería software basado en el modelado de familias de sistemas software tales que, dada la especificación de requerimientos en el diseño de un sistema concreto, pueda generarse automáticamente un producto intermedio o final perfectamente especializado y optimizado por ensamblado de componentes reutilizables más elementales y configurados de acuerdo con su uso en él, por medio de lo que se conoce como “conocimiento de configuración”. Por consiguiente, la PG se centra en familias de sistemas software en vez de en sistemas únicos. En lugar de construir miembros individuales de una familia desde cero, todos ellos pueden ser generados a partir de lo que se conoce como un **modelo generativo de dominio**, esto es, un modelo de una familia de sistemas compuesto por tres elementos:

i. **Dominio-problema**: metodología y formalismo empleados para especificar de forma precisa la funcionalidad de los miembros de la familia y las opciones de configuración que admiten.
ii. **Dominio-solución**: conjunto de implementaciones de los componentes disponibles a partir de los cuales cada miembro de la familia puede ser ensamblado y la descripción de los parámetros y mecanismos con los que se configuran cada una de sus opciones.

iii. El ya referido **conocimiento de configuración,** que representa la correspondencia entre la especificación de un miembro y el miembro finalizado, esto es, el conocimiento de las relaciones precisas entre las características de configuración especificadas para cada componente y la forma en que esas características se hacen efectivas en el producto. Visto desde el prisma de los dos primeros elementos, el conocimiento de configuración representa la forma en que se relacionan las características y opciones de configuración del dominio-problema con la selección de implementaciones y valores de los parámetros en el dominio-solución, así como las restricciones entre características y las optimizaciones que pueden conseguirse en el diseño de un sistema concreto.

En la figura 1.3 podemos ver una ilustración de estos tres elementos.

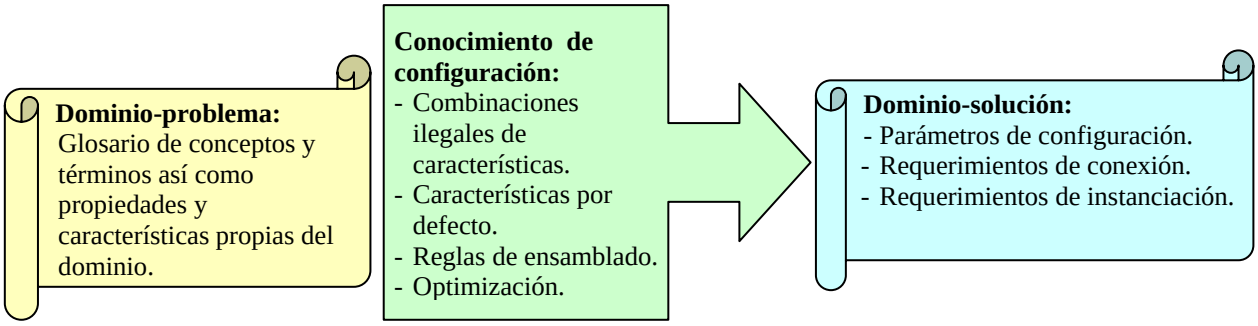



**Figura 1.3 - Elementos de un modelo generativo de dominio.**

El problema con la actual ingeniería software es que usualmente llegamos a un sistema software concreto pero sin saber cómo hemos llegado. La mayor parte del conocimiento de diseño se pierde y esto hace que el mantenimiento y evolución del software sea muy difícil y costosa. En PG queremos capturar el conocimiento de configuración en forma de programa, es más, aspiramos a capturar tanto conocimiento de producción en forma de programa como sea posible, el cual no sólo incluye el conocimiento de configuración, sino también otros aspectos como estrategias de testeo, diagnosis de errores, soporte para depuración, etc.

La PG engloba dos ciclos de desarrollo completos: uno para diseñar e implementar un modelo generativo de dominio (desarrollo para en el futuro poder aplicar reutilización), cuyo ámbito es una familia de sistemas y otro para emplear el modelo generativo en la producción de sistemas concretos (desarrollo aplicando reutilización), el cual ha de ser cuidadosamente diseñado para aprovechar los activos reutilizables en forma sistemática. El aspecto crucial del primer ciclo de desarrollo es la selección y delimitación de la familia de sistemas. Su primer paso es establecer la amplitud de su ámbito, esto es, decidir qué características deben incluirse y cuales no. Para ello hemos de analizar los objetivos del proyecto, los mercados actuales y potenciales, las posibilidades tecnológicas, etc. Una amplitud de ámbito extensa incrementa la reutilización del dominio en futuras aplicaciones, pero en contrapartida conlleva el desarrollo de elementos complejos que son difíciles de evolucionar y mantener. Conseguir un equilibrio en la amplitud del ámbito entre las necesidades actual y futura es un aspecto muy importante para el éxito de la tecnología en un determinado dominio. Un segundo paso consistiría en determinar las características comunes y variables de los miembros de la familia, así como las dependencias entre las características variables. Los resultados de este análisis proporcionan la base para establecer las categorías de implementaciones de componentes para una familia de sistemas y los medios de especificar los miembros de esa familia.

## *1.2 Programación generativa y tecnología de componentes.*

La tecnología de componentes software [2,3] constituye uno de los paradigmas de la ingeniería software con mayor futuro para incrementar la calidad del software, acortar los tiempos de desarrollo de los productos y gestionar el continuo incremento de su complejidad.

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

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

El diseño de las aplicaciones ensamblando componentes proporciona muchas ventajas:

i. La arquitectura que resulta es simple y está basada en interfaces y no en complejos protocolos de comunicación entre subsistemas.
ii. Permite escalar y reconfigurar el sistema que se diseña con sólo modificar el plan de despliegue y sin tener que modificar el código de bajo nivel.
iii. Simplifica la evolución y el versionado de los sistemas al requerir únicamente la sustitución de componentes y en su caso el desarrollo de nuevos componentes con funcionalidad y especificación bien definida.

Los componentes son siempre parte de un dominio de aplicación bien definido y pertenecen a una tecnología estandarizada que define la conectividad y los modelos de interacción entre ellos.

Existen diferentes tecnologías de componentes, cada una destinada a una determinada plataforma de ejecución. Por ejemplo COM, COM+, DCOM y .NET son tecnologías de componentes destinadas a la plataforma Windows de Microsoft, Java Beans y EJB para la plataforma Java, la tecnología CCM (***C**orba **C**omponent **M**odel)* de la organización OMG (***O**bject **M**anagement **G**roup*) requiere como plataforma el *middleware* CORBA (***C**ommon **O**bject **R**equest **B**roker **A**rchitecture*), etc.. La tecnología de componentes en que se encuadra esta tesis es la tecnología CCM (***C**ontainer **C**omponent **M**odel*) que se desarrolla en el grupo CTR (**C**omputadores y **T**iempo **R**eal) de la Universidad de Cantabria, tecnología que veremos en mayor detalle en la sección 1.4 y en el capítulo 2.

Una tecnología de componentes no tiene obligatoriamente que hacer uso de la PG. Cuando la plataforma de destino de una familia de componentes esta muy bien delimitada, los componentes pueden ser distribuidos con implementaciones de código binario directamente ejecutables y configurables durante su instanciación, asignando valores a los parámetros de configuración que tienen declarados. Sin embargo, cuando la plataforma es heterogénea y distribuida, se requiere que cada componente disponga de múltiples implementaciones y así, cuando se ensambla la aplicación, debe seleccionarse la instancia adecuada de acuerdo con la naturaleza del procesador en el que debe ejecutarse. Esta solución conlleva no sólo tener que incluir todas las posibles implementaciones, según el lenguaje de programación, sistema operativo o *middleware* de ejecución, sino que posiblemente, las implementaciones también tengan que ser actualizadas cuando la versión de alguno de los elementos de la plataforma se modifica.

La PG reduce el problema de la multiplicidad de las implementaciones. Como se muestra en la figura 1.4, el entorno de desarrollo puede almacenar cada componente como un paquete que contiene

diferentes modelos de especificación o comportamiento así como uno o un reducido número de elementos de código parametrizable, pero sin implementaciones ejecutables. En este caso, cuando en una aplicación se necesita una instancia del componente, una herramienta de generación de código analiza el plan de despliegue, y, en función de los requerimientos y de la naturaleza de la plataforma, genera el código ejecutable de la instancia del componente.

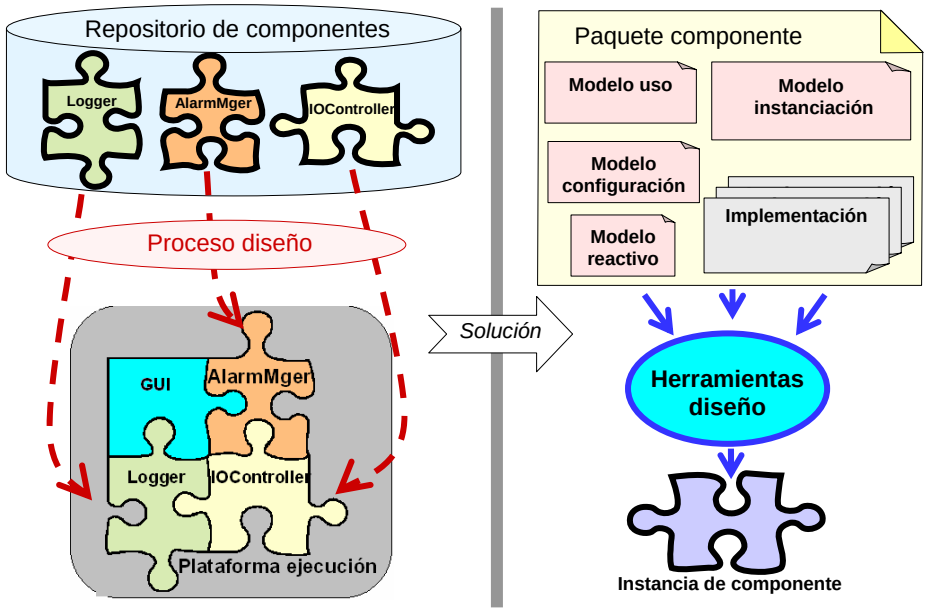



**Figura 1.4 - Utilización de la PG en las tecnologías de componentes.**

La asociación de la PG con las tecnologías de componentes reduce la complejidad de éstas y simplifica los procesos de mantenimiento y actualización de versiones. Por ello, las tecnologías de componentes que han aparecido en el último lustro incorporan el paradigma de PG formalizando el concepto de contenedor, que facilita la incorporación de código generado con herramientas automáticas. Como se muestra en la figura 1.5, la implementación de la instancia de un componente que se ensambla en una aplicación se compone de dos partes: el **código de negocio**, que es código generado por el diseñador del componente para implementar la funcionalidad de negocio propia del dominio de aplicación, y el **contenedor** (*wrapper)*, que representa la parte del código de la instancia que facilita la ejecución del componente en la plataforma de ejecución e implementa los mecanismos de conexión con otros componentes de acuerdo con los recursos del *middleware* disponible. El código del contenedor lo genera una herramienta automática en función de los modelos asociados al componente y del plan de despliegue que describe la plataforma en que se ejecuta y los recursos de comunicación con los que se establece la conexión con los otros componentes.

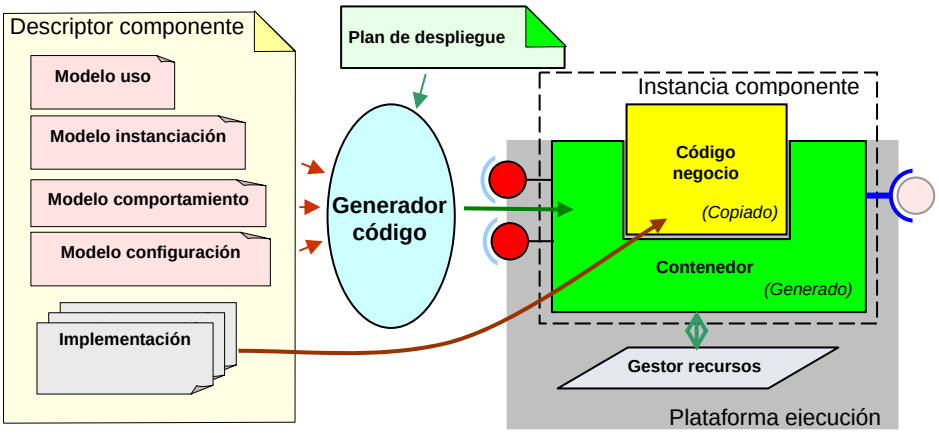



**Figura 1.5 - Tecnología de componentes para programación generativa.**

## 1.3 Programación generativa y modelos no funcionales.

Las tecnologías de componentes convencionales sólo consideran los aspectos relativos a los modelos funcionales de los componentes, por lo que describen los componentes a través de los servicios que ofrecen o los servicios que requieren para operar. En ambos casos esos servicios se describen mediante interfaces y la conectividad de los componentes se formula a través de la compatibilidad de las interfaces ofertadas por los componentes servidores con las interfaces requeridas por los componentes clientes. Así mismo, la instanciación de los componentes se formula a través de la compatibilidad entre los recursos que los componentes requieren para poder ser instalados, lo cual se describe en su modelo de instanciación y los recursos de que dispone la plataforma en la que el plan de despliegue indica que deben instalarse, que se describen en el modelo de la plataforma.

Sin embargo, en muchos casos esto no es suficiente ya que entre los requisitos de la aplicación que se diseña hay requisitos no funcionales (respuesta de tiempo real, uso de recursos, etc.) que también deben satisfacerse. Para estos casos se proponen [4, 5, 6] extensiones de la especificación de componentes que permitan describir los aspectos relacionados con su comportamiento no funcional y herramientas que analizan la adecuación de éstos a los requisitos de la aplicación en la que se integran.

Los modelos no funcionales de los componentes son básicamente estructuras de datos formuladas de acuerdo con una determinada especificación, que describen las características que son consecuencia del código del componente y que se necesitan conocer para analizar la adecuación del mismo en los aspectos no funcionales que se están considerando. Habitualmente estos modelos son simples conjuntos de datos que no son analizables de por sí para un determinado componente, sino que necesitan ser integrados con los de otros componentes en el contexto de una aplicación para que se genere un modelo completo de la aplicación, el cual sí es analizable y sus resultados pueden ser contrastados con los requisitos especificados.

Como se muestra en la figura 1.6, la gestión de los modelos no funcionales en una tecnología de componentes es similar a la PG, en la que en función de modelos parciales de los componentes se genera el modelo completo de la aplicación que proporciona la información no funcional requerida. No es propiamente PG, ya que su objetivo no es la generación del código, sino la generación de los modelos de comportamiento no funcional de las aplicaciones.

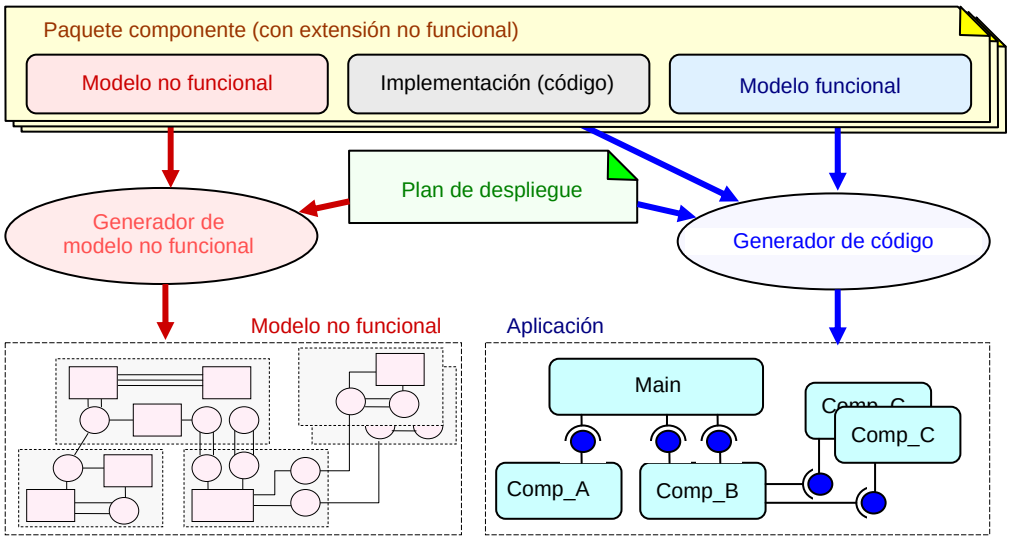



**Figura 1.6 - Generador de código y generador de modelo no funcional.**

## 1.4 La tecnología de componentes CCM, marco contextual de esta tesis.

Como hemos significado anteriormente, la tecnología de componentes en que se encuadra esta tesis es la tecnología CCM (***C**ontainer **C**omponent **M**odel*). Su objeto es desarrollar aplicaciones complejas basadas en componentes con capacidad de admitir requisitos de tiempo real estrictos y laxos y que puedan ser ejecutadas en plataformas embebidas con recursos limitados. La tecnología toma como punto de partida la especificación LwCCM (***L**ight **w**eight* **CCM**) [7], y en ella se hacen un conjunto limitado de modificaciones:

i. No se utiliza CORBA como soporte de comunicación entre componentes, sino que ésta se efectúa a través de conectores, que son un tipo especial de componente que engloba en su código los servicios de comunicación requeridos y que pueden basarse en diferentes mecanismos de comunicación. La conexión entre los componentes y el conector es siempre local.

ii. Se han introducido nuevos servicios y mecanismos en los contenedores que permiten controlar las estrategias de planificación de los *threads* que utilizan los componentes. El objetivo de estos servicios es garantizar la predecibilidad del comportamiento temporal de las aplicaciones.

iii. Para la descripción de las interfaces e implementaciones de los componentes, de las plataformas y de las aplicaciones se utilizan los formatos definidos en la especificación *Deployment and Configuration of Component-based distributed applications* Formal/06-04-02 (*D&C*, en adelante) [8] de OMG. Esta especificación también ha sido extendida [9] para que incorpore información introspectiva (*metadata*) del comportamiento temporal de los componentes y de las plataformas y que es utilizada para gestionar la planificabilidad de la aplicación.

Esta tecnología CCM que se está desarrollando en nuestro grupo CTR se enmarca en tres proyectos de investigación. En cada uno de ellos se abordan aspectos diferentes:

i. En el proyecto Thread [10, 11, 12] se aborda el tiempo real estricto, para lo cual la tecnología se adapta a una plataforma de ejecución de tiempo real, esto es, se utiliza lenguaje Ada y se emplean MaRTE OS como sistema operativo y RT-EP (***R**eal **T**ime **E**thernet **P**rotocol*) como protocolo de comunicación.

ii. En el proyecto HESPERIA [13,14] se adapta la tecnología a sistemas distribuidos de tiempo real laxo y se utiliza como plataforma el *middleware* ICE (***I**nternet **C**ommunication **E**ngine*) de la empresa ZeroC.

iii. En el proyecto FRESCOR [15] se adapta la tecnología para que opere en tiempo real estricto sobre plataformas abiertas haciendo uso de los contratos de servicio y los recursos virtuales que se desarrollan en el proyecto.

## 1.5 Objetivos de la tesis.

El objetivo de la tesis ha sido el desarrollo de un entorno integrado para la automatización de los procesos de generación de código y de generación de modelos no funcionales para la tecnología CCM. Dentro de este objetivo se ha buscado que el proceso de desarrollo de las herramientas sea sencillo y esté estandarizado.

En esta tesis proponemos un entorno de desarrollo en el que se organiza de forma estandarizada toda la información que requiere el proceso de desarrollo de componentes y definimos estrategias para desarrollar de forma sencilla las herramientas que se requieren en el mismo. Las actividades que hemos realizado son:

i. Exploración de la plataforma Eclipse [16] como base del entorno de desarrollo. Su utilización hace que el proyecto disponga de una forma estandarizada de interacción gráfica, y así mismo, le proporciona una infraestructura ya desarrollada rica en recursos y herramientas.

ii. Diseño del repositorio de la tecnología CCM, definiendo una arquitectura de ficheros estandarizada en la que cada documento tiene fijada su localización en función de su contenido y del papel que juega en la tecnología. Con la estandarización del repositorio se hace muy ágil la declaración de los parámetros de entradas y salidas de las herramientas.

iii. Definición de diferentes estrategias para desarrollo de herramientas:

   a. Herramientas implementadas mediante extensión de Eclipse, donde se hace uso de los puntos de extensión que ofrece Eclipse para la conexión con herramientas ya disponibles y otras desarrolladas a propósito.

   b. Herramientas diseñadas a partir de una meta-herramienta que permite construir código en función de un diccionario de patrones (segmentos parametrizados de código).

   c. Herramientas diseñadas de forma específica que se integran en el entorno a través de la estandarización de las interfaces de usuario y de la forma en que se invocan desde las cajas de herramientas que ofrece Eclipse.

iv. Validación del entorno y de las metodologías de desarrollo de herramientas, diseñando e implementando un conjunto de las herramientas que se requieren en los diferentes proyectos que actualmente están realizándose.

# 2. TECNOLOGÍA DE COMPONENTES CCM Y GENERACIÓN AUTOMÁTICA DE CÓDIGO.

## *2.1 Introducción.*

En este trabajo se aborda la aplicación de la PG al desarrollo de componentes y aplicaciones utilizando la tecnología CCM. En este capítulo se analiza esta tecnología desde el punto de vista del entorno de desarrollo, esto es:

- De los procesos de especificación, análisis, diseño y gestión que se utilizan.
- De la información que se maneja.
- De los actores que intervienen en el proceso.
- De las herramientas que se necesitan para automatizar los procesos de generación de código.

En general, el proceso de desarrollo de aplicaciones basadas en componentes se compone de dos fases independientes pero complementarias.

i. Desarrollo de los componentes: consiste en especificar, diseñar, empaquetar y distribuir los componentes como unidades independientes, reutilizables y concebidos en función de un dominio de aplicación, pero no destinados a una aplicación específica. Veremos esta fase del proceso en mayor detalle en la sección 2.2.

ii. Desarrollo de las aplicaciones: el desarrollo de una aplicación concreta se lleva a cabo para dar solución a un problema que se tiene especificado y planteado. En primer lugar se diseña la aplicación como un conjunto de instancias de los componentes registrados en el entorno, configurados e interconectados de forma adecuada para implementar la funcionalidad deseada en la aplicación. En segundo lugar, se identifica la plataforma en la que se va a ejecutar la aplicación, y, de acuerdo con la composición de ésta última, se decide su despliegue en la plataforma, se transfiere el código que proceda al nudo en que se va a ejecutar, y por último se lanza la ejecución coordinada de la misma. Analizaremos esta fase en mayor detalle en la sección 2.3.

Una tecnología de componentes se propone para que sea implementada por diferentes organizaciones y en el proceso de desarrollo deben operar herramientas creadas por diferentes entidades y posiblemente compartidas con otras tecnologías que tienen procesos comunes. A fin de tener un modelo de referencia común y garantizar la interoperatividad de las herramientas, la organización OMG ha definido la especificación *D&C*. Ésta es la especificación que se utiliza en el marco de la tecnología CCM. Estudiaremos este modelo de referencia en la sección 2.4.

Por último, en la sección 2.5 procederemos a una enumeración y breve descripción de las principales herramientas que se requieren en el proceso de desarrollo de los componentes y de las aplicaciones.

## 2.2 Proceso de desarrollo de un componente CCM.

En la figura 2.1 se muestran los tres niveles de abstracción en los que se desglosa la descripción de cualquier componente. A fin de simplificar, pondremos como ejemplo el componente *SoundGenerator*.

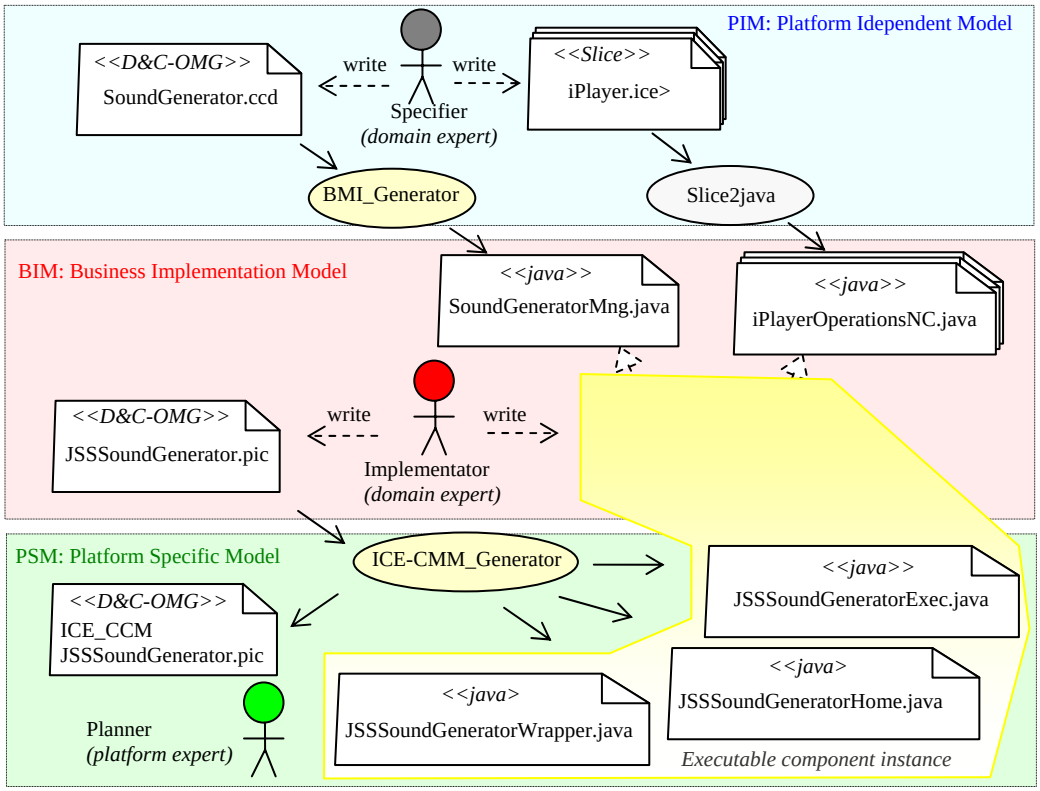



**Figura 2.1 - Niveles de abstracción de un componente.**

i. Nivel PIM (*Platform Independent Model*): Describe el comportamiento del componente con independencia de su implementación o del *middleware* con el que se accede a él. Es el modelo que utiliza el desarrollador de las aplicaciones al integrarlo en sus diseños para decidir la idoneidad de su funcionalidad, sus posibilidades de configuración y cómo debe ser utilizado. Su descripción se realiza mediante el fichero que describe la interfaz del componente (p.e. `SoundGenerator.ccd.xml`), siendo tanto su contenido como su formato conformes a la especificación *D&C*. A partir de él se tiene acceso a los ficheros que definen la funcionalidad de las interfaces y que también son parte de este modelo (p.e. `iPlayer.ice` y `iLogger.ice`). Como en ICE-CCM utilizamos por defecto el *middleware* ICE, resulta más fácil como lenguaje de especificación el lenguaje SLICE (***S**pecification **L**anguage for **ICE***), aunque en otras versiones CCM se utiliza IDL (***I**nterface **D**escription **L**anguage*). La información típica que corresponde a este nivel se muestra en la figura 2.2. Toda ella es información externa del componente: declaración de puertos (facetas y receptáculos), incluyendo la descripción de las interfaces que implementan, declaración de las interfaces de gestión del ciclo de vida del componente, declaración de los *thread* externos que requiere y declaración de los atributos de configuración.

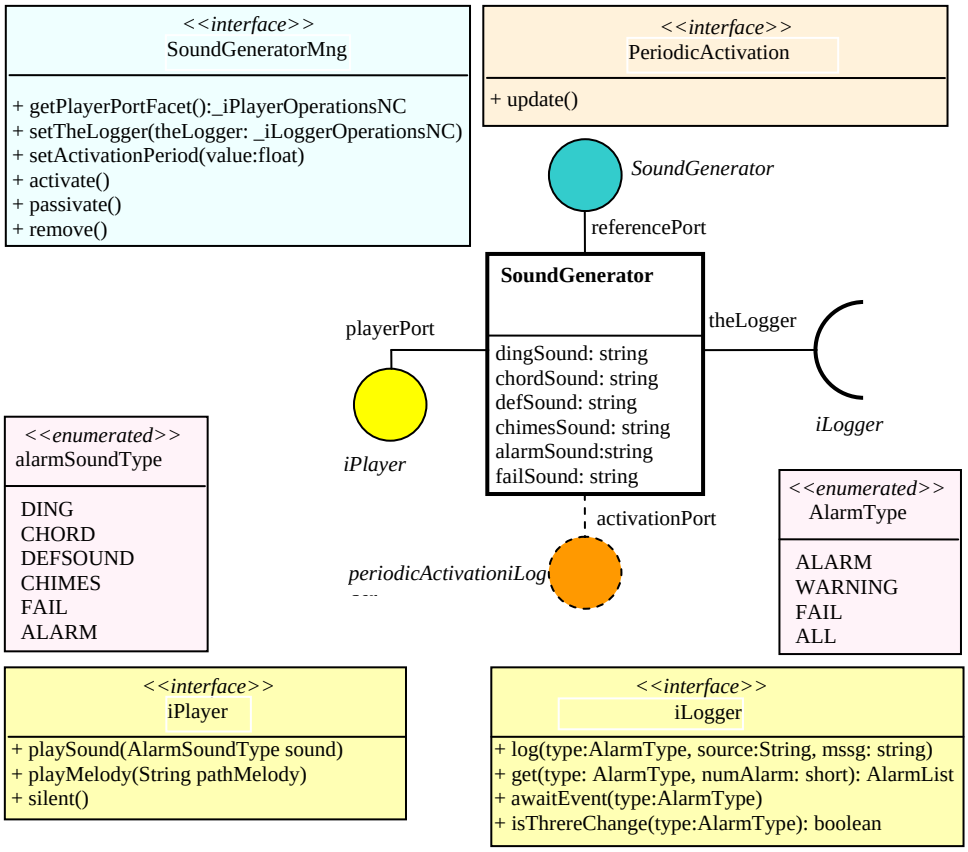



**Figura 2.2 - Ejemplo de la información que se maneja en el nivel PIM.**

ii. Nivel BIM (***B**usiness **I**mplementación **M**odel*): Describe una implementación concreta del componente en la que se han definido los recursos en que basa su funcionalidad. Es el modelo que se introduce para que el diseñador que desarrolla el código de negocio que implementa el componente pueda trabajar de forma libre y sin necesidad de conocer la tecnología de distribución subyacente. Un mismo componente puede poseer múltiples implementaciones con diferentes requerimientos en cuanto a recursos de la plataforma, pero todas ellas pueden ser adaptadas a la tecnología de una plataforma específica utilizando herramientas automáticas. Su descripción se realiza mediante el fichero de implementación (p.e. `JSSSoundGenerator.pid.xml`), cuyo formato y contenido son conformes a la especificación *D&C*. A partir de él se accede a los ficheros que contienen la interfaz que define la funcionalidad de gestión requerida por la tecnología del componente de negocio (p.e. `SoundGeneratorMng.java`) y a los ficheros que definen la funcionalidad que ofrecen las facetas del componente y la que requiere encontrar por sus receptáculos (p.e. `iPlayerOperationsNC.java` e `iLoggerOperationsNC.java`).

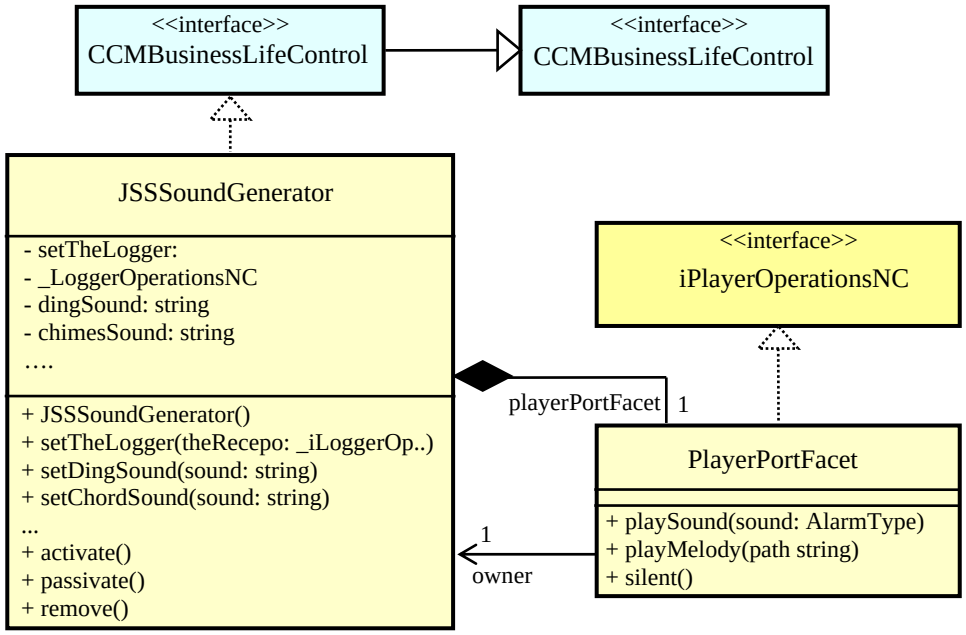



**Figura 2.3 - Ejemplo de la información que se maneja en el nivel BIM.**

iii. Nivel PSM (***P**latform **S**pecific **M**odel*): Describe una implementación **completa** del componente, dispuesta para ser instanciada y ejecutada en la plataforma. Integra en ella el código de negocio que aporta la funcionalidad del componente y los ficheros de código, generados automáticamente por herramientas, que proporcionan los recursos para poder operar sobre una plataforma determinada, ICE-CCM en este caso. Se describe mediante el fichero de descripción de la implementación, cuyo formato y contenido es conforme a la especificación *D&C* (p.e. `ICE-CCMJSSSoundGenerator.pid.xml`). Desde él se referencian los ficheros `*.java` que constituyen el código ejecutable del componente (`JSSSoundGeneratorWrapper.java`, `JSSSoundGeneratorExecutor.java` y `JSSSoundGeneratorHome.java`). En el diagrama de clases de la figura 2.4 se muestra la información que se maneja en el nivel PSM. Es básicamente un programa Java compuesto por la clase `JSSSoundGenerator` importada del nivel BIM y que se maneja sin modificar más que el conjunto de clases Java que constituyen el contenedor y cuyo código se ha generado mediante una herramienta automática.

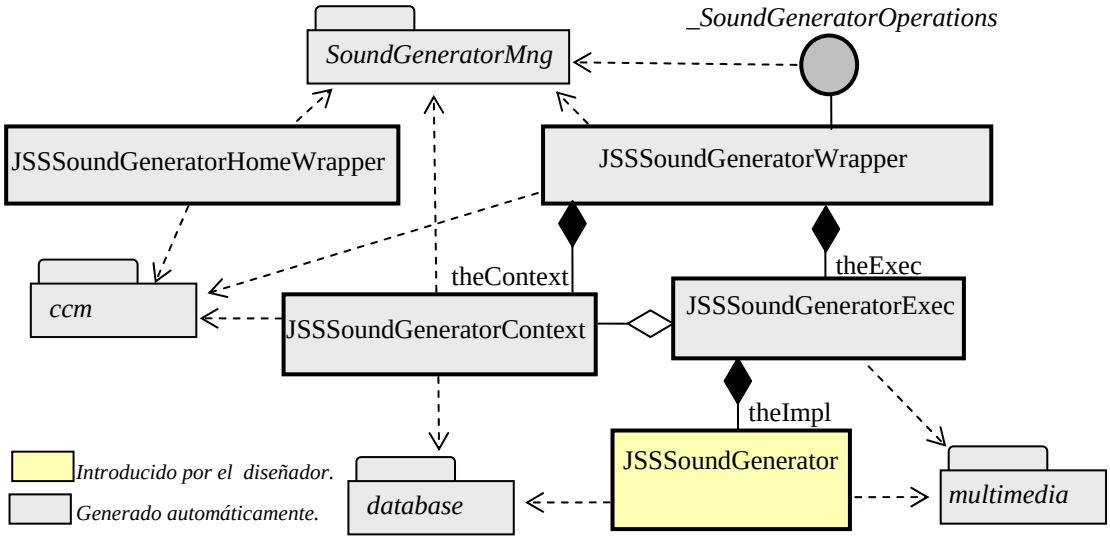



**Figura 2.4 - Ejemplo de la información que se maneja en el nivel PSM.**

## 2.3 Desarrollo de una aplicación basada en componentes.

En la figura 2.5 se muestran las fases junto con los actores, las herramientas y los productos que intervienen en el proceso de ensamblado, despliegue y ejecución de una aplicación.

El ensamblador (*assembler*) construye la aplicación como un conjunto de instancias de componentes interconectadas entre sí de forma que se implemente la funcionalidad requerida en su especificación. Estas instancias han de corresponder a componentes instalados en el entorno de desarrollo y las interconexiones deben satisfacer los requisitos establecidos en el modelo de cada componente. El ensamblador toma las decisiones basándose únicamente en la especificación de los componentes, sin necesidad de elegir en esta fase la implementación concreta del componente que se utiliza. La descripción de la aplicación se formula como la de un componente más (es una aplicación porque es útil de por si), en este caso como un componente compuesto, a través de un fichero `.cad.xml` *(Component Assembly Description*) acorde a la especificación *D&C*. Si la aplicación tiene requerimientos no funcionales (por ejemplo, tiempo real) también tiene que modelar **la carga de trabajo** (*workload*) que representa, esto es, declarar las transacciones que se ejecutan en ella, la frecuencia con que lo hacen y los requisitos temporales que se requieren.

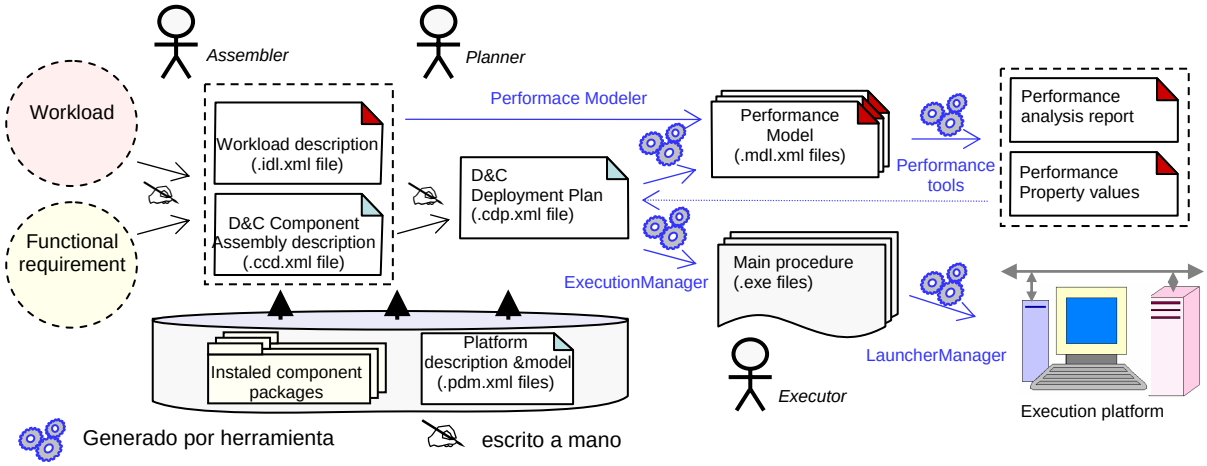



**Figura 2.5 - Proceso de desarrollo de una aplicación basada en componentes.**

La siguiente fase es el despliegue. En ella, el planificador (*planner*) parte de la descripción de la aplicación y construye el plan de despliegue (*deployment plan*), en el cual se asigna cada instancia de componente al nudo procesador en que se va a ejecutar, se formulan los valores de las propiedades de configuración de cada una de esas instancias y se asigna el mecanismo de comunicación a utilizar en la interacción entre cada dos instancias. El plan de despliegue se formula mediante un fichero `*.cdl.xml` definido en el estándar *D&C* y constituye de hecho la descripción de la aplicación. En el plan de despliegue se determina implícitamente (o explícitamente cuando se necesite) la implementación específica del componente que se utiliza en cada instancia de la aplicación. Esta selección se realiza en función del procesador en que se ejecuta, de los otros componentes a los que se conecta y de los valores que se asignan a los parámetros de configuración.

Si la aplicación tiene requisitos de tiempo real, se debe construir el **modelo de comportamiento** de la aplicación (a partir de la carga de trabajo y de los modelos de comportamiento de los componentes) sobre la plataforma de desarrollo. Este modelo de comportamiento debe ser evaluado y los resultados pueden ser utilizados para refinar los valores que se asignan a las propiedades de configuración de las instancias. En la última etapa del proceso, el ejecutor hace uso de las herramientas de ejecución y lanza la ejecución de la aplicación sobre la plataforma distribuida.

## 2.4 Especificación D&C sobre el proceso, la información y las herramientas de componentes.

Los objetivos del estándar *D&C* son unificar las diferentes especificaciones propuestas por los diferentes promotores de tecnologías y plataformas a fin de hacer ínteroperables las herramientas que constituyen los entornos de desarrollo de las aplicaciones distribuidas basadas en componentes y formalizar los procesos y la información que se gestiona en el despliegue de dichas aplicaciones. El despliegue es entendido en esta especificación como el conjunto de tareas a desarrollar entre la adquisición del software elaborado y la ejecución del mismo sobre una plataforma. Esto requiere especificación para las tareas en lo referente a:

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

Hay dos razones por las que se ha seleccionado la especificación *D&C*:

i. Es neutra respecto a la tecnología de componentes y por ello es más fácil aplicarla a una nueva tecnología que se desarrolla.
ii. Está formulada mediante un metamodelo UML (***U**nified **M**odeling **L**anguage*) que facilita su extensión para incorporar nuevos aspectos de diseño, como la gestión de recursos o el comportamiento temporal.

La adaptación del estándar *D&C* a una tecnología determinada se realiza a través de un conjunto de plantillas W3C-*schema* (*schemas*, en adelante) que definen el contenido y el formato de los documentos XML (***E**xtensible **M**arkup **L**anguage*) que describen las interfaces, las implementaciones, los paquetes de los componentes, las plataformas distribuidas y la formulación del plan de despliegue que definen las aplicaciones basadas en componentes. En la figura 2.6, se muestran las cinco plantillas que cubren el estándar y las dependencias entre ellas.

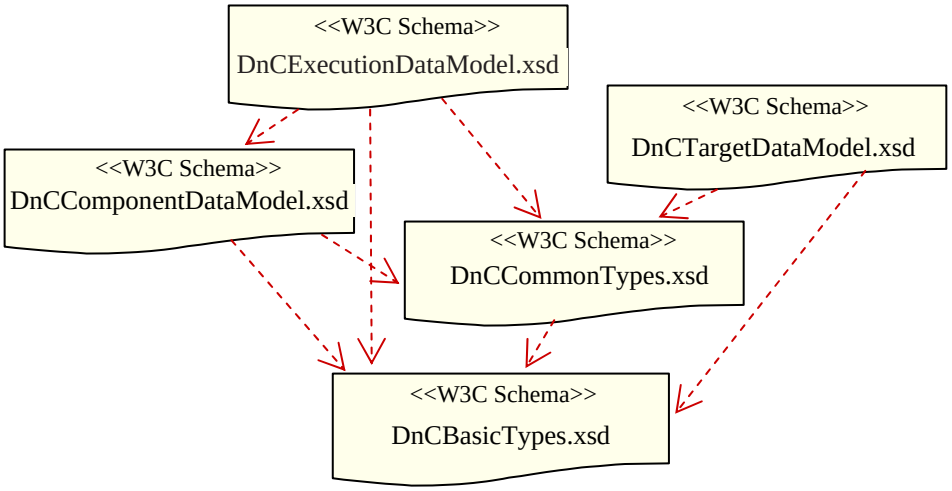



**Figura 2.6 - Plantillas W3C- *schema* de la especificación *D&C*.**

Las clases que se definen en la especificación *D&C* para describir un componente se recogen en el *schema* `DnCComponentDataModel.xsd` y son:

- La clase `PackageConfiguration` describe un paquete de componente como elemento distribuible. Cada paquete contiene una única descripción de la interfaz de un componente y una o múltiples implementaciones del mismo.
- La clase `ComponentInterfaceDescription` describe la citada interfaz del componente, la cual a su vez describe su funcionalidad desde un punto de vista externo. Dos componentes que ofrezcan una misma interfaz, son mutuamente sustituibles entre sí en cualquier aplicación. La interfaz describe las operaciones, atributos, puertos y parámetros de configuración, que, en conjunto, proporcionan al diseñador la información necesaria para determinar su funcionalidad y la forma de uso.
- La implementación de un componente puede ser monolítica, esto es, constituida por un conjunto de módulos de código (a*rtifact*) y por tanto independiente de otros componentes o puede estar recursivamente implementada como una agrupación plana de instancias de subcomponentes interconectados con una determinada topología, en cuyo caso la descripción del componente se formula con referencia a la descripción de sus subcomponentes. Las implementaciones monolíticas se describen mediante una clase `MonolithicImplementationDescription` y las implementaciones compuestas se describen mediante una clase `ComponentAssemblyDescription`.

El modelo de datos de una plataforma contiene la información para desplegar sobre ella cualquier aplicación y está compuesto por un conjunto de nudos interconectados por redes de comunicación, pudiendo haber puentes entre las redes. Las clases que se definen en la especificación *D&C* para describir una plataforma se recogen en el *schema* `DnCPlatformDataModel.xsd` y son:

- `Domain`: Describe la plataforma de ejecución como contenedor de más alto nivel. Un dominio está compuesto de uno o más nudos procesadores, redes de comunicación y puentes entre ellas. Tanto nudos como redes y puentes pueden tener asignados recursos, los cuales cualifican y cuantifican sus capacidades. Así mismo, los recursos compartidos por uno o más nudos también se consideran agregados directamente al dominio.
- `Node`: Los nodos tienen capacidad de procesamiento y en ellos se ejecutan los componentes. Están interconectados mediante canales de comunicación a través de los cuales intercambian información y acceden a servicios.

- `Interconnect`: Describe los canales de comunicación entre nudos.
- `Bridge`: Representa los *switches* o *routers* que interconectan los canales de comunicación y permiten complejas capacidades de comunicación entre nudos.
- `Resource`: Los recursos pueden estar asociados a los nudos, a los canales de comunicación o a los puentes y representan elementos que están agregados a ellos y que proporcionan las capacidades que pueden ser requeridas por los componentes para poder instalarse en ellos.
- `SharedResource`: Representa recursos que por su naturaleza son compartidos por diferentes nudos de la plataforma. Al no estar directamente agregados a ningún elemento se consideran agregados al dominio.

Una aplicación basada en componentes se describe estableciendo las instancias de los componentes que la constituyen, asignando a cada una de ellas el nudo de la plataforma en que debe instanciarse y valores a sus propiedades de configuración, así como definiendo y calificando las conexiones entre los componentes. Toda esta información se reúne en un documento que denominamos **plan de despliegue**. Las clases que se definen en la especificación *D&C* para describir un plan de despliegue se recogen en el *schema* `DnCExecutionDataModel.xsd` y son:

- `Deployment Plan`: representa el despliegue de una aplicación sobre una determinada plataforma. Contiene la información relativa a los ficheros de código que son parte del despliegue (`ArtifactDeploymentDescription`), la forma de crear las instancias de los componentes (`MonolithicDeploymentDescription`), y donde se han de instanciar (`InstanteDeploymentDescription`). Contiene también información relativa a la conexión entre los subcomponentes (`AssemblyConnectionDescription`) y a la correspondencia entre los puertos externos y los de los subcomponentes. Por último, contiene la descripción de la interfaz que es implementada por la aplicación que se despliega (`ComponentInterfaceDescriptor`), esto es, la descripción de su comportamiento visto desde fuera.
- `ImplementationDependency`: formula una dependencia de la implementación de un componente respecto del entorno de la plataforma e indica qué otras instancias de componentes o servicios deben estar instaladas en la plataforma antes de que la implementación sea desplegada.
- `PlanPropertyMapping`: identifica la correspondencia entre una propiedad o puerto de la aplicación que se despliega y la propiedad o puerto del subcomponente en el que delega.
- `ArtifactDeploymentDescription`: Describe un fichero relativo a la información de un componente que debe ser desplegado como parte del plan de despliegue de la aplicación. Contiene el localizador (*URL*) del fichero y los parámetros y requisitos de despliegue de cada componente, lo que hace autocontenido el plan de despliegue (`ImplementationArtifactDescription`).
- `MonolithicDeploymentDescription`: Describe el despliegue de un componente como parte de un plan. Referencia la descripción de los elementos que son parte del despliegue (`ComponentImplementacionDescription`)
- `PlanConnectionDescription`: Describe una conexión que se establece entre puertos de los componentes que constituyen la aplicación.
- `InstanceDeploymentDescription`: Contiene la información que es necesaria para desplegar una instancia de componente simple. Hace referencia a una descripción de un componente monolítico (`MonolithicDeploymentDescription`) e incluye el nombre de los nodos en los que es instanciado el componente. Además, contiene propiedades que son usadas para configurar la instancia del componente.

## 2.5 Especificación de herramientas.

El diseño y ejecución de una aplicación basada en componentes son procesos que se realizan en el entorno de desarrollo y son asistidos por herramientas que garantizan la validez "por construcción" de los artefactos que se generan. Como veremos en el tercer capítulo, el entorno de desarrollo de la tecnología CCM se ha basado en Eclipse [13], el cual proporciona un conjunto de *frameworks* y servicios que simplifican la gestión de los recursos y el desarrollo de las herramientas.

Existen dos elementos básicos que constituyen el entorno de desarrollo: el **repositorio** y las **herramientas**. El repositorio es una base de datos que está construida sobre el *workspace* de Eclipse y en ella se almacenan los productos que se introducen y los productos intermedios y finales que generan las herramientas. En la sección 3.7 entraremos a describir en detalle la estructura y características del repositorio que se ha diseñado. En cuanto a las herramientas que requiere el entorno son múltiples y de muy diferente naturaleza. En la siguiente lista se enumeran, ordenadas de acuerdo con la fase del proceso de desarrollo en que se utilizan, aquellas que se han concebido, algunas ya desarrolladas (aparecen subrayadas y se presentarán en mayor profundidad en la sección 4.5.1) y otras pendientes de serlo. Veámoslas:

- **Herramientas de gestión del repositorio:** tienen el objetivo de construir, mantener la coherencia y tanto introducir como extraer información de las diferentes secciones del repositorio.
  - **Inicializador del repositorio:** Reestructura el *workspace* de Eclipse en el que se invoca como un nuevo repositorio con la estructura y la información común apropiadas a la tecnología de componentes correspondiente.
  - **Importador de interfaces:** Introduce en el repositorio la información asociada a la interfaz que se importa, la cual viene definida por un fichero formulado en un lenguaje de especificación de interfaces, como IDL o SLICE. Comprueba que existe la información referenciada, necesaria para que pueda ser utilizada en el diseño de un componente.
  - **Exportador de interfaces:** Retorna el fichero con la información que se necesita para transferir la interfaz a otro entorno. Una interfaz se exporta como un fichero `*.zip` que empaqueta su descripción formal (IDL o SLICE) así como la descripción de las interfaces que referencia, necesarias para su uso en el diseño de un componente.
  - **Importador de plataforma:** Introduce en el repositorio la información asociada a la descripción de una determinada plataforma de ejecución, información en la que se incluyen los ficheros *D&C* que describen la estructura y los recursos de la plataforma y, opcionalmente, los ficheros que describen su comportamiento de tiempo real. La estructura y la interfaz de una plataforma se describen mediante un fichero `*.tdm` soportado por el *schema* `DnCTargetDataModel.xsd` y el fichero que describe su modelo de tiempo real es de extensión `*.rtp` y respaldado por el *schema* `rtmContainers.xsd`. Para la herramienta, la descripción de la plataforma viene dada en archivos con extensión `*.zip`.
  - **Exportador de plataforma**: Retorna el archivo `*.zip` con la descripción completa de la plataforma requerida para ser incorporada a otro entorno de desarrollo.
  - **Traductores entre IDL y XML y entre SLICE y XML:** Transforman el formato de ficheros de descripción de interfaces entre los formatos estandarizados `*.idl` o `*.ice` y los formatos `*.idl.xml` o `*.ice.xml` que utilizan las herramientas.
  - **Compiladores de SLICE a Java y de SLICE a C++**: Invocan las herramientas proporcionadas por el entorno ICE (herramientas externas) para generar los ficheros que deben enlazarse con las aplicaciones de forma que tengan acceso al *middleware*.
  - **Compilador de IDL a Ada:** Genera el código Ada que corresponde a una interfaz cuya funcionalidad se ha descrito en un fichero `*.idl`.

- **Herramientas de diseño de los componentes:**
  - **Importador de componente:** Introduce en el repositorio partes de la información relativa a un componente, las cuales vienen definidas por diferentes ficheros que describen su especificación, sus interfaces, sus implementaciones, su código y los correspondientes modelos no funcionales, todo ello empaquetado en un archivo `*.zip`. Además, comprueba que la información referenciada está disponible en el repositorio.
  - **Exportador de componente:** Retorna el archivo con la descripción completa existente en el repositorio del componente que se referencia.
  - **Generador de la interfaz de gestión de un componente:** Procesa la información referente a la especificación de un componente y, de acuerdo con ella, genera el código de las interfaces que deben ser implementadas por el código de negocio del componente.
  - **Generadores del contenedor de un componente:** Conjunto de herramientas parciales que generan el código del contenedor que adapta el código de negocio de un componente a la plataforma, de acuerdo con la descripción del componente y la implementación que se elija.
    - Generador de la clase con la que se implementa el caso de receptáculo múltiple (receptáculo con múltiples conexiones).
    - Generador de la clase ejecutor (*executor*) de un componente.
    - Generador de la clase contexto (*context*) de un componente.
    - Generador del contenedor propiamente dicho (*wrapper*) del componente.
    - Generador del código del constructor (*home)* del componente.
  - **Empaquetador de un componente**: Genera el paquete de información (*metadata*, código, modelos, etc.) que constituye al componente como elemento distribuible. En él se incluye toda la información necesaria para que el componente pueda ser incorporado al entorno de desarrollo de aplicaciones.
- **Herramientas de desarrollo de las aplicaciones:**
  - **Instalador de un componente**: Instala en el entorno de desarrollo de aplicaciones el paquete con el que se distribuye un componente, verifica que están instalados en el entorno los elementos referenciados en él y genera los elementos derivados de él que se requieren para su utilización dentro de dicho entorno.
  - **Importador de aplicaciones:** Introduce en el repositorio la información asociada a la aplicación que se importa, en forma de conjunto de ficheros que describen las diferentes partes de la aplicación (implementaciones, códigos y modelos no funcionales), todo ello empaquetado en un archivo `*.zip`.
  - **Exportador de aplicaciones:** Crea y retorna un archivo `*.zip` que contiene la información de la aplicación disponible en el entorno para que pueda ser transferida a otro entorno.
  - **Generador de aplicaciones**: Genera la información que se necesita para ejecutar una aplicación. Tiene como entrada el plan de despliegue de la aplicación y todos los elementos del repositorio referenciados en él. Genera como salida el conjunto de particiones de código dispuestas para ser ejecutadas en los nudos de la plataforma de ejecución.
- **Herramientas de gestión de los modelos no funcionales:**
  - **Generador de modelos:** Genera el modelo de comportamiento no funcional de una aplicación por composición de los modelos de comportamiento de los componentes que la conforman. Utiliza como entrada el plan de despliegue que describe la aplicación y todos los elementos del repositorio referenciados en él y genera como salida el modelo no funcional, formulado en un formato compatible con el entorno de análisis no funcional que se utilice.

- **Configurador del plan de despliegue:** Incorpora al plan de despliegue los parámetros de configuración generados en el análisis no funcional de la aplicación.

- **Herramientas de despliegue y lanzamiento:**
  - **Lanzador de aplicaciones:** Transfiere a los nudos de la plataforma de ejecución las particiones de código que constituyen la aplicación y ordena su ejecución.

El entorno Eclipse está muy bien dotado de editores especializados y por ello no se han desarrollado herramientas específicas para que el operador introduzca el código fuente o los ficheros XML de la especificación *D&C*. En el futuro desarrollaremos interfaces de usuario que asistan en la elaboración de los ficheros de descripción y modelado. Actualmente al disponer de las plantillas W3C-*Schema* que definen el formato y el contenido de estos documentos resulta fácil la introducción de esta información utilizando editores inteligentes que se guían por ellos.

# 3. EL ENTORNO ECLIPSE.

## *3.1 Introducción a Eclipse: historia y breve descripción.*

La primera versión de Eclipse apareció en Noviembre de 2001, anunciada por IBM como una donación de 40.000.000$ a la comunidad de Código Abierto. Desde entonces, Eclipse se ha apoderado del mundo Java (y no sólo del mundo Java) a pesar del hecho de que Sun Microsystems todavía no se ha involucrado. En la actualidad Eclipse está completamente gestionado por ***eclipse.org***, una organización independiente sin ánimo de lucro en la que, a pesar de todo, IBM juega un papel fundamental. Junto a ella, participan más de 150 compañías, como Ericsson, HP, Intel, etc. Pero no Microsoft.

Al tratar de describir Eclipse podríamos contestar que una GUI para aplicaciones Java o un IDE (***I****ntegrated* ***D****evelopment* ***E****nvironment*) de ese lenguaje, pero, según *eclipse.org*, Eclipse es una plataforma "para todo y para nada en particular". Que se pueda utilizar Eclipse para desarrollar programas Java (desde luego que es uno de los más completos IDEs para Java) es únicamente una aplicación más de esta plataforma. Realmente, gracias a su arquitectura modular, Eclipse es altamente adaptable a muchos paradigmas de trabajo. De hecho, el IDE de Java no es más que un ejemplo de complemento (*plug-in*, en adelante) para Eclipse y un gran número de ellos han sido desarrollados por numerosas compañías y desarrolladores, como por ejemplo *plug-in*s para UML, para C++, etc.

Eclipse es más que un entorno de desarrollo. Con sus bibliotecas gráficas SWT (***S****tandard* ***W****idget* ***T****oolkit*) y JFace proporciona una alternativa a las bibliotecas AWT (***A****bstract* ***W****indow* ***T****oolkit*) y Swing de Sun, permitiendo la creación de aplicaciones Java que se aproximan más a las aplicaciones nativas tanto en los formatos de apariencia (*look&feel*) como en el grado de respuesta.

Finalmente, Eclipse proporciona un amplio marco de referencia para implementar aplicaciones Java. Además de las bibliotecas SWT y JFace encontramos componentes de nivel superior, como editores, vistas, gestores de recursos, de tareas y de problemas, un sistema de ayuda y varios asistentes. Eclipse utiliza todos ellos para implementar elementos como el IDE de Java o el área de trabajo (*workbench*, en adelante), pero también pueden ser usados en nuestras propias aplicaciones pues el modelo de licencia de Eclipse permite a los usuarios embeber estos componentes en sus propias aplicaciones, modificarlos y distribuirlos como parte de ellas.

Básicamente, Eclipse no es más que la base de la infraestructura de la Comunidad del WSAD (***W****ebSphere* ***S****tudio* ***A****pplication* ***D****eveloper*) que ha sido promovida por IBM para promocionar la generación de productos de software libre. El núcleo es el mismo, siendo la principal diferencia que Eclipse, en su edición 3.0, consta de unos 90 *plug-ins* mientras que WSAD de unos 600.

## *3.2 El workbench de Eclipse.*

Los distintos componentes del *workbench* de Eclipse son los editores, las vistas y las perspectivas. Una perspectiva es una combinación y disposición de ventanas y herramientas orientadas a tareas concretas, de forma que una aplicación puede definir la disposición (*layout*) inicial de una página especificando una perspectiva. La plataforma Eclipse proporciona algunas predefinidas como la de Java, la de desarrollo de *plug-ins* o la de depuración y, naturalmente, las aplicaciones son libres de definir sus propias perspectivas. Así pues, un *plug-in* puede, sin ser obligatorio, añadir una o varias perspectivas al *workbench*. Además de los editores, cuya descripción creemos innecesaria, las vistas son el otro elemento fundamental de la ventana de trabajo de Eclipse. Cada perspectiva muestra inicialmente unas vistas determinadas dispuestas de forma concreta. Ejemplos de vistas son el `Explorador de Paquetes` y el `Navegador de Recursos`, a las que nos referiremos en la sección 3.4.4, así como las vistas de Problemas, de Tareas, etc. Una de las vistas más útiles para el desarrollo de programas es la vista `Outline`. Esta vista soporta navegación por dentro de un fichero de código

fuente. Para Java, la vista muestra entradas para campos y métodos así como sentencias *import* y clases internas.

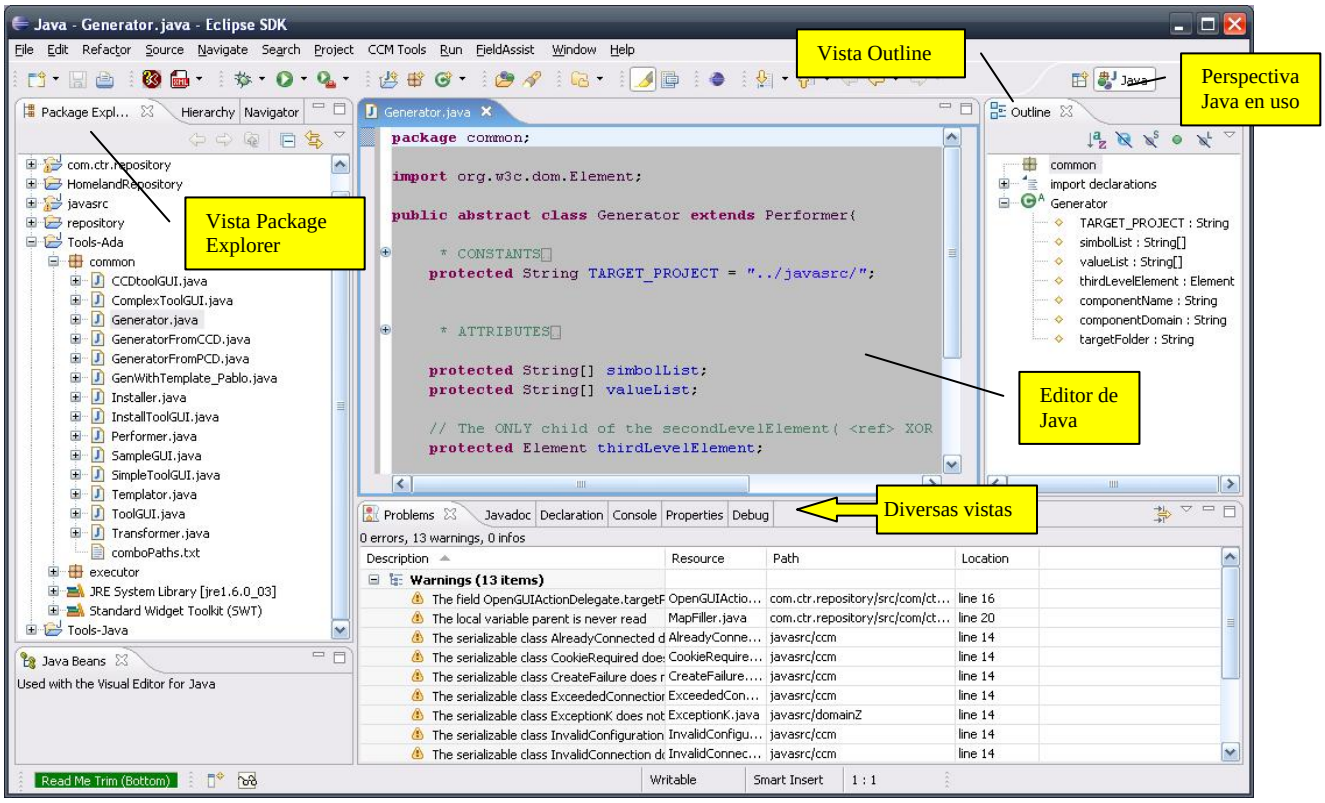



**Figura 3.1 – El *workbench* de Eclipse.**

## *3.3 Arquitectura de la plataforma Eclipse.*

### 3.3.1. Plug-ins y puntos de extensión.

La arquitectura basada en *plug-in*s es uno de los puntos fuertes de la plataforma Eclipse. Ésta tiene un pequeño núcleo cuyo propósito es la ejecución de los *plug-in*s que se instalan sobre él. Cualquier otra funcionalidad de Eclipse es proporcionada por ellos. De hecho, el propio núcleo también es formalmente un *plug-in*, constituyendo, junto con los *plug-in*s de compatiblidad y de lanzamiento, el mínimo conjunto de *plug-in*s requeridos por cualquier aplicación basada en Eclipse. Adicionalmente, el núcleo contiene algunas interfaces y clases de interés general, como la clase `Platform`, que gestiona todos los complementos instalados, la clase `Plugin` y la clase `Preferences`, que implementa una forma persistente de almacenar preferencias.

Esta arquitectura tiene la consecuencia de que Eclipse puede extenderse prácticamente de manera ilimitada desarrollando *plug-in*s y haciendo uso de la funcionalidad de los *plug-in*s ya existentes a través de los puntos de extensión que ofertan. Éstos juegan un papel crucial en la arquitectura basada en *plug-in*s. La idea central de construcción de un entorno en Eclipse es que un *plug-in* puede constituir una nueva extensión apoyándose en los puntos de extensión ofrecidos por otros plug-ins instalados, y así mismo, puede a su vez ofrecer sus propios puntos de extensión para que en ellos se puedan conectar posteriores *plug-in*s.

En la mayoría de los casos un *plug-in* consiste en un archivo Java. Lo que es requisito absoluto para cualquier *plug-in* es la existencia de un fichero de manifiesto `*.xml` que describa la configuración del *plug-in* y su integración en la plataforma. Este fichero es la clave para el desarrollo de *plug-in*s, pues controla cómo el *plug-in* se integra en el *workbench* así como el ensamblado del *plug-in* a partir de sus

componentes, es más, es aquí donde un *plug-in* describe a qué puntos de extensión se conecta y qué puntos de extensión añade a la plataforma.

### 3.3.2. Algunos *Plug-in*s.

Existen varios *plug-in*s disponibles para la implementación de GUIs. Estos incluyen SWT y JFace, pero también componentes de nivel superior como vistas, editores de texto y formularios, recogidos en los paquetes de la familia `org.eclipse.ui`. El *workbench* de Eclipse está implementado con la ayuda de estos *plug-ins* y podemos utilizarlos en nuestras propias aplicaciones sin dificultades en temas de licencias, pues todos ellos están cubiertos por la *Common Public License* 1.0.

También cabe mencionar la familia de *plug-ins* `org.eclipse.help`, que implementan un completo sistema de ayuda para el usuario final o la familia `org.eclipse.team`, que soportan el desarrollo de software en equipo.

### 3.3.3. Desarrollo de complementos.

Al crear *plug-ins* las cosas son bastante diferentes que en la creación de aplicaciones Java independientes, pues hemos de definir cómo se integra el *plug-in* en el *workbench* de Eclipse. Para soportar este proceso de desarrollo, Eclipse proporciona una perspectiva especial, la de desarrollo de *plug-ins*, que tiene como función especial el asistente para la creación de nuevos proyectos de desarrollo de *plug-in*s. Para crear uno nuevo hay varias plantillas disponibles y todas ellas resultan en *plug-ins* que pueden ser inmediatamente ejecutados y testeados. Una vez finalicemos de utilizar el asistente, éste crea los ficheros y la estructura de carpetas inicial del *plug-in*.

## *3.4 Gestión de recursos y entorno de programación.*

### 3.4.1. Jerarquía de recursos del *workspace* de Eclipse.

Eclipse distingue tres tipos básicos de recursos en un *workspace*: los proyectos, las carpetas y los ficheros.

a. Proyecto: es un nudo raíz de un árbol de recursos. Los proyectos son estructuras que contienen todos los recursos de un producto software y pueden controlar cómo éste es ensamblado a partir de sus componentes. Los proyectos no se pueden anidar, aunque pueden referirse a otros proyectos prerrequeridos y contener ficheros y carpetas, actuando como directorio raíz para ellos. Los proyectos pueden estar equipados con una o varias naturalezas. Cada naturaleza describe un aspecto conductual específico del proyecto.
b. Carpeta: puede contener ficheros y subcarpetas anidadas.
c. Fichero: nudo hoja en un árbol de recursos. Así pues, un fichero no puede contener otros recursos.


**Figura 3.2 – Jerarquía de recursos del *workspace***

### 3.4.2. Almacenamiento de los recursos.

En Eclipse todos los recursos se almacenan directamente en el sistema de ficheros del procesador en que se ejecuta (*host)* y, por tanto, la estructura de recursos en el *workspace* de Eclipse se correlaciona directamente con la estructura del sistema de ficheros del *host*, mapeándose los recursos directamente sobre los correspondientes elementos: proyectos y carpetas sobre directorios y ficheros sobre ficheros. Así pues, como cada recurso en el *workspace* se corresponde con un recurso en el sistema de ficheros anfitrión, cada uno tiene dos direcciones: la dirección dentro del *workspace* y la localización en el sistema de ficheros del *host*. La ventaja es que se puede acceder a los recursos aun cuando Eclipse no esté instalado o no esté funcional.

Por defecto, los recursos en Eclipse son almacenados en el directorio del *workspace* (representada su ruta por ../ ), el cual, también por defecto, es `eclipse/workspace` (naturalmente, es posible crear un directorio para el *workspace* en una localización diferente) y cada proyecto está contenido en un subdirectorio tal que `../nombreSubdirectorio`.

### 3.4.3. Sincronización de recursos.

Por cada recurso, Eclipse almacena una información reflectiva (*metadata*) en el directorio `../.metadata`. A veces ocurre que el estado de un recurso no coincide con el estado de los metadatos correspondientes (en particular esto sucede cuando un fichero del *workspace* es modificado fuera de Eclipse). En estos casos, para sincronizar de nuevo los recursos hay que aplicar la función de sincronización *Refresh*, función que puede ser aplicada también a carpetas y proyectos, con lo que podemos fácilmente resincronizar un árbol completo.

### 3.4.4. Navegación.

En el *workbench* de Eclipse, las dos vistas adecuadas a la navegación por los recursos son la vista `Navegador de Recursos` y el `Explorador de Paquetes`. La primera muestra los diferentes proyectos con su estructura de carpetas y ficheros dispuestos jerárquicamente en forma de árbol, de forma análoga al sistema de ficheros del *host* y nos permite navegar en la forma usual a través de ella. La segunda está contenida por defecto en la perspectiva Java y muestra los diferentes proyectos con su estructura de paquetes y las unidades de compilación. Los paquetes no son recursos reales sino virtuales y la estructura de paquetes de un proyecto se deriva de la declaración de paquete al comienzo de cada fichero Java. Las unidades de compilación pueden constar de varios recursos: el fichero fuente y uno o varios ficheros binarios (varios en el caso de haber clases internas o anidadas).

## *3.5 Las bibliotecas SWT y JFace.*

### 3.5.1. Introducción.

Eclipse no sólo posee un excelente entorno de desarrollo de Java, sino que también ofrece las librerías SWT y JFace que proporcionan recursos para implementar GUIs muy avanzadas, pudiendo sustituir con ventaja a las librerías AWT y Swing. La librería SWT implementa una interfaz independiente de la plataforma adaptada en cada caso al procesador en el que se encuentra instalado Eclipse, y sus clases simplemente delegan en las funciones de este sistema de ventanas nativo (*host WS*, en adelante). La principal ventaja de SWT es la integración de una aplicación basada en SWT con el *host*, con lo que tanto el *look&feel* como el grado de respuesta de aplicaciones SWT no son diferentes de los de aplicaciones nativas. Así pues, las aplicaciones basadas en SWT adoptan la apariencia del OS en el que se ejecutan y son indistinguibles de las interfaces de usuario de aplicaciones nativas. En cuanto a desventajas, SWT requiere gestión explícita de recursos porque utiliza recursos del *host WS* para las imágenes, los colores y las fuentes, recursos que deben ser liberados cuando ya no son necesarios. Finalmente, apuntar que, como la plataforma Eclipse está completamente implementada sobre la base de SWT, SWT debería ser nuestra primera elección cuando implementemos *plug-ins* para Eclipse.

JFace es una librería basada en el API SWT que proporciona al programador componentes para implementar GUIs de un nivel superior de abstracción. Ejemplos típicos de componentes ofrecidos por JFace son las acciones, los cuadros de diálogo, asistentes, visores, etc. Debido a esto, los componentes GUI de JFace también exhiben el *look&feel* nativo, a pesar de no poseer homólogos nativos. Aunque SWT proporciona una interfaz directa hacia los elementos nativos (*widgets)*, está limitado a usar tipos de datos simples y, aunque eso está bien para un gran número de aplicaciones, representa una severa

carencia al tratar datos orientados a objetos (OO, en adelante) que necesitan ser presentados en listas, tablas y árboles. Aquí es donde entran en juego los ya mencionados visores JFace -utilizados extensamente en el desarrollo de *plug-in*s para Eclipse- para proporcionar envoltorios OO entorno a los *widgets* SWT básicos, porque esa es precisamente la característica más notable de los visores (y en general de los componentes JFace): cada instancia envuelve un *widget* SWT (tabla, árbol, etc.) que es responsable de la representación de datos, haciendo más fácil el tratar con objetos de alto nivel de un dominio.

### 3.5.2. Visores JFace y su jerarquía.

A pesar del nombre visores, no solo soportan la visualización de contenidos, sino también la modificación de éstos. Su nombre proviene del patrón de diseño MVC (***M**odel **V**iewer **C**ontroller*), que define cooperación entre tres componentes: el modelo, que gestiona los datos del dominio, el visor, que es responsable de la representación de los datos en pantalla y el controlador, que maneja la interacción con el usuario. Además de una separación clara de conceptos, este patrón de diseño tiene la ventaja de que permite varias instancias de visor para una única instancia de modelo, lo cual a su vez permite mostrar los mismos datos de maneras distintas simultáneamente.

La primera distinción que podemos realizar es entre visores de texto, que facilitan el tratar con documentos de texto que requieren estilos complejos y los visores JFace de lista, de tabla y de árbol, que son visores estructurados y nos permiten utilizar directamente los objetos de nuestro modelo de dominio. Hacen esto proporcionando interfaces adaptadoras para tareas como, entre otras, obtener la etiqueta (imagen y texto) de un elemento y acceder a los hijos de un elemento, clasificar y filtrar elementos en una lista. A la derecha podemos ver un esquema de la jerarquía de visores de JFace y debajo varios ejemplos de visores.

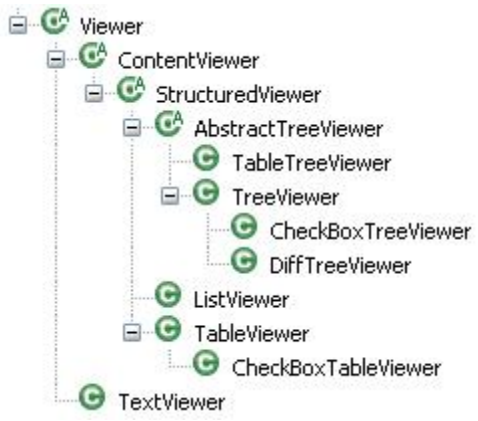



**Figura 3.3 – Jerarquía de visores**


**Figura 3.4 – Visor de texto.**

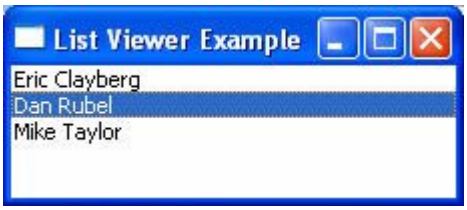



**Figura 3.5 – Visor de lista.**

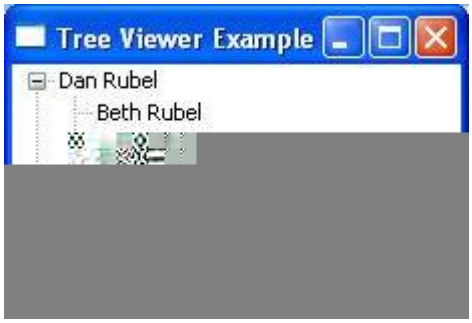



**Figura 3.6 – Visor de árbol.**

**Figura 3.7 – Visor de tabla.**

### 3.5.3. Adaptadores.

i. Proveedor de etiquetas (*label provider*): es otro tipo muy común de adaptador utilizado en visores estructurados. Es responsable de proporcionar una imagen y un texto para cada ítem contenido en el visor y, al igual que el *content provider*, el *label provider* acepta d-objetos como argumentos, es decir, es empleado para mapear un d-objeto en una o más imágenes y textos representables en el elemento SWT del visor. Los dos tipos más comunes de *label provider*s son los que implementan la interfaz `ILabelProvider`, utilizados en listas y árboles, y los que implementan `ITableLabelProvider`, empleados en tablas. El primer tipo mapea un d-objeto en un único par imagen-texto mientras que el segundo mapea un d-objeto en múltiples pares imagen-texto (uno por cada columna en la tabla).

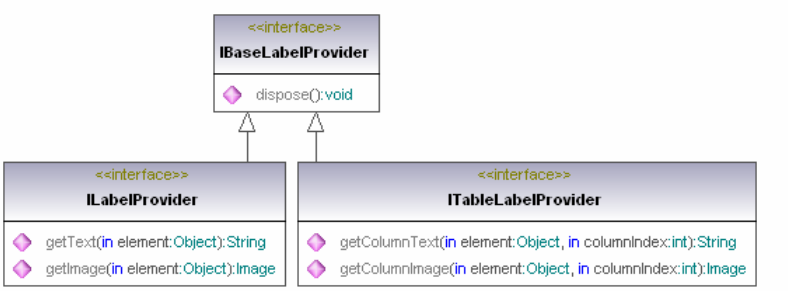



**Figura 3.8 – Jerarquía de proveedores de etiquetas.**

ii. Proveedor de contenido (*content provider*): es uno de los más comunes tipos de adaptador utilizados en visores estructurados. Al trabajar con un visor necesitamos proporcionarle información sobre cómo transformar nuestros d-objetos en ítems en el elemento SWT de la interfaz de usuario. Éste es el propósito de un *content provider*. Así pues, es usado para mapear entre un d-objeto (o una colección de ellos) utilizado(s) como entrada para el visor y la estructura interna requerida por el propio visor. Los dos tipos más comunes de *content provider*s son los que implementan `IStructuredContentProvider`, utilizados en listas y tablas, y los que implementan `ITreeContentProvider`, empleados en árboles. El primer tipo mapea una entrada del modelo de dominio (dominio, de ahora en adelante) en un *array* mientras que el segundo añade soporte para recuperar el padre y/o hijos de un elemento en un árbol.

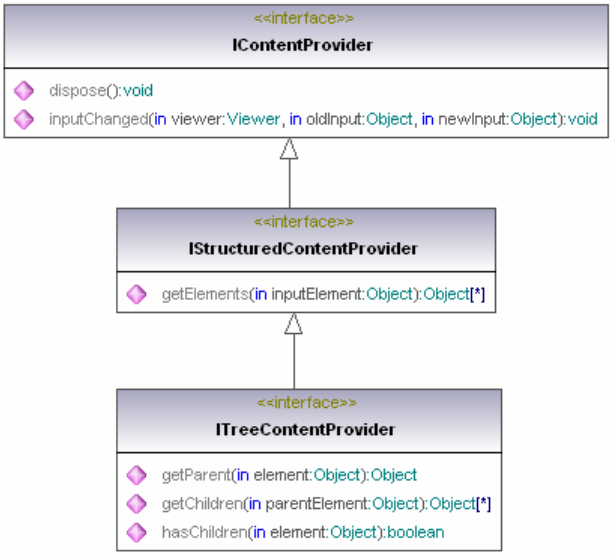



**Figura 3.9 – Jerarquía de proveedores de contenido.**

iii. Clasificador (*sorter*): es utilizado para ordenar los elementos proporcionados por el *content provider*. Si un visor no posee clasificador, los elementos son mostrados en el orden retornado por el *content provider*.

iv. Filtro (*filter*): es utilizado para mostrar un subconjunto de los elementos proporcionados por el *content provider*. Si un visor no posee filtro todos los elementos son mostrados.

### 3.5.4. Funcionamiento de un visor JFace.

En este apartado vamos a explicar los siguientes puntos:

1. Cómo poblar un visor.
2. Cómo reaccionar ante cambios en el dominio y cambios en la GUI.
3. Cómo seleccionar elementos en el visor.

Como hemos venido diciendo, JFace es un conjunto de herramientas GUI que ayudan a resolver tareas de programación comunes y que también actúa como puente entre *widgets* SWT de bajo nivel y nuestros objetos de dominio (d-objetos, en adelante), pues los *widgets* SWT interactúan con el *host* OS y por tanto no tienen conocimiento de nuestros d-objetos.

Una de las vías a través de las que JFace puentea el espacio entre los *widgets* SWT y nuestro modelo de dominio (simplemente dominio, en adelante) se basa en emplear visores. Los visores JFace consisten en un *widget* SWT (`Tree`, `Table`, etc.) más nuestros d-objetos y seremos nosotros los encargados de proporcionar a los visores la información que necesitan para poblar su *widget* SWT subyacente a partir de esos d-objetos. Un visor es capaz de clasificar y filtrar nuestros d-objetos así como actualizar el *widget* cuando dichos d-objetos cambian.

En nuestra herramienta los d-objetos serán los elementos de los ficheros XML mencionados en la sección …

Es crucial entender la relación modelo/vista utilizada por los visores JFace. Conceptualmente, todos los visores llevan a cabo dos tareas principales:

i. ayudan a adaptar nuestros d-objetos en entidades visualizables.
ii. proporcionan notificaciones cuando esas entidades visualizables son seleccionadas o cambiadas a través de la UI.

Más específicamente, cuando trabajamos con un visor, empleamos nuestros d-objetos como argumentos en los métodos del API. No necesitamos traducir nuestros d-objetos en elementos UI pues el visor lo hace por nosotros. ***Hacemos disponible nuestro d-objeto raíz para el visor invocando el método setInput() y así dicho d-objeto se convierte en la entrada del visor***. Cuando establecemos la entrada del visor, éste trabaja en colaboración con el *content provider* y el *label provider* para visualizar el *widget* SWT. De igual forma, cuando preguntamos al visor por los ítems seleccionados, éste responderá con los d-objetos, no con los recursos UI subyacentes.

#### 3.5.4.1 Selección.

En la mayoría de aplicaciones, el usuario selecciona un ítem del visor con el propósito de llevar a cabo alguna acción específica. Podemos ser notificados de estas selecciones añadiendo un escuchador de cambios de selección al visor y así, cuando una selección ocurre en su seno, él notificará a cada uno de sus escuchadores de cambio de selección, pasando un evento que describe qué ha sido seleccionado. Notar que estos eventos fluyen en sentido opuesto a los eventos generados por cambios en el dominio.

#### 3.5.4.2 Respondiendo al cambio.

Las aplicaciones no son muy útiles si no manejan el cambio. Conceptualmente, el cambio puede ser descrito de dos formas. Cuando los d-objetos cambian, la interfaz de usuario usualmente refleja esos cambios y, asimismo, acciones de usuario en la interfaz pueden requerir actualizaciones en los d-objetos.

- Respondiendo a cambios en el modelo de dominio.

Como se ha indicado previamente, cuando un cambio ocurre en el dominio, la UI necesita reflejar ese cambio. Puesto que no queremos contaminar los d-objetos con conocimiento acerca de la interfaz de usuario porque si el modelo y la vista están demasiado vinculados se vuelven frágiles al cambio, emplearemos un patrón de notificación de eventos para romper ese acoplamiento no deseado.

En nuestro caso, conseguimos esto creando una interfaz escuchadora a la que notifican nuestros d-objetos cuando ocurre un cambio interesante. Ahora necesitamos proporcionar una instancia de ella para escuchar esos cambios. Típicamente esa instancia será el *content provider* del visor, que se registrará a si mismo como escuchador de los cambios en d-objetos de forma que pueda notificar al visor de esos cambios, lo cual lo hará llamado a uno de sus métodos de actualización y es que un *tree viewer* proporciona tanto un método `update`, que simplemente actualiza la etiqueta del d-objeto dado, como un método `refresh`, que refresca el d-objeto y todos sus hijos.

- Respondiendo a cambios en la UI.

También necesitamos responder a cambios hechos en la UI, cambios que a menudo causan que un d-objeto cambie. Para ello, típicamente, crearemos escuchadores y los añadiremos al visor. Los métodos de nuestros escuchadores recobrarán del visor o del objeto de eventos pasado los d-objetos necesarios. Por ejemplo, en nuestra herramienta estamos recuperando el d-objeto a partir del evento de selección, pero igualmente podríamos haber preguntado al visor por los objetos seleccionados.

## *3.6 Aplicación de la tecnología XML: W3C-Schema y herramientas de análisis.*

### 3.6.1. El lenguaje XML (*e**X**tensible **M**arkup **L**anguage*).

XML es un lenguaje meta-etiquetado propuesto por W3C para formatear documentos de texto de forma que sean fácilmente interpretables mediante programas. Algunas de sus características son que la información en un documento XML viene siempre expresada mediante cadenas de caracteres y dispuesta en forma de elementos delimitados por una etiqueta que describe su naturaleza y semántica, de tal manera que el marcado del contenido del documento representa la estructura de los datos que alberga. Sin embargo, su principal característica es que está orientado a la portabilidad -de hecho es el lenguaje más portable que se haya propuesto desde los ficheros basados en ASCII-, en el sentido de que pretende codificar datos los cuales puedan ser mostrados y modificados por cualquier editor de texto y puedan ser interpretados por cualquier plataforma de cualquier versión.

Esto último representa una motivación particularmente importante para su utilización tanto en este trabajo como en general en la tecnología de componentes CCM en la que se enmarca pues uno de sus posibles ámbitos de aplicación es el desarrollo de aplicaciones distribuidas cuyo plan de despliegue puede considerar una plataforma heterogénea compuesta por nodos procesadores de distinto tipo.

### 3.6.2. Tecnología XML.

XML tiene asociados un conjunto de recursos para explicitar su estructura, su interpretación y la forma en que los documentos XML han de ser visualizados, así como diferentes APIs para el procesado de dichos documentos desde programas. Entre estos recursos se incluyen lenguajes para extender, transformar y describir el contenido de documentos XML así como estándares para la presentación de documentos XML y para formular documentos de dominios especiales.

De entre ellos vamos a detenernos en los siguientes tres, debido a su utilización tanto en esta tesis como en el la tecnología que la engloba.

### 3.6.3. W3C-Schema.

Es un lenguaje que permite formalizar el modelo de la estructura de datos de un tipo de documento XML así como las restricciones de los tipos de datos que se pueden incorporar en él. Al definir una plantilla *schema* para un tipo de documento XML estamos definiendo la estructura de datos que puede contener el documento, los tipos de datos que pueden declararse en él y un vocabulario que a su vez define la semántica de los datos. Así mismo, estaríamos proporcionando una plantilla que sirve de referencia para que diversas herramientas validen automáticamente los documentos XML que sean implementaciones suyas y asistan a la creación del documento.

Un ejemplo de utilización de la tecnología W3C-Schema es la adaptación del estándar *D&C* de OMG a la tecnología CCM (figura 2.6).

### 3.6.4. DOM (***D**ocument **O**bject **M**odel*).

DOM es un API estándar para crear, procesar y transformar información representable mediante estructuras de tipo árbol. Gracias a ello, una de las aplicaciones de DOM, que es anterior a XML, es manejar la información contenida en un documento XML, aunque esto no implica que funcione en sentido inverso, esto es, no define los recursos para construir la información a partir de otros formatos como pueda ser un documento XML.

Su utilización ha sido básica en la implementación de las herramientas relacionadas en esta tesis, pues tanto en la herramienta de navegación, que veremos en la sección 3.6, como en las herramientas de generación automática de código que aparecen en el capítulo 4, la entrada siempre es, o al menos engloba, un fichero XML cuya estructura en forma de árbol nos interesa tener a nuestra disposición para poder navegar por él.

### 3.6.5. SAX (***S**imple **A**PI for **X**ML*).

Mencionamos en último lugar SAX, otro API para el procesado de documentos XML desde programas, en particular para el procesado secuencial de un fichero XML.

Puesto que no aporta la funcionalidad de navegación sobre estructuras de tipo árbol que proporciona DOM, nos hemos limitado a su utilización para un fin muy concreto durante la creación de la herramienta de navegación. Veremos en la sección 3.6 que esta herramienta comprende varios ficheros XML de soporte (uno por cada *schema* `*_Data_Model.xsd` de la figura … ) que contienen la información asociada a cada uno de los elementos que constituyen los ficheros XML validados por dichos *schemas*. Veremos también que la herramienta necesita analizar los primeros antes de poder representar el contenido de los segundos en forma de árbol, y como ese análisis sólo es necesario realizarlo una vez y puede ser en forma secuencial, hemos preferido hacer uso de SAX en lugar de DOM.

## *3.7 Repositorio de componentes y plataformas (Repository).*

### 3.7.1. Repositorio.

Como ya mencionamos en la sección 2.5, el repositorio es uno de los dos elementos básicos que, junto a las herramientas, conforman el entorno de desarrollo propuesto en este trabajo. Se trata de una estructura de ficheros que se define como un proyecto de naturaleza general en el *workspace* de Eclipse para, jugando el rol de base de datos, registrar toda la información requerida y generada en el entorno de desarrollo de componentes CCM. Su propósito es múltiple:

i. Almacenar la información del entorno de desarrollo de aplicaciones basadas en componentes. Con ello, toda la información asociada a los componentes disponibles, provengan estos de

nuevos desarrollos o de adquisiciones de otros desarrollados por terceros, se registra en el repositorio, donde, así mismo, existe almacenada información sobre las plataformas de ejecución especificadas y sobre aplicaciones disponibles ya desarrolladas y dispuestas para su ejecución.

ii. Servir de base a las herramientas de diseño y desarrollo de aplicaciones.

iii. Asignar una localización preestablecida a cada producto que se genera, lo cual reduce el número de parámetros de entrada y salida que requieren las herramientas.

iv. Ofrecer un contexto adecuado para que puedan operar las herramientas predefinidas del entorno, como los editores, compiladores, depuradores, etc.

En función de la información almacenada en dicho repositorio se analiza el plan de despliegue, se seleccionan las implementaciones de los componentes que son compatibles con los recursos de los procesadores en donde se instancian, se introduce el tipo de conector que corresponde a las conexiones entre los componentes y se distribuye y ejecuta el código de cada componente de acuerdo con las instrucciones incluidas en sus modelos.

### 3.7.2. Estructura del repositorio.

El repositorio almacena de forma organizada la información relativa a los elementos registrados que se utilizan en el desarrollo de las aplicaciones. En el primer nivel de organización se definen seis carpetas que almacenan de forma separada lo referente a las diferentes categorías de elementos: **`interfaces`**, **`components`**, **`applications`**, **`platforms`**, **`technology`** y **`schemas`**. Internamente, la información se organiza por dominios que definen diferentes espacios de nombres. Cualquier elemento en el repositorio se identifica utilizando los cuatro segmentos `<section>/<domain>/<name>.<extension>`

- Carpeta **`applications`:** Contiene la información relativa a las aplicaciones que se encuentran bajo desarrollo.
- Carpeta **`components`:** Contiene la información relativa a la descripción de los componentes instalados en el repositorio del entorno de desarrollo.
- Carpeta **`DnC schemas`:** contiene los *schemas* que adaptan el estándar *D&C*.
- Carpeta **`interfaces`:** Contiene la información asociada a la descripción de las interfaces instaladas en el repositorio del entorno de desarrollo.
- Carpeta **`platforms`:** Contiene la información asociada a las plataformas de ejecución.
- Carpeta **`technology`:** Almacena la información relativa a la tecnología de componentes que se utiliza en el desarrollo de los componentes y de las aplicaciones

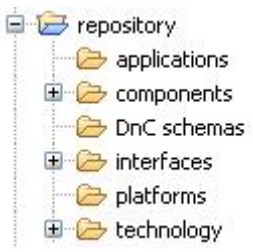



**Figura 3.10 – Estructura del repositorio.**

## *3.8 Desarrollo de una herramienta de navegación en forma de plug-in.*

### 3.8.1. Objetivo del trabajo.

Hemos aprovechado la funcionalidad ofrecida por Eclipse en cuanto a desarrollo de *plug-in*s para implementar en esa forma una herramienta de navegación con la que visualizar en forma de árbol los elementos de los ficheros XML almacenados en nuestro repositorio que se hallen soportados por alguno de los *schemas* referidos en la sección 2.4 (figura 2.6).

El objetivo que se pretende con el desarrollo de este *plug-in* es disponer de una perspectiva `(Repository)` que incorpore la vista predefinida `Resources Navigator` y las vistas propias de este *plug-in*, llamadas `DnC Outline` y `Attributes`, que deseamos que aparezcan en cuadro de diálogo `Show View` bajo una nueva categoría también llamada "`Repository`". Si alguno de los ficheros mostrados en la vista `Navigator` es un fichero XML soportado por esos *schemas*, al seleccionarlo su contenido ha de mostrarse en forma de árbol en la vista `DnC Outline`, comenzando por el elemento raíz del fichero XML como nudo raíz de dicho árbol, el cual aparecerá inicialmente colapsado. La vista `Attributes` muestra una tabla con tres columnas tituladas "`name`", "`type`" y "`value`" y su función es mostrar esa terna de ítems para cada uno de los atributos del elemento sobre el que hayamos hecho clic en la vista `DnC Outline`. En caso de que el elemento no posea atributos, la tabla de la vista `Attributes` no muestra nada.

Al principio concebimos la herramienta de forma que cada elemento del fichero XML seleccionado aparecería como nudo en el árbol `DnC Outline`, mostrándose como un par icono-etiqueta. El icono lo creamos en base a la semántica del elemento, pudiendo suceder que dos elementos pertenecientes a un mismo tipo definido en el *schema* correspondiente tuviesen asociados iconos diferentes en caso de diferir su semántica o que dos elementos pertenecientes a tipos diferentes tuviesen asignado el mismo icono. Respecto a las etiquetas, realizamos un análisis de cada tipo declarado en los *schemas* para elegir qué atributo de entre los poseídos por un elemento aportaría su valor como etiqueta en el árbol, tomando como criterio adicional que si un tipo no declara atributos o el atributo elegido para un tipo no es poseído por el elemento a mostrar por ser éste un atributo opcional, entonces se mostraría en el árbol el nombre local del elemento entre signos '<' y '>'. Como excepción, en algunos casos optamos por mostrar un literal predeterminado. Sin embargo, un estudio posterior de los *schemas* considerados fue realizado con el objetivo de implantar ciertas imposiciones sobre la estructura jerárquica que se deseaba representar en el árbol. De esta forma convenimos que los elementos correspondientes a ciertos tipos declarados en los *schemas* no se mostrarían en el árbol, sino que, aun aportando su icono, su lugar sería ocupado por sus hijos, los cuales aportarían la etiqueta necesaria. Inicialmente decidimos que los tipos no mostrables fuesen aquellos que poseen un elemento hijo a elegir entre `<ref>`, `<location>` o `<description>`, los cuales, como ya hemos dicho, no poseerían icono asociado, sino que utilizan el de su padre (elemento no visible). Además, si el elemento hijo es uno de los dos primeros, poseedores de un nudo de texto como única descendencia, dicho nudo de texto -que contiene una ruta, relativa al repositorio en el caso de `<ref>` o absoluta en el caso de `<location>`- será la etiqueta a mostrar. Estos nudos `<ref>` o `<location>` serán en su estado inicial nudos de hoja, esto es, aparecerán sin una cruz de expansión, pero un doble clic sobre ellos permite invocar al fichero XML referenciado y añadirlo como subárbol al ya existente, colgando el elemento raíz del referenciado a partir del nudo hoja, que dejará de serlo. Las consideraciones extraídas del estudio de los *schemas* se recogen en la tabla del anexo […].

Por último, hemos querido dotar a la vista `Resources Navigator` de varios filtros para ocultar ficheros de determinadas extensiones. En un principio hemos añadido las opciones de filtrar los formatos `*.java`, `*.idl` y `*.xml`.

### 3.8.2. Recursos creados: iconos y archivos XML origen de mapeados.

i. Iconos para mostrar en el árbol contenido en la vista `DnC Outline`. Ha sido necesario crear toda una colección de iconos, uno para cada d-objeto (elemento XML) que lo requiriera. Todos ellos se han diseñado utilizando el programa Adobe Photoshop CS2 en formato `*.gif` y en un tamaño de 16 x 16 píxeles.

ii. Ficheros XML de soporte (uno por cada *schema*) que contienen la información asociada a cada uno de los elementos que constituyen los ficheros XML validados por dichos *schemas*. Estos

ficheros XML de soporte obedecen a la denominación `NombreDelSchema_Map.xml` y la información se presenta en la forma:

```
<root>
    <node localName="localNameOfTheElement"
          labelToShow="nameOfOneOfHisAttributes"
          iconToShow=" localNameOfTheElement.gif"
          appear="true / false"/>
    <node ...
    .....
<root/>
```

Durante la fase de desarrollo del *plug-in*, estos recursos permanecen almacenados en las carpetas `icons` (que se crea por defecto al iniciar en Eclipse un nuevo proyecto de desarrollo de *plug-ins*) y `maps` (creada al efecto). Sin embargo, cuando un *plug-in* está terminado y en disposición de ser distribuído como archivo `*.jar`, estas carpeta `icons` y `maps` penderán directamente de la raíz de dicho archivo JAR.

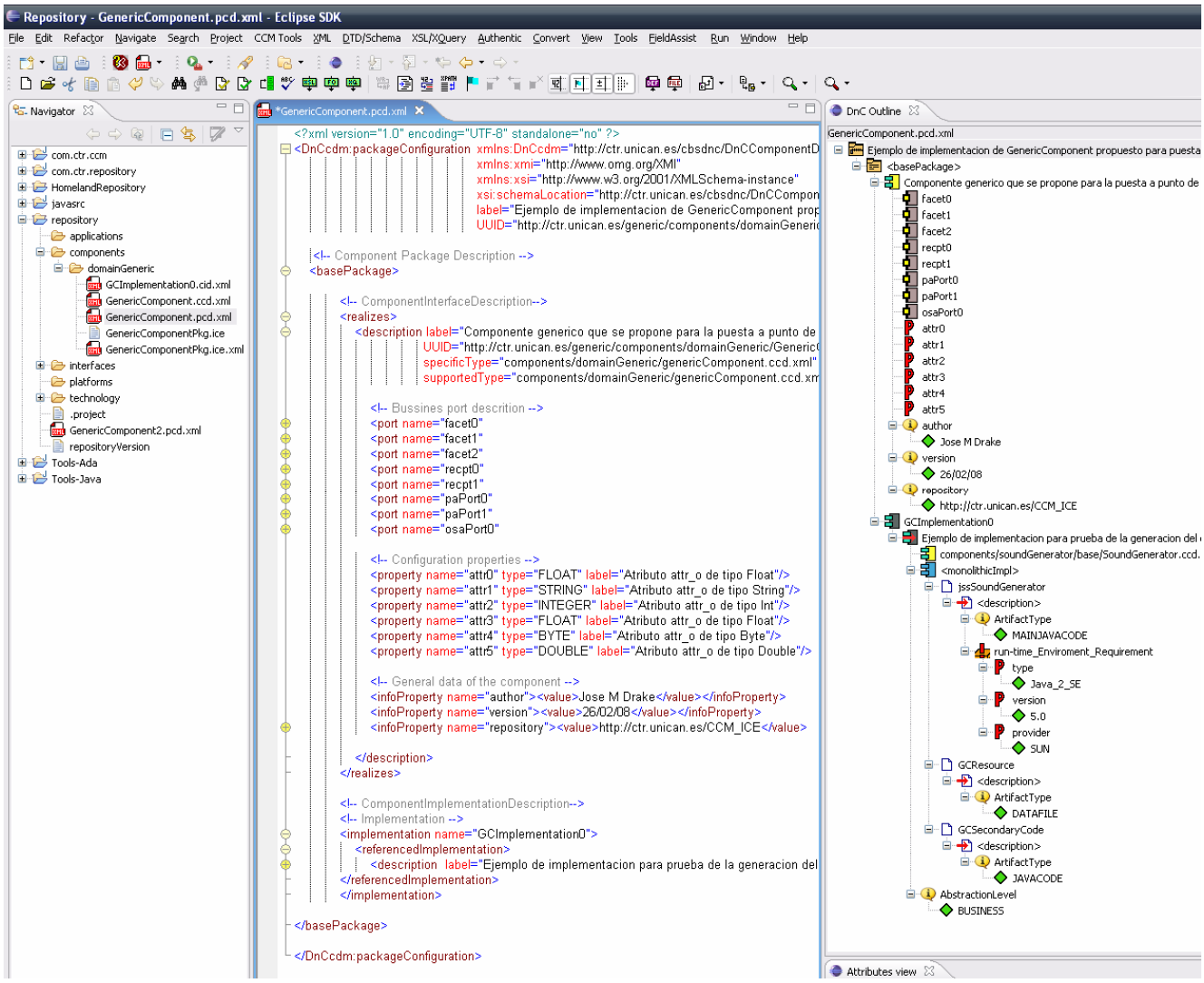



**Figura 3.11 – Herramienta de navegación.**

# 4. DISEÑO DE HERRAMIENTAS DE GENERACIÓN DE CÓDIGO.

## *4.1 Generación de código en base a plantillas.*

El proceso de generación de código en base a plantillas que hemos desarrollado en este trabajo tiene por entradas y salidas los elementos que se muestran en la figura 4.1.

Entradas:

- Modelo de datos (almacenado en el repositorio de componentes)
- Diccionario de patrones.
- Reglas de generación de código.

Salida:

- Código generado.

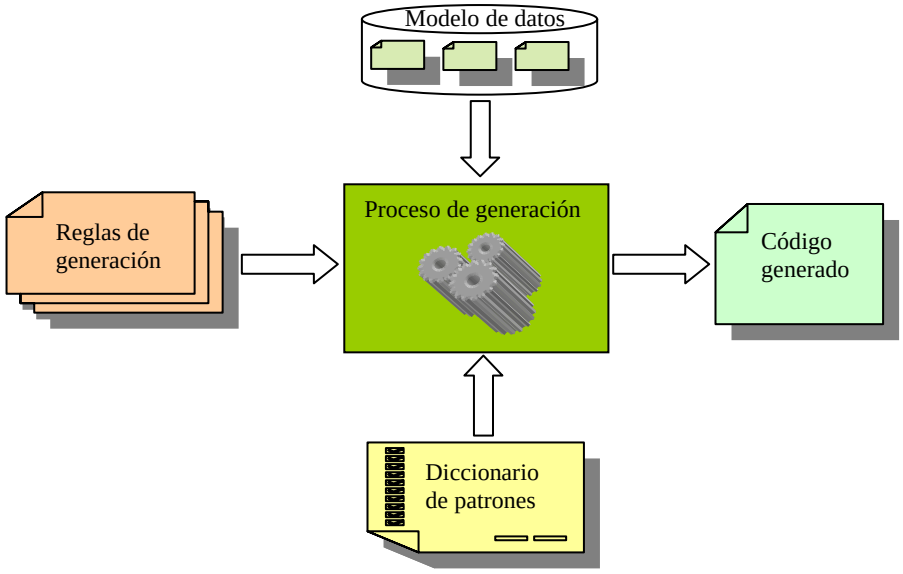



**Figura 4.1 – Elementos de entrada y salida en el proceso de generación de código en base a plantillas.**

La herramienta de generación es un programa diseñado para que lleve a cabo las tareas correspondientes a los tres pasos en que se descompone este proceso (ver figura 4.2) y en base a las directrices establecidas por las reglas de generación de código que gobiernan la sección de generación de código de la última de las tres etapas, sección que siempre supone el grueso del proceso.

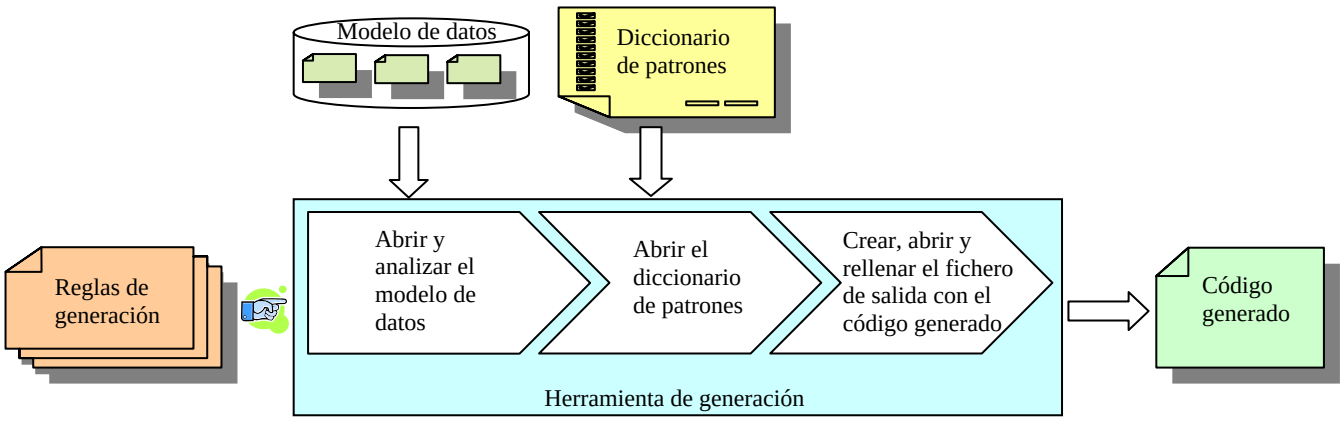



**Figura 4.2 – Herramienta de generación de código. Entradas y salidas.**

La etapa de generación del contenido del fichero de salida viene descrita por un conjunto de reglas de generación estructurado como normas para escribir bloques de código y que están apoyadas en los patrones recogidos en el fichero del mismo nombre. La estrategia seguida para confeccionar estas normas y los patrones en que se basan es analizar la estructura genérica del tipo de fichero de salida que queremos producir y determinar qué colección de fragmentos de código (caracteres, expresiones, sentencias completas o incluso secuencias de líneas) podemos encapsular en forma de patrones (cuya estructura veremos en la siguiente sección) de modo que la composición a que dan lugar dichos patrones siguiendo las reglas de generación constituya el fichero de salida deseado.

En ocasiones un mero tratamiento secuencial de los patrones por parte de las reglas permite generar el código de salida, pero en general no será tan sencillo y por ello el espectro de patrones ha de ser tal que siempre podamos encontrar una combinación adecuada de ellos de manera que seamos capaces de cubrir todas las posibilidades en cuanto a estructura de código que pueda adoptar el fichero de salida.

## *4.2 Especificación de las plantillas.*

Un diccionario de patrones suele presentarse como un fichero de texto que contiene una sucesión de patrones con la siguiente estructura:

```
<nombrePatrón>
 contenido del patrón
 </nombrePatrón>
```

No hay límite en cuanto a la extensión del contenido del patrón, pudiendo contener un número arbitrario de saltos de línea y el sangrado que quien haya creado los patrones haya estimado conveniente. Los patrones pueden estar parametrizados (con los parámetros delimitados por el carácter '@' en la forma `@parámetro@`) o no estarlo, y, tanto si lo están como si no, pueden emplearse para generar código en una única ubicación del fichero de salida o en múltiples posiciones, mediante repetición en caso de ser un patrón no parametrizado o mediante reutilización otorgando diferentes valores a los parámetros en caso de ser un patrón parametrizado.

- Ejemplo de patrón no parametrizado empleado para generar código en una única ubicación:

```
<importBegin>
import Ice.*;
import ccm.*;
</importBegin>
```

- Ejemplo de patrón parametrizado empleado para generar código en una única ubicación:

```
<package>package @domain@;
</package>
```

- Ejemplo de patrón no parametrizado empleado para generar código en múltiples ubicaciones:

```
<try>
try{</try>
```

- Ejemplo de patrón parametrizado empleado para generar código en múltiples ubicaciones:

```
<catch>
}catch(@exception@ e){
    throw new @exception@(e.mssg);</catch>
```

### *4.3 Especificación de la funcionalidad de los generadores de código.*

En este trabajo hemos diseñado y adoptado un formato estándar para especificar la funcionalidad de una herramienta de generación de código que finalmente ha dado lugar a la siguiente plantilla.

**Herramienta para generar [X]**

- Entradas y salidas:
  - Entrada(s):
    1. **Modelo de datos**: Especificar aquí el formato en que se proporciona el modelo de datos y aportar un ***anexo que contenga lo más relevante de su estructura y una leyenda de términos*** en la que se detalle el significado de cada parámetro que aparezca en el diccionario de patrones o en rutas y nombres de ficheros.
    2. **Diccionario de patrones**: Especificar aquí el medio en que se proporciona el diccionario de patrones.
    3. **Entradas adicionales**: Especificar aquí otros datos complementarios que necesite la herramienta.
  - Salida(s):
    1. **Código** Java de [X] en el fichero `nombreFichero.java`

- Pasos del proceso de generación:
  1. **Apertura y análisis del modelo de datos**: Especificar aquí qué acciones se han de llevar a cabo sobre el modelo de datos.
  2. **Acceso al diccionario de patrones**: Especificar aquí qué acciones de acceso al diccionario de patrones se han de llevar a cabo.
  3. **Creación y apertura del fichero** que contendrá el código generado: Especificar aquí la ubicación donde ha de crearse el fichero.
  4. **Rellenado del fichero de salida** según las siguientes reglas de generación: Especificar a continuación, en forma de lista enumerada, qué bloques de código se han de ir escribiendo y en qué patrón se hallan contenidos (mostrar a continuación dicho patrón).
     i. Escribir bloque NOMBRE_BLOQUE_1 (contenido en el patrón <**nombrePatrón_1**>)

     ```
     <nombrePatrón_1>
     Contenido del patrón_1.
     </nombrePatrón_1>
     ```

     ii. Escribir bloque NOMBRE_BLOQUE_2 (contenido en el patrón <**nombrePatrón_2**>)

     ```
     <nombrePatrón_2>
     Contenido del patrón_2.
     </nombrePatrón_2>
     ```

     iii. Y así sucesivamente…
  5. **Cerrar el fichero de salida**.

- Mostrar el aspecto del código de salida

La documentación entregada al desarrollador de la aplicación sigue este formato expuesto. En el anexo A se puede ver como ejemplo la plantilla aplicada al caso de la herramienta *HomeWrapper* junto con su propio anexo (anexo B) conteniendo la estructura y leyenda del modelo de datos.

Herramientas de generación.

La estrategia que hemos seguido para el diseño de herramientas de generación de código contempla los siguientes aspectos y requerimientos:

a. **Integración en el entorno de desarrollo:** Nuestras herramientas se incorporan al *workbench* de Eclipse en forma de *plug-in*, el cual añade:
   i. Un nuevo elemento de menú a la barra de menús de Eclipse, el cual, al desplegarse, ofrece acceso a todas las herramientas creadas. Seleccionando una de ellas aparece en pantalla una GUI adecuada a la herramienta elegida y mediante la cual llevar a cabo el proceso de generación de código.
   ii. Un nuevo submenú al menú contextual que aparece al hacer clic en un fichero del workspace (típicamente del repositorio) con el botón derecho del ratón, tanto en la vista Explorador de Paquetes como en la vista Navegador de Recursos. Este submenú, a diferencia del anterior menú de la barra de menús, sólo ofrece, de entre todas las herramientas creadas, aquellas susceptibles de ser aplicadas al fichero seleccionado.

b. **Interacción con el usuario:** La GUI mencionada en el apartado anterior ha de poseer el aspecto y la funcionalidad apropiada a la herramienta a la que está asociada, pero en cualquier caso su comportamiento obedece al siguiente patrón:
   i. Permite al usuario introducir los datos requeridos, como la ruta del fichero con el modelo de datos o cualquier otra información adicional. Respecto a esa ruta, si se accede a la herramienta mediante el submenú contextual, la GUI que aparece en pantalla la recibe por defecto en un elemento GUI destinado a albergarla (cuadro de texto, combo desplegable, etiqueta, etc.) y si se accede mediante el menú de la barra de menús, ha de poder introducirse tecleando directamente en ese elemento GUI o navegando a través del sistema de ficheros del *host OS*. En este último caso, el punto de inicio de la navegación y el tipo de ficheros mostrados durante ella los establece el desarrollador de la aplicación.
   ii. Una vez que la ruta deseada se encuentra contenida en el elemento GUI al efecto, la GUI ha de informar al usuario de la validez o invalidez del fichero escogido, según esté establecido en la implementación de la herramienta qué extensiones son válidas y, en caso de ser un formato válido, habilitar en secuencia ordenada los subsiguientes elementos GUI de introducción de información adicional (en caso de ser ésta necesaria) hasta, finalmente, habilitar el elemento GUI que permite ejecutar el proceso de generación y cerrar la ventana.

c. **Arquitectura:** Cada una de las herramientas que implementemos utilizando la arquitectura que hemos diseñado (ver figura 4.3) se compone de una clase que representa su generador de código (núcleo de la herramienta) más otra clase que representa su GUI (ver figura 4.4). Las dos clases fundamentales en esta arquitectura son las clases abstractas `Performer` y `ToolGUI`. La clase `Performer` juega el papel de superclase de aquellas clases que representen un generador concreto de código para una herramienta dada, mientras que la clase `ToolGUI` es la superclase de la que heredan las diferentes GUIs de nuestras herramientas.

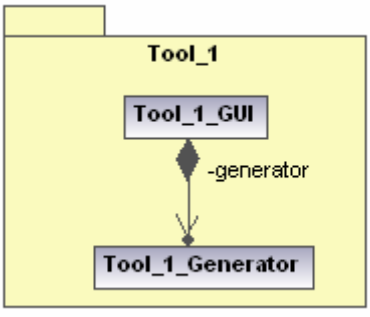



**Figura 4.3 – Composición de una herramienta.**

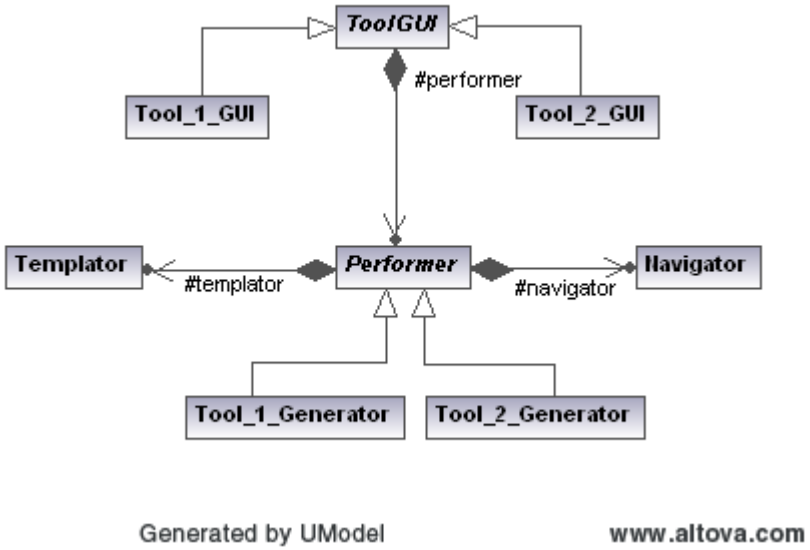



**Figura 4.4 – Arquitectura de herramientas de generación.**

- Clase `Performer`:
  - ATRIBUTOS.
    - `protected Templator templator`
      Este campo proporciona a la herramienta la funcionalidad en cuanto a creación del fichero de salida y su rellenado con código generado en base a patrones.
    - `protected Navigator navigator`
      Este campo proporciona a la herramienta la funcionalidad en cuanto a la obtención de los elementos hijos de un nudo en un árbol DOM.

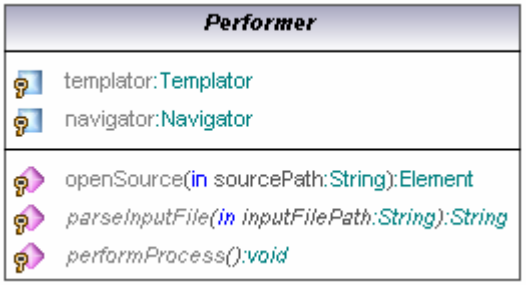



**Figura 4.5 – Clase `Performer`**

  - MÉTODOS.
    - `protected Element openSource(String sourcePath)`
      Este método abre y parsea, haciendo uso de DOM, el fichero XML de entrada almacenado en la ruta especificada por el argumento y retorna su elemento raíz.
    - `protected abstract String parseInputFile(String inputFilePath)`
      Este método invoca al método anterior `openSource` para obtener el elemento raíz del fichero XML de entrada y navega a partir de él obteniendo la información que sea necesaria.
    - `protected abstract void performProcess()`
      Este método ejecuta el proceso de generación consumiendo las etapas citadas en apartado 4.1.

Los dos últimos métodos son métodos abstractos que han de ser implementados por las subclases de `Performer` que implementen generadores concretos de código.

- Clase `Templator`: La clase `Templator` ofrece un API para el manejo de los ficheros involucrados en el proceso de generación (obtención del contenido del fichero de patrones, apertura y parseo del fichero XML de entrada y creación, apertura y escritura del código generado en el fichero de salida).

  - ATRIBUTOS.
    - `private TreeMap<String,String> templatesMap`
      Este campo representa un mapa donde almacenamos la información recogida en el fichero de plantillas en forma clave-valor, esto es, cada patrón del fichero de plantillas se registra como entrada en el mapa donde su etiqueta es la clave y su contenido es el valor.
    - `private PrintWriter pWriter`
      Este campo representa el *stream* de caracteres utilizado para la escritura del fichero de salida
    - `private String offSet`
      Este campo representa el sangrado que aplicamos al texto que escribimos en el fichero de salida. Comienza inicializado al *string* nulo y hemos de gestionarlo sumando o restando tabuladores de forma que el sangrado del texto de salida sea coherente.

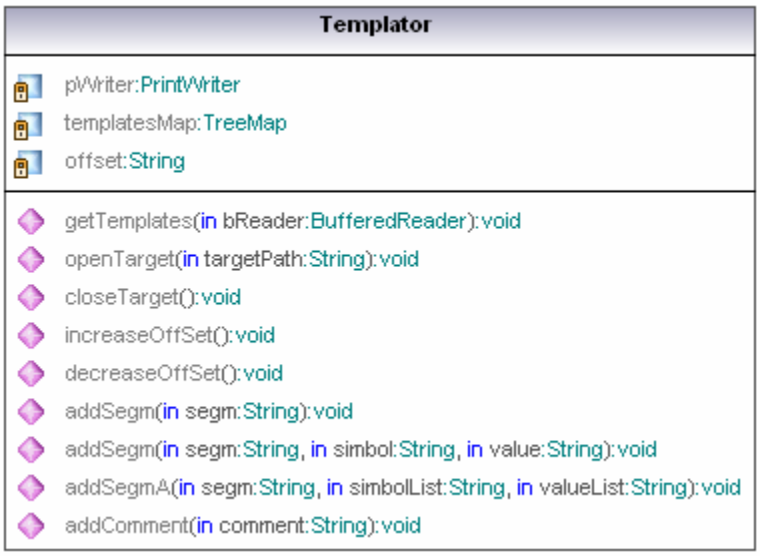



**Figura 4.6– Clase `Templator`.**

  - MÉTODOS.
    - `public void getTemplates(BufferedReader bReader) throws IOException`
      Este método acepta un *stream* de caracteres que ha de proporcionar el contenido del fichero de plantillas y para cada patrón copia su etiqueta y contenido en forma de entrada clave-valor en el atributo `templatesMap`.
    - `public void openTarget(String targetPath) throws IOException`
      Este método crea y abre el fichero de salida en la ruta especificada por el argumento. En el caso de que la localización no exista, se crea la estructura de carpetas requerida.
    - `public void closeTarget()`
      Este método cierra el *stream* de caracteres utilizado para escribir en el fichero de salida.
    - `public void increaseOffSet()`
    - `public void decreaseOffset()`
      Estos métodos incrementan y decrementan respectivamente en un tabulador el atributo `offSet`.
    - `public void addSegm(String segm)`
    - `public void addSegm(String segm, String simbol, String value)`
      Este método está sobrecargado. En ambas versiones añade al fichero de salida el valor de la entrada en el atributo `templatesMap` correspondiente a la clave indicada por el parámetro `segm`, esto es, el contenido del patrón cuya etiqueta es `<segm>` en el fichero de plantillas. La diferencia es que en la segunda versión se reemplaza cada aparición del parámetro `symbol` (que se supone que ha de aparecer entre '`@`' en los patrones) por el parámetro `value`.

- `public void addSegmA(String segm, String[] simbolList, String[] valueList)`
  Al igual que en los casos anteriores, este método añade al fichero de salida el valor de la entrada en el atributo `templatesMap` correspondiente a la clave indicada por el parámetro `segm`, esto es, el contenido del patrón cuya etiqueta es `<segm>` en el fichero de plantillas, pero ahora reemplazando cada aparición de cada uno de los *strings* en el parámetro `symbolList` (que se supone que han de aparecer entre '`@`' en los patrones) por el respectivo *string* en el parámetro `valueList`.
- `public void addComment(String comment)`
  Este método escribe en el fichero de salida y en forma de comentario Java, esto es, precedido de //, el texto pasado como parámetro.

- Clase `Navigator`: Esta clase ofrece un API para obtener los elementos hijos de un nudo en un árbol DOM.

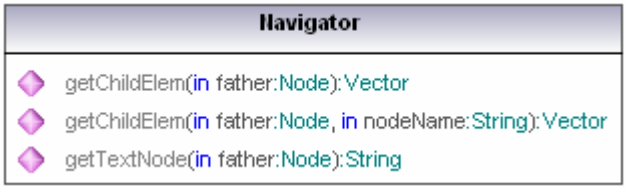



**Figura 4.7– Clase `Navigator`.**

  - MÉTODOS.
    - `public static Vector<Element> getChildElem(Node parent)`
    - `public static Vector<Element> getChildElem(Node parent, String nodeName)`
      Este método está sobrecargado. En ambas versiones retorna un vector que contiene elementos hijos del nudo especificado por el parámetro `parent`. La diferencia es que en la primera versión retorna todos los elementos hijos y en la segunda los filtra y sólo retorna aquellos con nombre igual al parámetro `nodeName`.
    - `public static String getTextNode(Node father)`
      Este método retorna el nudo de texto hijo del nudo especificado por el parámetro `parent`. Si no existe tal nudo de texto hijo el método retorna el *string* nulo.

## *4.4 Ejemplos de herramientas.*

### 4.4.1. Relación de herramientas desarrolladas.

En el marco de este trabajo hemos desarrollado toda una familia de herramientas que podemos clasificar en tres tipos:

- Herramientas de generación de código: la documentación entregada para especificar la funcionalidad de cada uno de estos generadores de código se adjunta en el anexo A.
  - H1: BIMGenerator.
  - H2: ReceptacleConnectionGenerator.
  - H3: ExecutorGenerator.
  - H4: ContextGenerator.
  - H5: WrapperGenerator.
  - H6: HomeWrapperGenerator.

- Herramientas de transformación de código:
  - XML2SliceTransformer.
  - Slice2XMLTransformer (*)
  - Slice2JavaCompiler (**)
- Herramientas de importación:
  - InterfaceInstaller.
  - ComponentInstaller.

**(*)** Es esta una herramienta de transformación de código que no ha seguido el paradigma de generación de código en base a plantillas, pero que se incluye aquí por simetría respecto a la herramienta xml2slice.

**(**)** Es esta una herramienta que simplemente consiste en la llamada a una herramienta externa de ICE. Se incluye en el *plug-in* por practicidad.

Además de esta clasificación semántica también es muy importante atender al formato del fichero de entrada y a si es necesario algún tipo de información adicional más. En el caso de las herramientas H1 y H2 el modelo de datos puede venir descrito indistintamente en un fichero `*.ccd.xml` o en un fichero `*.pcd.xml` y no se necesita ninguna información adicional de entrada, mientras que en el caso de las herramientas H3, H4, H5 y H6 necesitan que se escoja una implementación concreta del componente y por tanto sólo ficheros `*.pcd.xml` son aceptables.

### 4.4.2. Integración en el entorno de desarrollo e interacción con el usuario.

Como ya dijimos, la estrategia seguida consiste en que las herramientas creadas se integren en el *workbench* de Eclipse incluidas en un *plug-in* que aporta un elemento de menú a la barra de menús y un submenú contextual para proporcionar el **acceso** a dichas herramientas.

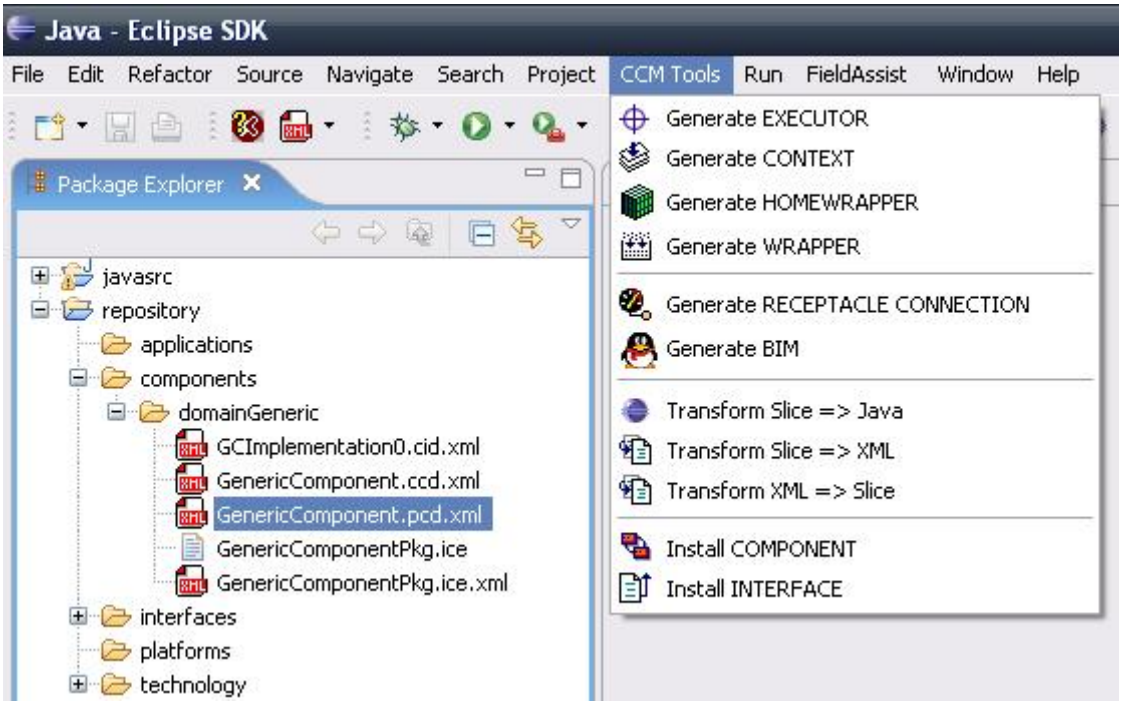



**Figura 4.8 – Menú añadido a la barra de menús del *workbench* de Eclipse.**

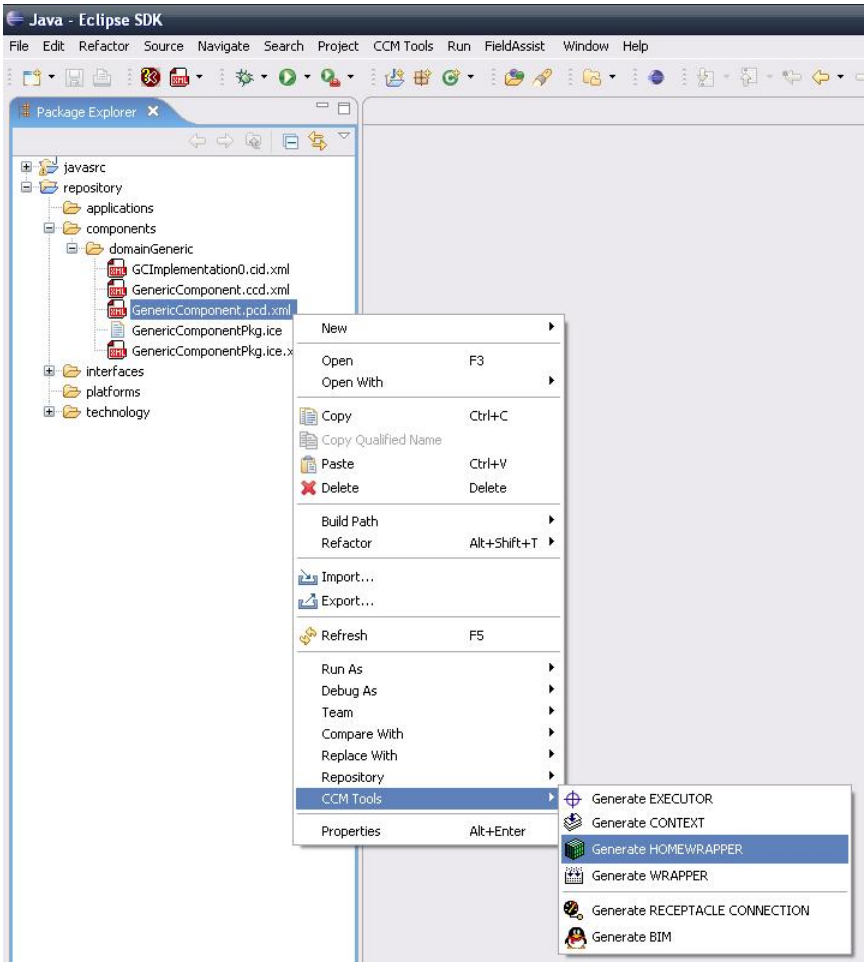



**Figura 4.9 – Submenú añadido al menú contextual.**

Para las herramientas H1, H2, H7, H8 y H9 hemos diseñado el formato de GUI de la figura 4.10 y para las herramientas H3, H4, H5 y H6 hemos diseñado una variante de la primera un poco más compleja (figura 4.11) para poder dar cabida a la necesidad de elegir la implementación del componente.


**Figuras 4.10 y 4.11 – GUI básica y variante más compleja.**

Como puede apreciarse, hemos optado por utilizar como elemento GUI para la introducción de la ruta del fichero de entrada que representa nuestro modelo de datos un objeto de la clase `Combo`, luego, además de poder introducirla tecleando, podemos recuperar de su lista desplegable una ruta ya insertada anteriormente.

Para permitir la búsqueda del fichero de entrada mediante navegación a través del sistema de ficheros en la forma habitual que proporcione el *host WS*, Eclipse ofrece la clase `FileDialog` (paquete `JFace`) que, como vemos en la figura 4.12 representa un cuadro de diálogo con esa funcionalidad y con aspecto nativo. Al pulsar en el botón Browse… de la GUI se abre una instancia de esta clase. Como ya dijimos, el punto de inicio de la navegación y el tipo de ficheros mostrados los establece el desarrollador de la aplicación (en este caso la carpeta `components` del repositorio y ficheros de extensión `*.pcd.xml`) y a partir de ahí navegamos hasta dar con el fichero deseado, lo seleccionamos y pulsamos en el botón Abrir. Tras ello, el cuadro de diálogo se cierra y la ruta del fichero seleccionado aparece en el combo.

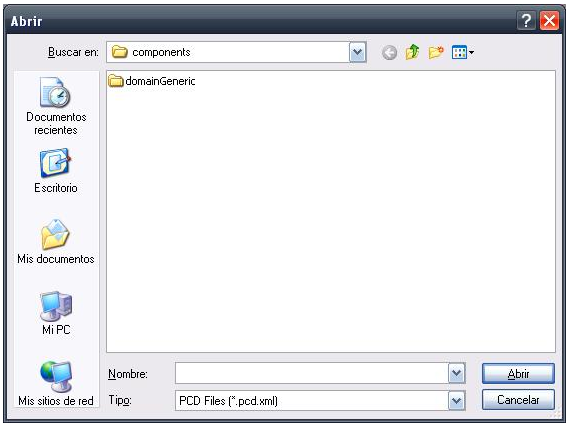



**Figura 4.12 – Cuadro `FileDialog`.**

Una vez introducida la ruta de entrada (si se ha tecleado directamente hay que pulsar a continuación ENTER), la GUI informa de la validez o invalidez del fichero mostrando en el cuadro de información de proceso los mensajes **"VALID input file name"** o **"NON-VALID input file name"** respectivamente. En caso de ser válido, se habilita el segundo combo de elección de la implementación del componente y, finalmente, una vez elegida se habilita el botón de ejecución del proceso para que, pulsando sobre él, tenga éste lugar. Si se trata del formato simple de GUI, el hecho de que un fichero sea válido conduce directamente a la habilitación del botón de ejecución.

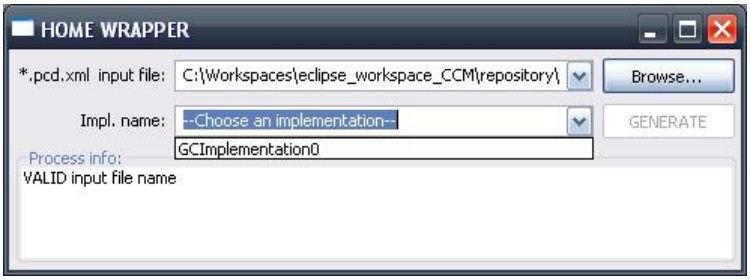
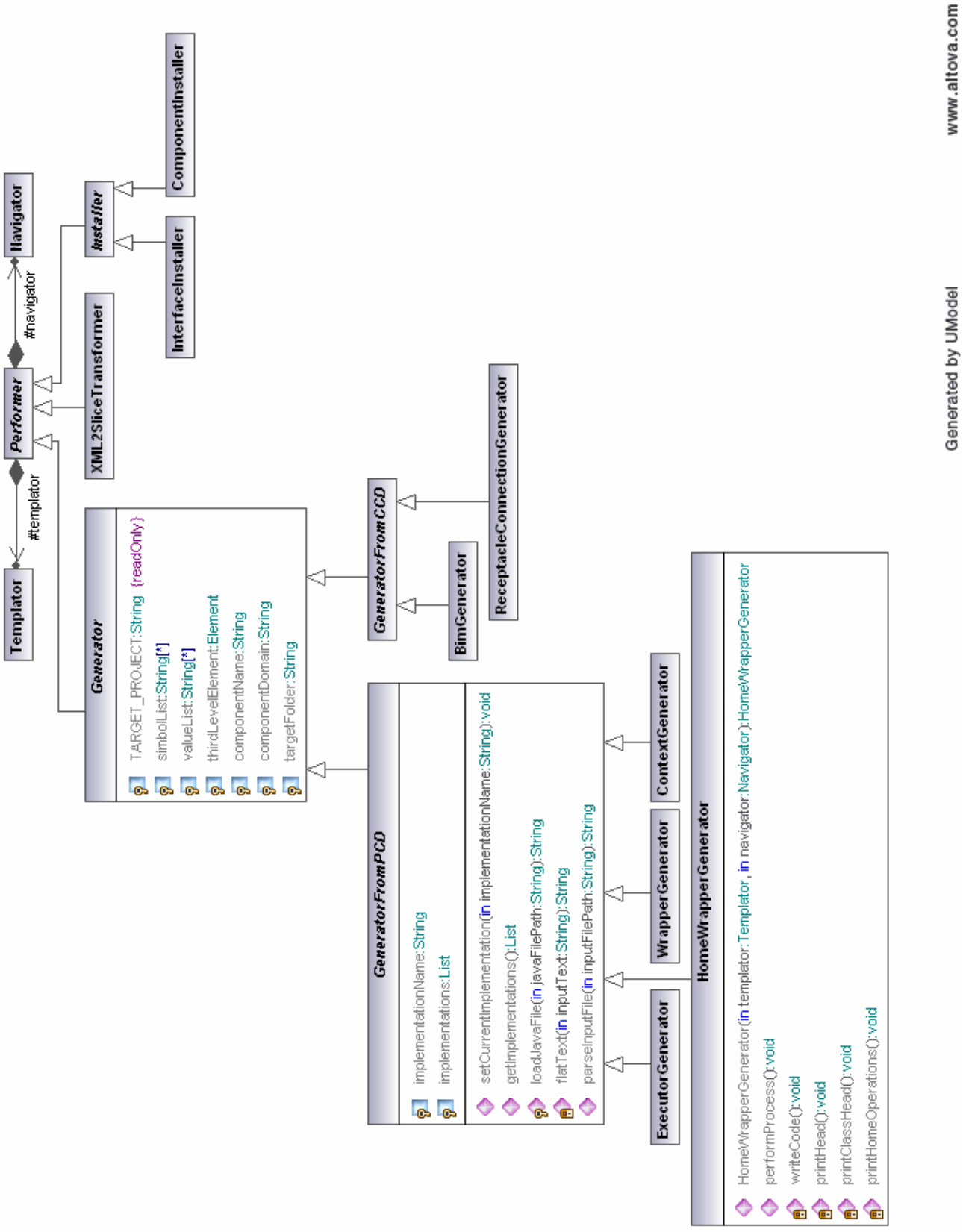



**Figura 4.13 - Elección de la implementación del componente.**

### 4.4.3. Arquitectura:

La arquitectura de la familia de herramientas desarrolladas se fundamenta en la arquitectura mostrada en las figuras 4.3 y 4.4, pero con múltiples niveles de herencia para posibilitar la reutilización máxima de código, tanto al extender a la clase abstracta `Performer` como al extender a la también clase abstracta `ToolGUI`. Lo vemos en forma de diagramas UML en las figuras 4.14 y 4.15, donde mostramos en detalle a las clases `HomeWrapperGenerator` y `HomeWrapperGUI` a modo de ejemplo ilustrativo.


Navigator
#navigator
Performer
#templator
Templator
Installer
InterfaceInstaller
ComponentInstaller
XML2SliceTransformer
Generator
TARGET_PROJECT:String {readOnly}
simbolList:String[*]
valueList:String[*]
thirdLevelElement:Element
componentName:String
componentDomain:String
targetFolder:String
GeneratorFromCCD
BimGenerator
ReceptacleConnectionGenerator
GeneratorFromPCD
implementationName:String
implementations:List
setCurrentImplementation(in implementationName:String):void
getImplementations():List
loadJavaFile(in javaFilePath:String):String
flatText(in inputText:String):String
parseInputFile(in inputFilePath:String):String
ExecutorGenerator
WrapperGenerator
ContextGenerator
HomeWrapperGenerator
HomeWrapperGenerator(in templator:Templator, in navigator:Navigator):HomeWrapperGenerator
performProcess():void
writeCode():void
printHead():void
printClassHead():void
printHomeOperations():void
Generated by UModel
www.altova.com

- **Clase `Generator`.**

  - CONSTANTES
    - `protected static final String TARGET_PROJECT = "../javasrc/";`
      Nombre y ubicación en el workspace del proyecto donde han de almacenarse los ficheros con el código generado.

  - ATRIBUTOS
    - `protected String[] simbolList;`
    - `protected String[] valueList;`
      Atributos para ser utilizados al invocar métodos de la clase `Templator`.
    - `protected Element thirdLevelElement;`

    - `protected String componentName = "";`
    - `protected String componentDomain = "";`
      Atributos para representar información del componente.
    - `protected String targetFolder = "";`

- **Clase `GeneratorFromPCD`**

  - ATRIBUTOS
    - `protected String implementationName = "";`
    - `protected List<String> implementations = null;`
      Atributos para almacenar la implementación del componente elegida y la lista de todas ellas.

  - MÉTODOS
    - `public void setCurrentImplementation(String implementationName)`
    - `public List<String> getImplementations()`
      Métodos getter y setter para los atributos.

    - `protected String loadJavaFile(String javaFilePath) throws IOException`
    - `private String flatText(String inputText)`
    - `public String parseInputFile(String inputFilePath)`
      Ver método abstracto del mismo nombre en la clase `Performer`.

- **Clase `HomeWrapperGenerator`.**

  - CONSTANTES
    - `private static final String TEMPLATE_FILE_NAME = "HomeWrapperTemplate.txt";`
      Fichero de patrones.

  - MÉTODOS
    - `protected void performProcess(String inputFilePath) throws IOException`

- `public void performProcess() throws IOException`
  Ver métodos abstractos del mismo nombre en la clase `Performer.` En este caso se implementa la versión sin argumentos.

- `private void printCode()`
- `private void printHead()`
- `private void printClassHead()`
- `private void printHomeOperations()`
  Estos métodos simplemente encapsulan código dentro del método `performProcess`

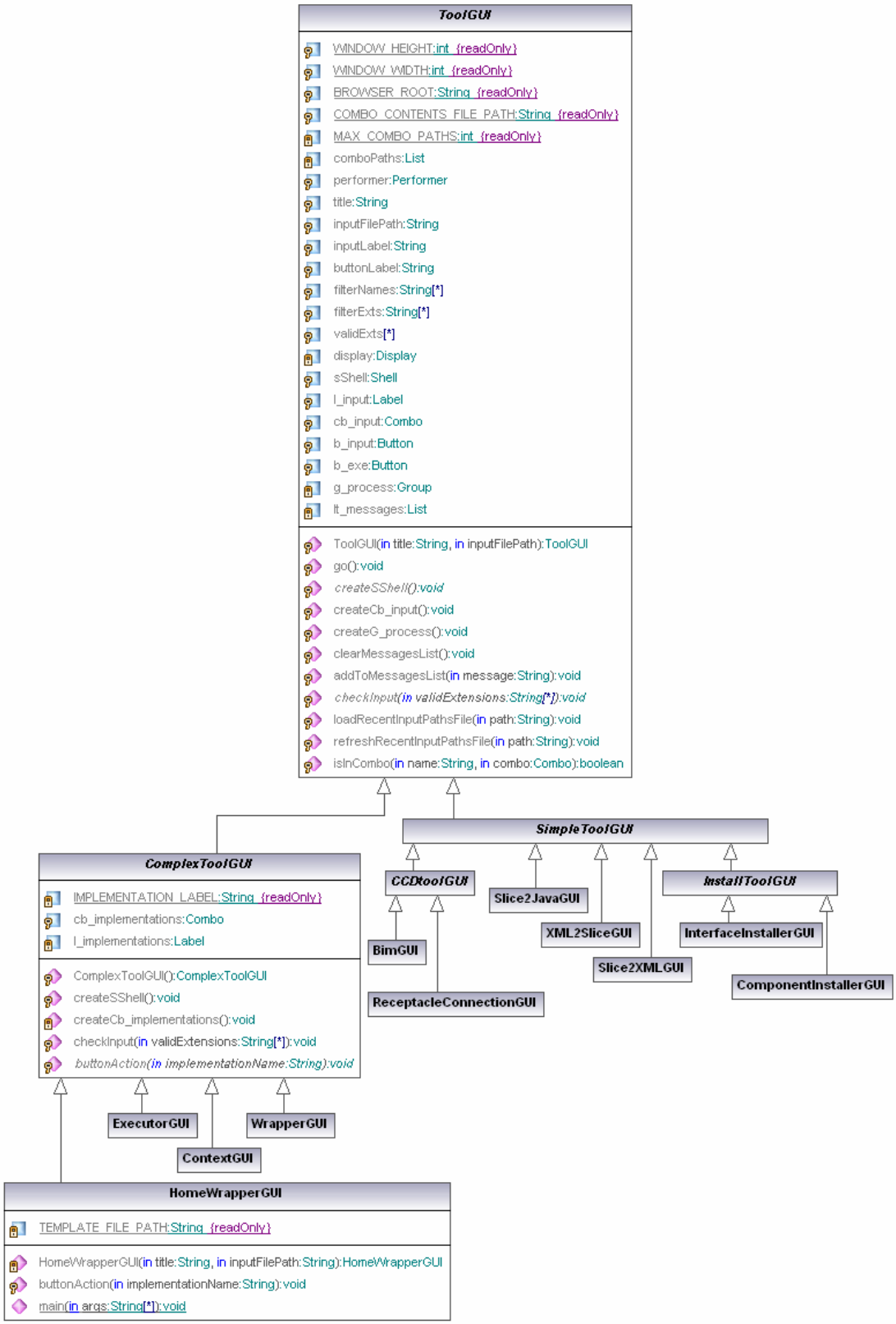



**Figura 4.15 – Arquitectura de la parte gráfica de usuario en nuestras herramientas.**

- **Clase `ToolGUI`**

  - CONSTANTES
    - `private static final int WINDOW_WIDTH`
    - `private static final int WINDOW_HEIGHT`
      Dimensiones de la ventana de la GUI.

    - `private static final String BROWSER_ROOT`
      Punto de inicio del navegador al abrirse el cuadro de diálogo tras pulsar el botón Browse…

    - `private static final String COMBO_CONTENTS_FILE_PATH`
      Ubicación del fichero de persistencia.

    - `private final int MAX_COMBO_PATHS`
      Máximo número de rutas que puede almacenar el combo de entrada en su lista desplegable.

  - ATRIBUTOS.
    - `private java.util.List<String> comboPaths`
    - `protected Performer performer`

    - `protected String title`
    - `protected String inputLabel`
    - `protected String buttonLabel`
      Texto del los atributos `sShell`, `l_input` y `b_exe`

    - `protected String inputFilePath`
      Atributo donde se almacena la ruta del fichero de entrada introducida por el usuario.

    - `protected String[] filterNames;`
    - `protected String[] filterExts;`
      Tipos de ficheros mostrados en el navegador del cuadro de diálogo tras pulsar el botón Browse..

    - `protected String[] validExts;`
      Extensiones que definen qué ficheros son válidos para la herramienta. Se establecen en el constructor de la GUI concreta de la herramienta.

    - `private Display display`
      Este atributo conecta la aplicación Java al OS.

    - `protected Shell sShell` `protected Label l_input`
    - `protected Combo cb_input` `protected Button b_input`
    - `protected Button b_exe` `private Group g_process`
    - `private List lt_messages` Elementos GUI (ver figura )

  - MÉTODOS.
    - `protected abstract void createSShell()`
    - `protected void createCb_input(String[] validExtensions)`
    - `protected void createG_process()`
      Estos métodos son para creación e inicialización de los elementos GUI representados por los atributos `sShell`, `cb_input` y `g_process`

- `protected void clearMessagesList()`
- `protected void addToMessagesList(String message)`

- `protected void buttonAction(String implementationName)`
  Este método ejecuta el proceso de generación de código y cierra la ventana.

- `protected void go()`
  Este método ejecuta la aplicación. En él se llevan a cabo las siguientes acciones:
  i. Se captura una instancia `Display`.
  ii. Se carga el contenido del fichero de persistencia.
  iii. Se crea y abre la ventana en que se basa la GUI.
  iv. Se implementa el bucle de eventos del que, en SWT, es responsable el programador. Sin este bucle la interfaz se bloquearía mientras se ejecuta el programa, lo cual se resuelve con ayuda de un método que lee eventos en espera en la instancia `Display` y los pasa al escuchador de la interfaz. Si no hay eventos en espera se invoca a un método de suspensión que espera hasta que se desencadena un nuevo evento.

- `protected abstract void checkInput(String[] validExtensions)`
  Este método comprueba si el nombre de un fichero (típicamente fichero con el que finaliza la ruta que constituye el contenido del combo `cb_input`) es válido, esto es, si termina en alguna de las extensiones indicadas por el argumento. De ser así, se habilitan ciertos controles de la GUI, la ruta es añadida a la lista desplegable del combo `cb_input` si no se encuentra ya en ella y el fichero de persistencia es actualizado.

- `protected void loadRecentInputPathsFile(String path)`
  Este método crea y abre un fichero de persistencia (concebido para albergar las rutas de entrada recientes) en la ubicación especificada por el argumento y carga sus primeras 10 entradas en un objeto `java.util.List`

- `protected void refreshRecentInputPathsFile(String path)`
  Este método actualiza el fichero de persistencia mencionado anteriormente, el cual se encuentra almacenado en la ruta especificada por el argumento.

- `protected boolean isInCombo(String name, Combo combo)`
  Este método lleva a cabo una búsqueda del texto especificado por el primer argumento en el `Combo` especificado por el segundo argumento y devuelve un booleano indicando el resultado de la búsqueda.

- **Clase `ComplexToolGUI`**

  - CONSTANTES
    - `private static final String IMPLEMENTATION_LABEL = "Impl. name:"`
      Texto del atributo `l_implementations.`

  - ATRIBUTOS.
    - `protected Label l_implementations`

- `private Combo cb_implementations`
  Elementos GUI (ver figura)

- MÉTODOS.
  - `protected void createSShell()`
    [Ver método abstracto del mismo nombre]

  - `private void createCb_implementations()`
    Método para creación e inicialización del elemento GUI representado por el atributo `cb_implementations`

  - `protected void checkInput(String[] validExtensions)`
    [Ver método abstracto del mismo nombre]

  - `protected abstract void buttonAction(String implementationName)`
    Este método ejecuta el proceso de generación de código y cierra la ventana.

- **Clase `HomeWrapperGUI`**

  - CONSTANTES
    - `private static final String TEMPLATE_FILE_PATH = "./homeWrapper/HomeWrapperTemplate.txt";`
      Ubicación del fichero de patrones.

  - MÉTODOS.
    - `private void buttonAction(String implementationName)`
      [Ver método abstracto del mismo nombre]

    - `public static void main(String[] args)`
      Este método representa el punto de entrada de la aplicación. En él se instancia la GUI y se invoca a su método encargado de ejecutar la aplicación.

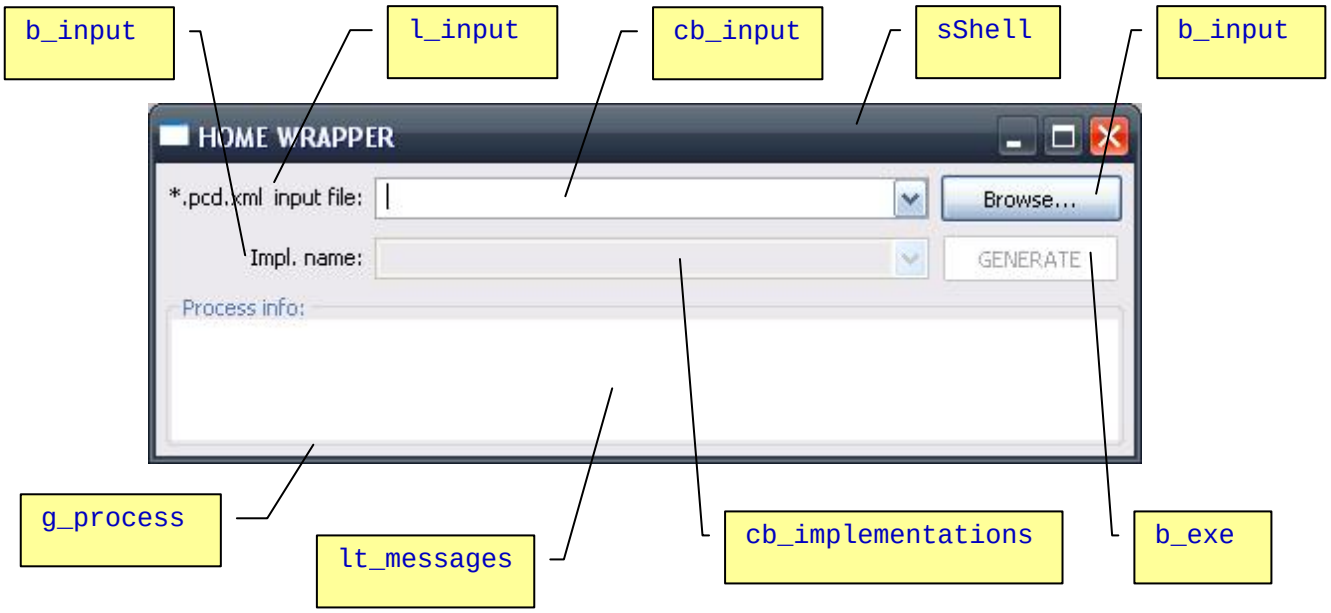



**Figura 4.16 – Elementos GUI.**

# 5. CONCLUSIONES Y LÍNEAS FUTURAS.

Como respuesta a la necesidad que existe en el grupo de investigación de crear un entorno de desarrollo para la tecnología de componentes para sistemas embebidos y de tiempo real, en este trabajo se han abordado muchas tecnologías diferentes pero complementarias que son la base de las herramientas software y los entornos CASE, y que creemos que transcienden el objetivo inicial planteado. Entre estas tecnologías podemos citar:

- **Tecnologías XML**: La tecnología XML busca sobre todo la transportabilidad. Es una idea antigua: Un documento codificado en caracteres alfanuméricos puede ser generado en cualquier máquina y leído en cualquier otra máquina. Al ser actualmente la base de la tecnología WEB, se ha difundid a todos los campos, y se ha creado ardedor de ella una tecnología y un conjunto de herramientas que facilitan su uso. El contenido y formato de un documento XML puede describirse mediante otro documento XML (W3C-Schema) lo que facilita su gestión automática en las herramientas. Así, por ejemplo, si se utiliza herramientas DOM, se pueden analizar sintácticamente documentos de estructura muy compleja con sólo un número mínimo de sentencia de lenguaje.
- **Interfaces de usuario**: La tecnología de interfaces gráficas de usuario ha sido la solución a la interacción del hombre-maquina. En los años 90, se desarrollo la tecnología con diferentes familias de elementos que permitían construir cualquier interfaz con sólo componerlos en un entorno de desarrollo. El avance actual, es la introducción además de patrones de diseño, como por ejemplo con `JFace` de Eclipse, que hacen posible que con sólo unas líneas de código, crear un elemento complejo de interacción un visor, un editor o un navegador ya adaptado a la aplicación concreta en la que se necesita.
- **Entorno Eclipse**: Eclipse representa la puesta a disposición de la comunidad que genera software libre de una infraestructura abierta y flexible que sirva de crisol de todos los que contribuyen al desarrollo de entornos de desarrollo de software basados en herramientas automáticas. Su arquitectura modular basada en la capacidad de interconectar componentes (*plug-ins*) de forma estandarizada permite construir entornos de desarrollo complejos sin escribir una línea de código y con sólo componer los módulos que ya están disponibles en los catálogos.
- **UML**: La admisión por la comunidad software del lenguaje universal de modelado (UML) como herramienta para describir el software de forma gráfica y conceptual, e independiente del código ha abierto la puerta al desarrollo de software a partir de modelos, utilizando estrategias de programación generativa, en las que los diseñadores de aplicaciones sólo se mueven en el mundo conceptual de los modelos, y son las herramientas las que interpretan los modelos y generan el código que ejecutan los computadores.

Todas estas tecnologías tienen como substrato común el lenguaje de programación Java. Su modernidad basada en el paradigma orientado a objetos, que permite abordar eficientemente la complejidad, su sencillez basada en un núcleo de especificación mínimo y en un riquísimo conjunto de librería, y la independencia de la plataforma que transmite a la aplicaciones que se desarrollan con él, han hecho que el mundo de las herramientas y de los entornos orientados a la interacción con los usuario, entre otros, se basen en él. Sin embargo, el que sea Java la base de la tecnología no esta libre de peligros, ya que las aplicaciones basadas en él, requieren muchos recursos de memoria, y lo que es peor, recursos de memoria que son muy difíciles de estimar. En la situación actual, los entornos basados en Java no son escalables, y cuando con ellos se utilizan sistemas complejos se hacen muy ineficientes.

La línea de trabajo sobre la que considero que debe continuar este trabajo es la utilización de la tecnología MDA (***M**odel **D**riven **A**rchitecture*) en el desarrollo de entornos de herramientas para el diseño

de software. La técnica que hemos seguido en este trabajo se basa en el desarrollo directo de las herramientas por un programador. Esta es una estrategia muy pesada ya que al final requiere que un experto en el entorno desarrolle las aplicaciones siguiendo las indicaciones de los expertos en la tecnología que se está automatizando. La alternativa es no desarrollar mediante código herramientas, sino meta-herramientas, esto es herramientas para generar herramientas. La metodología MDA, propone como estrategia el desarrollo de meta-herramientas, que reciban como entrada los modelos de la información de la que se parte y los modelos de la información a la que se quiere llegar, y sea capaz de generar la herramienta que lleve a cabo esa transformación. Para este paradigma, actualmente se está estandarizando las formas en que deben formularse los modelos para facilitar la generación de herramientas. Los estándares EMF (***E**clipse **M**odeling **F**ramework*), MOF (***M**eta-**O**bject **F**acility*), CWM (***C**ommon **W**arehouse **M**etamodeling*) XMI (**X**ML ***M**odel **I**nterchange*), etc. van todos orientados en esta dirección.

# 6. REFERENCIAS.

[1] K.Czarnecki and U. Eisenecker: “Generative Programming: Methods, Tools and Applications” Addison-Wesley Professional (2000).

[2] C. Szyperski: “Component Software – Beyond Object-Oriented Programming”, Addison-Wesley, 1997.

[3] I Crnkovic and M Larsson. Building Reliable Component-Based Software Systems. Artech House Publishers, 2002. ISBN 1-58053-327-2.

[4] P. López, J.M. Drake, and J.L. Medina: Real-Time Modelling of Distributed Component-Based Applications In: Proc. of 32h Euromicro Conference on Software Engineering and Advanced Applications, Croatia, August 2006.

[5] P. López, J.L. Medina y J.M. Drake: ” Análisis de la aplicación de la especificación de despliegue y configuración de OMG a sistemas de tiempo real basados en componentes” Jornadas de Tiempo Real, Valladolid, Febrero, 2006.

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

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

[8] P. López, J.M. Drake, and J.L. Medina: “Real-Time Extensions to "Deployment and Configuration of Component-based Distributed Applications”. Workshop on Distributed Object Computing for Real-time and Embedded Systems” Washinton, July, 2008.

[9] Proyecto THREAD (TIN 2005-08665-C03-02) Comisión Interministerial de Ciencia y Tecnología. (http://polaris.dit.upm.es/%7Estr/proyectos/thread/)

[10] P. López Martínez, José M. Drake, Pablo Pacheco, and Julio L. Medina: ” An Ada 2005 Technology for Distributed and Real-Time Component-based Applications” Lecture Notes on Computer Science, Springer, LNCS 5026, June, 2008, ISBN: 3-540-68621-7, pp. 254-267.

[11] P. López Martínez, José M. Drake, Pablo Pacheco, Julio L. Medina: “Ada-CCM: Component-based Technology for Distributed Real-Time Systems”. CBSE’08: Conference on Component-Based Software Engineering, , Karlsruher (Germany), 2008

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

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

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

IST project FRESCOR: Framework for Real-time Embedded Systems based on Contracts http://www.frescor.org.

[9] M. Henning y M. Spruiell: “Distributed Programming with ICE”. http://www.zeroc.com.

[Szi97] C. Szyperski: “Component Software – Beyond Object-Oriented Programming”, Addison-Wesley, 1997.

# 7. ANEXOS.

## *7.1 Anexo A. Especificación de la funcionalidad del generador de código del constructor (home) de un componente.*

### Herramienta para generar la clase que implementa la interfaz equivalente del *home* de un componente (*home wrapper*).

- Entradas y salidas:
  - Entrada(s):
    1. **Modelo de datos**: fichero con la descripción D&C del paquete del componente.
       Ej.: `GenericComponent.pcd.xml`
    2. **Diccionario de patrones**: fichero `HomeWrapperTemplate.txt`
    3. Nombre de la implementación del componente.
       Ej.: `GCImplementation0`
  - Salida(s):
    1. Código Java del *home wrapper* en el fichero `@implName@HomeWrapper.java`
       Ej.: `GCImplementation0HomeWrapper.java`

- Pasos del proceso de generación:
  1. **Apertura y análisis del modelo de datos**: Abrir, parsear y cerrar el fichero del paquete del componente.
  2. **Acceso al diccionario de patrones**: Abrir y leer el fichero de patrones.
  3. **Creación y apertura del fichero** que contendrá el código generado en la carpeta `javasrc/@implName@/@componentDomain@`
     Ej: `javasrc/GCImplementation0/domainGeneric/GCImplementation0HomeWrapper.java`
  4. **Rellenado del fichero de salida** según las siguientes reglas de generación:
     i. Escribir bloque HEAD (patrón **`<head>`**)

```
/**************************************************************************
*                          PROYECTO HESPERIA
*                 Grupo Computadores y Tiempo Real (CTR)
*                       UNIVERSIDAD DE CANTABRIA
*
* Description: Class that implements the equivalent interface of
*              the home
*
* @Author  @author@
* @Version @version@
**************************************************************************/
```

     ii. Escribir bloque IMPORT (patrón **`<imports>`**)

```
import Ice.*;
import ccm.*;
import @componentDomain@.*;
```
import correspondiente al dominio del componente

iii. Escribir el encabezamiento de la clase (contenido en el patrón **<headClass>**)

```
public class @implName@HomeWrapper implements @componentName@HomeOperations{
```

iv. Escribir el bloque ATTRIBUTES (patrón **<attributes>**)

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

v. Escribir bloque GRAL_COMPONENT_HOME_IMPLICIT_OPERATIONS (patrón **<homeOper>**)

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

   @implName@Wrapper theComp = new @implName@Wrapper();
   @componentName@Prx theCompPrx = @componentName@PrxHelper.checkedCast(theComp.basePrx);

   if(theCompPrx != null){
       return theCompPrx;
   }else{
       throw new ccm.CreateFailure();
   }
}
```

vi. Escribir bloque HOME_CONFIGURATION_OPERATIONS (patrón **<homeOper>**)

```
/*Method for setting a configuration for home object*/
public void setConfigurationValues(ConfigValue[] config, Ice.Current __current) {
   theConfigValues = config;
}

/*Method for determining if the configuration will be called*/
public void completeComponentConfiguration(boolean b, Ice.Current __current) {
}

/*Method for disabling the configuration of the home*/
public void disableHomeConfiguration(Ice.Current __current) {
   adapter.deactivate();
}
```

vii. Escribir bloque KEYLESS_CCM_OPERATIONS (patrón **<homeOper>**)

```
/*Method for creating a new component*/
public CCMObjectPrx createComponent(Ice.Current __current) throws CreateFailure {

@implName@Wrapper @implName@theComp = new @implName@Wrapper();
   theComp.setConfiguration(theConfigValues[0].name);
   CCMObjectPrx theCompPrx = ccm.CCMObjectPrxHelper.checkedCast(@implName@theComp.basePrx);

   if(theCompPrx != null){
       return theCompPrx;
   }else{
       throw new ccm.CreateFailure();
   }
}
```

viii. Escribir bloque CCM_HOME_OPERATIONS (patrón **<homeOper>**)

```
/*Method for removing the component*/
public void removeComponent(CCMObjectPrx comp, Current __current) throws RemoveFailure {

   if(ic != null){
       ic.destroy();
   }else{
       throw new RemoveFailure();
   }
}
```

ix. Escribir bloque MAIN (los args[] del "main" se leerán del fichero de despliegue) (patrón **<homeOper>**)

```
/*Main method for creating a home servant*/
public static void main(String[] args) {

@implName@HomeWrapper theHome = new @implName@HomeWrapper();
   ic = Ice.Util.initialize(args);
   adapter = ic.createObjectAdapterWithEndpoints("@implName@HomeWrapper", "default -p " +
args[0]);
   _@componentName@HomeTie servant = new _@componentName@HomeTie(theHome);
   adapter.add(servant, ic.stringToIdentity("@componentName@HomeWrapper"));
   adapter.activate();
   ic.waitForShutdown();
}
```

x. Escribir el cierre de la clase (patrón **<close>**)

```
<close>}
</close>
```

5. **Cerrar el fichero de salida**.

Código de salida:

```
/**********************************************************************
 *                    PROYECTO HESPERIA
 *             Grupo Computadores y Tiempo Real (CTR)
 *                  UNIVERSIDAD DE CANTABRIA
 *
 * Description: Class that implements the equivalent interface of
 *            the home
 * @Autor  Jose M Drake
 * @Version 26/02/08
 **********************************************************************/
import Ice.*;
import ccm.*;
import domainGeneric.*;
import domainZ.*;
import domainX.*;

public class GCImplementation0HomeWrapper implements _GenericComponentHomeOperations{

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

    /*Method for creating a new component*/
    public GenericComponentPrx create(Ice.Current __current) throws ccm.CreateFailure {
        GCImplementation0Wrapper theComp= new GCImplementation0Wrapper();
        return GenericComponentPrxHelper.checkedCast(theComp.basePrx);
    }
    /*Method for setting a configuration for home object*/
    public void setConfigurationValues(ConfigValue[] config, Ice.Current __current) {
         theConfigValues = config;
    }
    /*Method for determining if the configuration will be called*/
    public void completeComponentConfiguration(boolean b, Ice.Current __current) {
```

```
    }
    /*Method for disabling the configuration of the home*/
     public void disableHomeConfiguration(Ice.Current __current) {
          adapter.deactivate();
    }
    /*Method for creating a new component*/
     public CCMObjectPrx createComponent(Ice.Current __current) throws CreateFailure {
    GCImplementation0Wrapper theComp= new GCImplementation0Wrapper();
          theComp.setConfiguration(theConfigValues[0].name);
          return ccm.CCMObjectPrxHelper.checkedCast(theComp.basePrx);
    }
     /*Method for removing the component*/
          public void removeComponent(CCMObjectPrx comp, Current __current) throws RemoveFailure {
          adapter.destroy();
          ic.destroy();
    }
    /*Main method for creating a home servant*/
    public static void main(String[] args) {
          GCImplementation0HomeWrapper theHome =  new GCImplementation0HomeWrapper();
          ic = Ice.Util.initialize();
          Ice.ObjectAdapter adapter = ic.createObjectAdapterWithEndpoints("GCImplementation0HomeWrapper",args[0]);
          _GenericComponentHomeTie servant = new _GenericComponentHomeTie(theHome);
          adapter.add(servant, ic.stringToIdentity("GCImplementation0HomeWrapper"));
          adapter.activate();
          ic.waitForShutdown();
    }
}
```

## 7.2 Anexo B. Estructura y leyenda del modelo de datos utilizado (fichero `*.pcd.xml`)

- **Descripción D&C del paquete del componente.** Esqueleto de un fichero `*.pcd.xml`:

```
<DnCcdm:packageConfiguration>    => rootElement

    <basePackage> OR <importedPackage> OR <referencedPackage>  => topLevelElement

        <realizes>   => secondLevelElement

            <description UUID=""> OR <ref> OR <location>    => thirdLevelElement

                <!-- PORTS -->
                    <port name="" specificType="" supportedType="" kind=""/>

                <!-- PROPERTIES -->
                    <property name="" type="" label=""/>

                <!-- INFO PROPERTIES -->
                    <infoProperty name="author">
                        <value>Jose M Drake</value>
                    </infoProperty>

            </description>
        </realizes>

        <!-- IMPLEMENTATION -->
        <implementation name=""/>
        …….

        <!-- CONFIG PROPERTIES, no interesan -->
        <configProperty/>
        …….

        <!-- INFO PROPERTIES, no interesan -->
        <infoProperty/>
        ……
    </basePackage>


    <!-- SELECT REQUIREMENTS, no interesan -->
    <selectRequirement/>
    …….

    <!-- CONFIG PROPERTIES, no interesan -->
    <configProperty/>
    …….

</DnCcdm:packageConfiguration>
```

**Leyenda de términos utilizados cuya info encontramos en el `*.pcd.xml`:**

- **Faceta**: Cada uno de los elementos <port> con atributo kind = `FACET` y atributo name != supports.
- **Receptáculo**: Cada uno de los elementos <port> con atributo kind = `*RECEPTACLE` (`SIMPLEXRECEPTACLE` o `MULTIPLEXRECEPTACLE`)
- **Activación**: Cada uno de los elementos <port> con atributo kind = `*ACTIVATION` (`ONESHOTACTIVATION` o `PERIODICACTIVATION`)
- **Propiedades**: Cada uno de los elementos <property>

- **Leyenda de parámetros en las plantillas:**
  - Respecto al elemento **<description>:**
    - `@componentDomain@` = penúltimo campo del atributo UUID.
    - `@componentName@` = último campo del atributo UUID.

      UUID = "http://ctr.unican.es/generic/components/**domainGeneric/GenericComponent**"
    - `@propietaryClassName@` = `CCM_@componentName@`.
    - `@homeName@` = `@componentName@_home`.
  - Respecto al elemento **<implementation>:**
    - `@implName@` = valor del atributo name.
  - Respecto a elementos **<infoProperty>**:
    - `@author@` = nudo de texto dentro del <value> hijo del <infoProperty> cuyo atributo name="author".
    - `@version@`= nudo de texto dentro del <value> hijo del <infoProperty> cuyo atributo name="version".
  - Respecto a **facetas**:
    - `@facetNum@` = nº de ellas.
    - `@facetName@` = valor del atributo name.
    - `@facetDomain@` = penúltimo campo del atributo specificType.
    - `@facetInterface@` = último campo del atributo specificType.

      specificType = "interfaces/domainX/FileGeneric1.ice::**domainX**::**InterfaceA**"
  - Respecto a **receptáculos**:
    - `@recepNum@` = nº de ellos.
    - `@recepName@` = valor del atributo name.

- `@recepDomain@` = penúltimo campo del atributo specificType.
- `@recepInterface@` = último campo del atributo specificType.

specificType = "interfaces/domainZ/FileGeneric3.ice::**domainZ**::**InterfaceC**"

- Respecto a **activaciones**:
  - `@portName@` = valor del atributo name.
  - `@interfaceName@` = valor del atributo specificType.

- Respecto a **propiedades**:
  - `@propertyName@` = valor del atributo name.

- Respecto a … :
  - `@supportedInterface@` = último campo del atributo specificType cuyo name =”supports” y kind = FACET.
  - `@supportedDomain@` = penúltimo campo del atributo specificType de elementos cuyo name = “supports” y kind = FACET.