Computación y Sistemas Vol. 3 No. 1 pp. 38-49
© 1999, CIC - IPN. *ISSN 1405-5546* Impreso en México

Generación Automática de Prototipos en Entornos Internet/Intranet a Partir de Modelos Conceptuales OO

Oscar Pastor, Vicente Pelechano, Emilio Insfrán
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
46071 Valencia (SPAIN)
email: [opastor,pele,einsfran]@dsic.upv.es

Jaime Gómez
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Alicante
03690 Alicante (SPAIN)
email: jgomez@dlsi.ua.es

Artículo recibido el 2 de abril, 1998; aceptado el 11 de julio, 1999

Resumen

Este artículo presenta un proceso de generación automática de código en entornos Internet/Intranet, Java en particular. OO-Method (Pastor et al., *1997) es la metodología OO que da soporte a este proceso. Esta metodología está basada en OASIS (Pastor* et al., *1992; Pastor & Ramos, 1995), un lenguaje de especificación formal y orientado a objetos. El punto de partida del proceso de generación es el modelo conceptual obtenido en la fase de análisis de OO-Method. Un modelo de ejecución preciso establece una representación del modelo conceptual en el entorno de desarrollo elegido (atendiendo aspectos estáticos y dinámicos), y siguiendo la estrategia de ejecución subyacente a dicho modelo de ejecución, se obtiene de forma automática una implementación completa del sistema modelado. El resultado final es una aplicación Web funcionalmente equivalente a la especificación del sistema, con una arquitectura de tres capas, implementada en Java, que utiliza una base de datos relacional como repositorio de objetos.*

Palabras Clave:
Modelización conceptual, Lenguajes de especificación, Generación de código, Tecnologías Internet/Intranet.

1 Introducción

Un problema clásico en la ingeniería del software es cómo derivar software ejecutable a partir de los requisitos de un sistema y cómo podría sistematizarse este proceso. La aproximación orientada a objetos proporciona un modelo común durante todo el proceso de desarrollo (desde el análisis a la implementación) permitiendo una suave transición entre cada una de estas fases.

En este contexto, a lo largo de estos años han surgido varios métodos de modelado conceptual. De entre estos métodos, OMT (Rumbaugh *et al.*, 1991), OOSE (Jacobson *et al.*, 1992), Booch (1991), y ahora el lenguaje de modelado UML (Booch *et al.*, 1997), son los que están soportados por la mayoría de herramientas CASE del mercado como Rational ROSE/Java (Rational-Software, 1995) o Paradigm Plus (Platinum-Technology, 1997). Estas herramientas intentan resolver la problemática asociada al desarrollo de sistemas, incluyendo 'generación de código' a partir de los modelos de análisis. Sin embargo, si examinamos más a fondo el código generado encontraremos que no se obtiene un producto funcionalmente equivalente a la descripción capturada en el modelo conceptual, lo que se obtiene en realidad no es más que un conjunto de plantillas para la declaración de las clases donde no se implementa ningún método.

En este artículo se presenta un proceso de generación automática de código propuesto por OO-Method, un método basado en un modelo de objetos formal. Este proceso constituye una solución operativa al problema de la generación de código funcionalmente equivalente a

la especificación de un sistema de información.

La característica principal de este método es que los esfuerzos de los desarrolladores se centran en la etapa de modelado conceptual, y a continuación de forma automática se obtiene una implementación completa del sistema, siguiendo un modelo de ejecución preciso (que incluye estructura y comportamiento).

El resultado final es, en este caso, una aplicación Web con una arquitectura de tres capas, implementada en Java sobre una base de datos relacional como repositorio de objetos.

La herramienta OO-Method/CASE[1], da soporte a este método y constituye una aproximación operacional a las ideas del paradigma de programación automática (Blazer *et al.*, 1983): recolección de las propiedades del sistema de información, generación automática de una especificación formal del sistema y obtención de un prototipo de software funcionalmente equivalente a la especificación del sistema.

El contenido del artículo se estructura de la siguiente manera: en el punto 2 se comentan brevemente las características principales del lenguaje de especificación OASIS y se presenta OO-Method, describiendo los modelos que permiten especificar el sistema gráficamente, y el modelo de ejecución que se utiliza como patrón en el proceso de generación de código. En el punto 3, se explican las características de la arquitectura de un prototipo generado en Java. Para comprender mejor este proceso de traducción, en el punto 4 se desarrolla un caso de estudio relativo al Servicio de Mantenimiento de un Hospital. Finalmente, se presentan las conclusiones y las líneas futuras de actuación que derivan de este trabajo.

2 OO-Method y OASIS

OO-Method es una metodología de producción automática de software basada en técnicas de especificación formales, que usa diagramas gráficos similares a los empleados tradicionalmente por las metodologías convencionales (OMT, Booch, UML). El proceso de producción de software empieza con la fase de modelado conceptual, donde se recogen con modelos gráficos las propiedades relevantes del sistema a desarrollar. Una vez que tenemos la descripción del sistema, se obtiene de forma automática una especificación formal y OO del sistema. Esta especificación es la fuente de un modelo de ejecución que determina de forma automática las características del sistema dependientes de la implementación (interfaz de usuario, control de acceso, activación de servicios, consulta y manipulación de información). Esta propuesta proporciona un marco bien definido que nos permite construir herramientas de modelado y generación de código, basadas en el paradigma de programación automática.

2.1 OASIS: Un Modelo Formal Orientado a Objetos

OO-Method se ha creado sobre las bases de OASIS, un lenguaje formal y OO de especificación de sistemas de información. Vamos a dar un breve repaso de las principales características de OASIS. En una especificación OASIS, una clase es un patrón o conjunto de propiedades que comparten todos sus ejemplares. Este patrón tiene un nombre y permite la declaración de un mecanismo de identificación. La signatura de la clase incluye atributos, eventos y un conjunto de fórmulas de la lógica dinámica que nos permiten describir el resto de sus propiedades:

- restricciones de integridad estáticas y dinámicas.
- evaluaciones que especifican cómo cambian los atributos debido a la ocurrencia de eventos.
- derivaciones que especifican el valor de un atributo derivado en función de otros.
- precondiciones que establecen condiciones que deben satisfacerse para activar un evento.
- disparos que provocan la activación espontánea de un evento como consecuencia de la satisfacción de una condición.

Además, un objeto puede definirse como un proceso observable, por lo que la especificación de la clase se enriquece con un álgebra de procesos básica (Pastor & Ramos, 1995) que permite declarar las vidas posibles de un objeto como términos de este álgebra, cuyos elementos son eventos y transacciones, combinados con operadores de alternativa y secuencia de procesos.

Por razones de espacio, la descripción del lenguaje de especificación es introductoria. El lector puede encontrar una descripción mas detallada de OASIS en (Pastor *et al.*, 1992), y la descripción completa de su semántica y fundamentos formales en (Pastor & Ramos, 1995).

2.2 Modelado Conceptual

Los conceptos formales asociados a OASIS determinan la información relevante a capturar en la fase de modelado conceptual. Para ello, OO-Method proporciona tres modelos con los que recoger las propiedades relevantes de un sistema de información. Los diagramas asociados a cada

[1]El desarrollo de la herramienta OO-Method/CASE ha sido parcialmente subvencionada por el IMPIVA a través de un proyecto del Plan de Ciencia y Tecnología Valenciano.

uno de esos tres modelos respetan la notación gráfica habitual en contextos de análisis, aunque su semántica está concebida para declarar con precisión sólo aquel conjunto de información que es realmente necesaria para describir el Sistema. Como decíamos, ese conjunto queda determinado por las secciones de especificación de una clase en OASIS.

- modelo de Objetos: es un modelo gráfico en el cual se definen las clases del sistema (incluyendo atributos, servicios y relaciones entre clases). Las relaciones pueden ser de agregación (parte-de), especialización (es-un), generalización y de agente (introducidas para especificar qué objetos pueden activar los servicios ofrecidos por cada clase).

- modelo Dinámico: es un modelo gráfico que permite especificar las vidas válidas de los objetos de una clase y las interacciones entre objetos. Para ello se utilizan dos tipos de diagramas:

  - Diagramas de Transición de Estados: se utilizan para describir genéricamente la secuencia correcta de eventos en la vida de los objetos de una clase.

  - Diagramas de Interacción entre objetos: se utilizan para representar las interacciones entre los objetos del sistema. En este diagrama se definen dos mecanismos básicos de interacción: los disparos (servicios que se activan de forma automática como consecuencia del cumplimiento de una condición sobre el estado de un objeto) e interacciones globales (transacciones que involucran a servicios de diferentes objetos).

- modelo Funcional: es un modelo que se utiliza para capturar la semántica asociada al cambio de estado de un objeto como consecuencia de la ocurrencia de un evento. El efecto de los eventos sobre el estado de los objetos se corresponde con una evaluación de la lógica dinámica, que se especifica siguiendo una estrategia de clasificación de los atributos en categorías (definidas en (Pastor *et al.*, 1997)) y les asigna nuevos valores en función de los eventos ocurridos. Ésta es una aportación interesante de este método, que nos facilita la generación de forma automática de una especificación completa en OASIS. Finalmente, a partir de la información recogida en estos tres modelos, se obtiene, utilizando una estrategia de traducción bien definida, una especificación formal y OO en OASIS que constituye un repositorio de alto nivel del sistema.

2.3 El Modelo de Ejecución

Una vez especificado el sistema, un modelo de ejecución establece una representación del modelo conceptual en el entorno de desarrollo elegido (atendiendo a aspectos estáticos y dinámicos), y una estrategia de ejecución asociada.

2.3.1 Estrategia de Generación de Código

A continuación se explica, el patrón genérico y los componentes software a utilizar para implementar en un entorno de desarrollo las propiedades del sistema utilizando una arquitectura lógica de tres niveles (three-tier):

- Nivel de interfaz: en este nivel se sitúan los componentes que implementan la interacción con los usuarios finales, mostrando una representación visual de los datos y los servicios que ofrecen los objetos del sistema.

- Nivel de aplicación: en este nivel se sitúan los componentes que implementan de forma completa el comportamiento de las clases especificadas en la fase de modelado conceptual.

- Nivel de persistencia: en este nivel se sitúan los componentes que proporcionan los servicios necesarios para dar persistencia a los objetos del nivel de aplicación. La persistencia de los objetos actualmente se realiza recurriendo a un sistema gestor de bases de datos relacional (SGBDR).

2.3.2 Estrategia de Ejecución

Para animar el sistema especificado, se define una estrategia de ejecución e interacción. Esta estrategia es cercana a las técnicas de realidad virtual, en el sentido de que un objeto activo[2] se introduce en la sociedad de objetos como miembro de ella e interactúa con los demás enviando y recibiendo mensajes. Para iniciar una sesión de ejecución, los pasos a seguir son:

1. Identificación del usuario (control de acceso): consiste en la conexión del usuario al sistema. Una vez conectado, se le proporciona una visión clara de la sociedad de objetos (ofreciéndole qué clases de objetos puede ver, los servicios que puede activar y los atributos que puede consultar).

2. Activación de servicios: el usuario podrá activar cualquier servicio (evento o transacción) disponible en su visión de la sociedad. Además, podrá realizar observaciones del sistema (object queries).

[2] Un objeto activo es una instancia de una clase que actúa como agente de los servicios de alguna clase.

Las clases que implementan las tareas de control de acceso y construcción de la vista del sistema (clases y servicios visibles) se implementarán en el nivel de interfaz. La información necesaria para configurar la vista del sistema está incluida en la especificación del sistema (relaciones de agente) obtenida en la fase de modelado conceptual. Cualquier activación de un servicio tiene dos partes: la construcción del mensaje y su ejecución (si es posible). El algoritmo de activación de un servicio sigue los siguientes pasos:

(a) Identificación del objeto servidor: si el objeto existe, el nivel de persistencia se encargará de recuperar el objeto servidor de la base de datos y si el servicio solicitado es un evento de creación reservará espacio para su almacenamiento.

(b) Introducir los argumentos necesarios para la ejecución del evento: el nivel de interfaz preguntará por los argumentos del evento que va a activarse (sí es necesario).

Una vez el mensaje se ha enviado, se identifica el objeto servidor[3], y se procede a seguir la siguiente secuencia de acciones sobre dicho objeto:

1. Transición válida de estado: se verifica en el diagrama de transición de estados que exista una transición válida para el servicio seleccionado.
2. Satisfacción de precondición: se comprueba que se cumpla la precondición asociada al servicio en ejecución (si existe).

   Si no se cumplen 1 y 2 se generará una excepción informando que el servicio no puede ejecutarse.
3. Evaluaciones: se modifican los valores de los atributos afectados por la ocurrencia del servicio (como se especificó en el modelo funcional).
4. Comprobación de las restricciones de integridad: las evaluaciones del servicio deben dejar al objeto en un estado válido. Se comprueba que no se violan las restricciones de integridad (estáticas y dinámicas). Si alguna de ellas se viola, se generará una excepción y el cambio de estado producido se ignorará.
5. Comprobación de las relaciones de disparo: después de un cambio de estado válido, y como acción final, se debe verificar el conjunto de reglas condición-acción que representa la actividad interna del sistema. Si alguna de ellas se cumple, se activará el servicio correspondiente.

[3]La existencia del objeto servidor es una condición implícita para ejecutar cualquier evento, excepto si se trata del evento de creación.

Una vez finalizadas con éxito las acciones precedentes, los componentes de la capa de persistencia se encargan de actualizar (UPDATE) la BDR correspondiente.

Los pasos anteriores guiarán la implementación de cualquier aplicación para asegurar la equivalencia funcional entre la descripción del sistema recogida en el modelo conceptual y su reificación en un entorno de programación de acuerdo con el modelo de ejecución.

3 Arquitectura del Prototipo Generado en Java

El modelo abstracto de ejecución mostrado anteriormente está orientado a la generación de prototipos con arquitectura de tres capas. A continuación, se presenta una implementación concreta utilizando tecnología WEB y Java como lenguaje de programación.

3.1 Traducción de un Modelo Conceptual a Clases Java

En este apartado, se presentan las clases que implementan las capas de interfaz, aplicación y persistencia. En la **capa de interfaz** están las clases que implementan la interacción entre el usuario y la aplicación (interfaz gráfica de usuario):

- Clase Control_de_Acceso: se implementa como un applet que contiene los típicos widgets que permiten al usuario su identificación como miembro de la sociedad de objetos (proporcionando su identificador, password y la clase a la cual pertenece).

  Este applet se crea cada vez que el usuario quiere conectarse al sistema (implementando el primer paso del modelo de ejecución).

  La declaración de esta clase es la siguiente:

```
import java.awt.*;
import java.applet.*;
public class Control_Acceso
extends Applet {...}
```

- Clase Vista_del_Sistema: cuando un usuario está conectado al sistema un objeto de la clase Vista_del_Sistema mostrará un frame con un menú bar asociado que tendrá tantos submenús como clases el objeto puede interactuar. Cada submenú tendrá tantos items como servicios de dicha clase el usuario puede activar. La declaración de esta clase es la siguiente:

```
import java.awt.*;
public class Vista_Sistema
extends Frame {...}
```

- Clase Activación_de_Servicio: esta clase definirá el interfaz típico de WWW para entrada de datos: una caja de diálogo donde se solicitan los argumentos relevantes del servicio. El objeto Activación_de_Servicio será genérico para todos los servicios de todas las clases, y dependiendo del servicio seleccionado mostrará las correspondientes etiquetas con sus celdas de edición. Una vez que hayan sido rellenadas, el usuario enviará el mensaje al destinatario o ignorará la acción de petición de datos. Su declaración es:

```
import java.awt.*;
public class Activacion_Servicio
extends Dialog {...}
```

En la **capa de aplicación** tendremos las clases que implementan el comportamiento de las clases especificadas en el modelo conceptual (clases del dominio). Para garantizar que los objetos de estas clases se comportan como objetos OASIS (objetos que siguen el modelo de objetos propuesto en OASIS) necesitamos de un mecanismo que permita a estas clases implementar los métodos necesarios para soportar el algoritmo de ejecución de servicios del modelo de ejecución, tal y como se comentó en el punto 2.3.2. Este mecanismo se implementa en Java como un interface (que denominamos OASIS) y que proporciona las plantillas de los métodos que obligatoriamente deben implementarse en las clases del dominio. El interfaz tendrá la siguiente estructura:

```
import java.awt.*;
interface OASIS {
void Check_T_Estad  (string event);
void Check_Precond  (string event);
void Check_Restricc ();
void Check_Disparos ();
}
```

Estos métodos se corresponderán respectivamente con los pasos 1, 2, 4 y 5 del algoritmo de ejecución de servicios del modelo de ejecución.

La **capa de persistencia** estará integrada por la base de datos, es decir, por las tablas relacionales que representan las clases del diagrama de clases y sus relaciones (herencia y agregación). Dicho de otro modo, cada clase del diagrama de clases se representará como una tabla cuyas tuplas serán instancias de esa clase. La implementación de las relaciones (agregación y herencia) se realiza conforme a (Letelier & Ramos, 1995). Además en esta capa se implementará la clase GestorPersistencia, que proporcionará los métodos para salvar, borrar y recuperar objetos de la base de datos. La clase GestorPersistencia tiene la siguiente estructura genérica:

```
class GestorPersistencia extends Object {
void borrar();
void salvar();
void recuperar();
}
```

Las clases JDBC de Java se utilizarán en la implementación de estos métodos para acceder de forma sencilla a los servidores relacionales que almacenen el repositorio de sistema. Finalmente, para permitir que las clases del dominio sean persistentes. Éstas heredarán de la clase GestorPersistencia mediante la claúsula extends en la declaración de la clase, y redefinirán para cada una de ellas los métodos correspondientes.

Es importante resaltar que aunque en el artículo nos centremos en Java, este diseño permitirá traducir a cualquier otro lenguaje de programación distribuyendo adecuadamente los componentes de la aplicación, dependiendo de las características y necesidades del entorno destino.

3.2 Distribución de Clases Java en una Arquitectura WWW

Una vez que todas las clases han sido generadas, se distribuyen en una arquitectura en tres capas como se ilustra en la figura 1.

Un navegador Web descargará la página HTML de la aplicación desde un servidor Web, junto con el applet que conforma el interfaz de la aplicación. El usuario interactuará con los objetos Java del sistema a través del cliente Web. Los objetos Java del sistema se almacenarán en el servidor Web de la aplicación que consultará o actualizará el estado de los objetos almacenados en el servidor de datos a través de los servicios proporcionados por la clase GestorPersistencia. Los componentes de esta arquitectura se generarán de forma automática usando la herramienta OO-Method/CASE.

4 Caso de Estudio

La herramienta OO-Method/CASE nos permite modelar y generar automáticamente prototipos Java funcionalmente equivalentes al modelo Conceptual construido. Para comprender mejor la arquitectura y el funcionamiento de las clases Java generadas, presentamos como caso de estudio el Servicio de Mantenimiento de un Hospital:

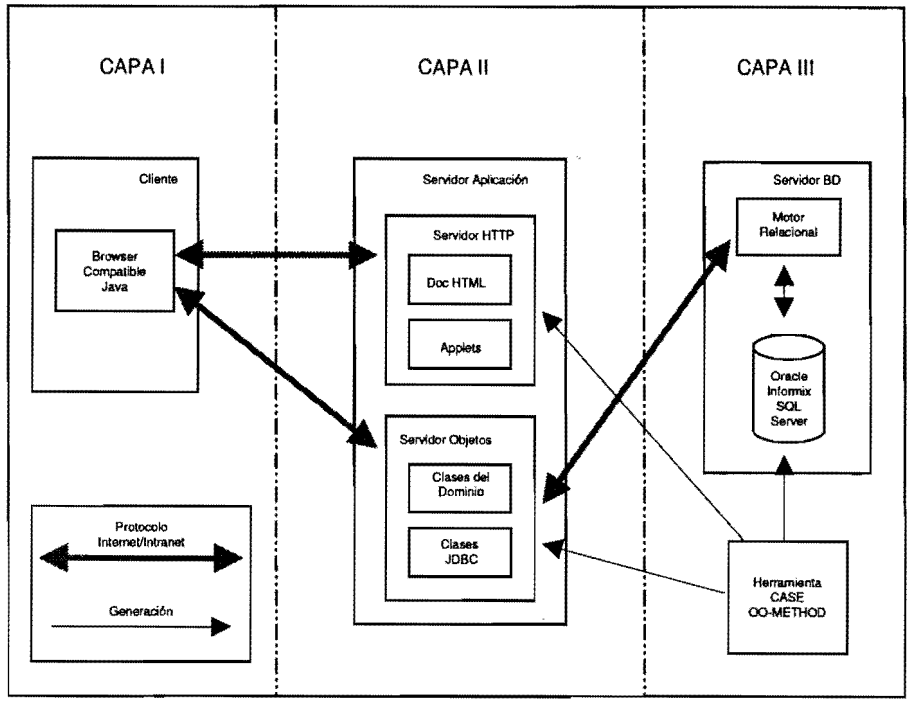

Figura 1: *Arquitectura WWW de tres capas*

"El servicio de mantenimiento de un hospital gestiona las averías que se producen en él. El servicio está formado por 1 responsable, 7 maestros encargados de cada una de las áreas del servicio y unos cuantos operarios asignados a cada área. Diariamente el responsable clasifica los partes de averías que se reciben, en función del área a asignar y la urgencia que precise. Cada maestro recibe los partes de avería correspondientes a su área y procede a la resolución de la avería asignado los trabajos a sus operarios. A veces ocurre que las averías se producen sobre maquinaria muy específica que el servicio de mantenimiento no es capaz de atender (ascensores, rayos-X, ...). En estos casos, el maestro debe de comprobar si la máquina tiene contrato de mantenimiento con alguna empresa externa y, en caso afirmativo, generar un aviso de avería. Respecto a las máquinas, debemos tener en cuenta que:

- cuando se da de alta una máquina en el servicio, no se le asocia ningún contrato de mantenimiento.
- sólo se podrá asignar un contrato de mantenimiento a una máquina si ésta no tiene ningún contrato anterior en vigor.
- para cualquier máquina con contrato, se puede modificar y/o cancelar su contrato de mantenimiento.
- el responsable es el único que puede dar de alta una máquina.
- el responsable es el único que puede asignar, modificar o cancelar un contrato de mantenimiento entre una máquina y una empresa mantenedora."

A partir de esta especificación construimos el Modelo Conceptual (modelo de objetos, dinámico y funcional) identificando las clases, las relaciones entre éstas, y especificando su comportamiento. Vamos a ilustrar estas tareas paso a paso.

4.1 Modelo de Objetos

Vamos a centrarnos en la especificación que afecta a los contratos de mantenimiento. En la figura 2 podemos ver la porción del diagrama de clases que se corresponde con esta especificación. La clase contrato_mantenimiento es una agregación que tiene como componentes la clase máquina y la clase empresa_mantenedora.

Además podemos observar la relación de agente asociada a la clase máquina, de ella podemos deducir que la clase responsable es agente de los eventos de la clase máquina enlazados a través de las líneas punteadas. Fijémonos que a partir de las relaciones de agente podemos obtener la vista del sistema que le corresponde a cualquier objeto instanciado de una clase.

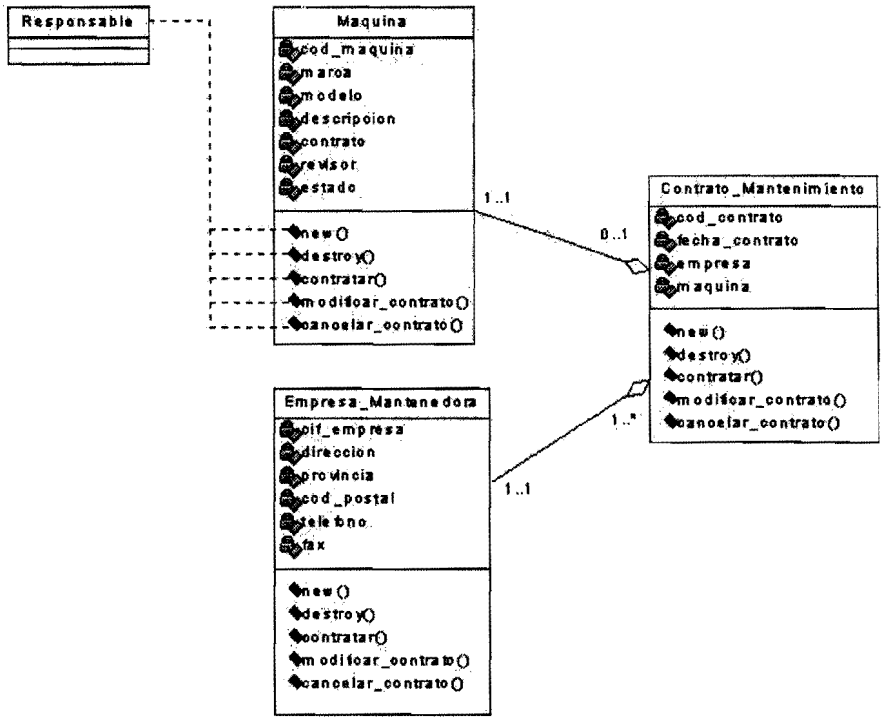

Figura 2: *Diagrama de clases del sistema*

4.2 Modelo Dinámico

Para cada una de las clases obtenidas en el paso anterior, especificamos un diagrama de transición de estados. La figura 3 refleja el diagrama de transición de estados para la clase máquina del diagrama anterior. Podemos observar como la creación de un objeto de esta clase viene caracterizada por la acción RESP:new que implica la activación del evento de creación por parte del responsable, produciendo la transición al estado Maquina0. A partir del estado Maquina0, se pueden producir dos cambios de estado (Maquina1 o Destrucción), caracterizados nuevamente por las acciones asociadas a las transiciones entre estos estados. Además, las acciones pueden ser enriquecidas con información de precondiciones que han de cumplirse para llevar a cabo una acción.

Además, en el modelo dinámico, con el diagrama de interacción de objetos se pueden representar las interacciones entre objetos en términos de disparos y transacciones globales.

4.3 Modelo Funcional

Finalmente con el modelo funcional especificamos el efecto que tienen las operaciones sobre el estado del objeto como respuesta a los eventos, indicando cómo cambia su estado. Cada uno de los atributos variables de la clase tendrá una entrada en el modelo funcional que especificará como cambia su valor en función del evento que se ejecute. La figura 4, representa la especificación del atributo estado de la clase Máquina en el modelo funcional. Este atributo se categoriza como de 'pertenencia a una situacion', lo que indica que toma su valor en un dominio limitado. Más concretamente, si la acción que se produce es RESP:contratar y el atributo estado valía 'SIN_CONTRATO', su nuevo valor pasará a ser 'CON_CONTRATO'.

```
CLASE: Máquina  ATRIBUTO: estado
CATEGORIA: pertenencia a una situación
-----------------------------------------
Antes          Acción          Después
-----------------------------------------
SIN_CONTRATO   RESP:contratar CON_CONTRATO
CON_CONTRATO   RESP:cancelar  SIN_CONTRATO
-----------------------------------------
```

Figura 4: *Modelo funcional del atributo* estado *de la clase* máquina

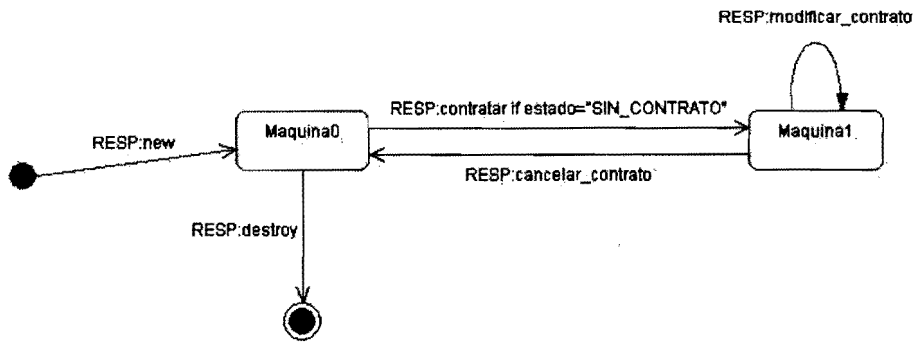

Figura 3: *DTE para la clase máquina*

4.4 Modelo de Ejecución

Una vez especificado el sistema, el modelo de ejecución establece la representación del modelo conceptual en el entorno de desarrollo elegido (en este caso Java), según lo explicado en el punto 3. Como resultado del proceso de generación de código definido en el modelo de ejecución, obtendremos la arquitectura de la aplicación para un entorno Web de forma automática. Como se comentó en el punto 3.1, esta arquitectura se organiza en tres capas que incluyen:

4.4.1 Capa de Interfaz

La capa de interfaz estará formada por las clases Java que implementan el interfaz de usuario según el modelo de ejecución.

Estas clases se estructuran como sigue:

- Control_Acceso.java: se obtiene la información que se necesita para implementar esta clase, consultando las clases del dominio que aparecen en el diagrama de clases. En el ejemplo son: responsable, contrato_mantenimiento, máquina y empresa_mantenedora.
- Vista_Sistema.java: se obtiene la información que se necesita para implementar esta clase consultando del diagrama de clases las relaciones de agente, las clases del dominio y los eventos; y del diagrama de interacción de objetos las transacciones globales. En el ejemplo, la vista del sistema para la clase responsable viene caracterizada por sus relaciones de agente, se corresponde con: la clase máquina y con ella sus eventos new, destroy, contratar, modificar_contrato y cancelar_contrato.
- Activacion_Servicio.java: se obtiene la información que se necesita para implementar esta clase, consultando del diagrama de clases los atributos necesarios para ejecutar un servicio. En el ejemplo, para ejecutar el servicio contratar necesitamos el cod_referencia de la clase máquina y el cif_empresa de la clase empresa_ mantenedora.

4.4.2 Capa de Aplicación

La capa de aplicación estará formada por clases Java que implementan las clases del dominio del problema. Examinando esta plantilla (ver abajo), podemos observar que toda clase deriva de GestorPersistencia heredando los métodos que se necesitan para asegurar la persistencia de los objetos de la clase. Además, implementa el Interface Oasis, unas líneas más adelante entenderemos que quiere decir esto.

```
Public class <NombreClase> extends,
<NombreGestorPersistencia>
implements <NombreInterfaceOasis>; {
<TipoAtributo>  <Atributo>;

// servicios de la clase

protected void <NombreEvento>
(<Parametros> [] x);
protected void <NombreTransaccion>
(<Parametros> [] x);

// servicios para implementar el
// modelo de ejecucion

protected void <NombreCheckTransEst>
(String <NombreEvento>) throws
<NombreExcepcionTransEst>;
protected void <NombreCheckPrecond>
(String <NombreEvento>)throws
<NombreExcepcionPrecond>;
protected void <NombreCheckRestricc>
(String <NombreEvento>) throws
<NombreExcepcionRestricc>;
protected void <NombreCheckDisparos>
(String <NombreEvento>)
throws <NombreExcepcionDisparos>;
public void <ActivarNombreEvento>
```

```
(<Parametros> [] x);
public void <ActivarNombreTransaccion>
(<Parametros> [] x);
}
```

A continuación se declaran los atributos y se implementan los métodos que se corresponden con los servicios de la clase (eventos y transacciones). Más adelante, se implementa el Interface Oasis, es decir, se implementan los métodos necesarios para satisfacer el modelo de ejecución. Estos métodos se corresponden con los servicios para: chequear las precondiciones, la transición de estados, las restricciones y los disparos. Finalmente se implementan los métodos que se necesitan para la activación de los servicios de la clase, habrá un método de este tipo para cada uno de los servicios de la clase (eventos y transacciones).

La visibilidad de los métodos que se corresponden con los servicios de clase y la de los servicios que implementan el Interfaz Oasis es protected (es decir, sólo son visibles para los objetos de la clase y sus clases descendientes). Sin embargo, la visibilidad de los métodos encargados de la activación de los servicios de la clase (<ActivarNombreEvento> si es un evento o <ActivarNombreTransaccion> si es una transacción) es public para permitir que puedan ser solicitados desde objetos instanciados de otras clases. Más concretamente, estos métodos serán solicitados por los agentes válidos.

Es importante resaltar el hecho de que toda la información que se necesita para implementar este patrón de diseño se obtiene a partir de los distintos diagramas del modelo conceptual, lo que asegura el proceso de generación automática. Los patrones de diseño para implementar las precondiciones, transición de estados, restricciones y disparos, no se comentan por razones de espacio. Siguiendo esta plantilla de diseño se genera automáticamente el código asociado a las clases del dominio. A continuación, se muestra resumido el código Java que implementa la clase del dominio máquina.

```
Package capa_aplicación;
import Gestor_Persistencia;
import Oasis.*;
import excepciones.*;

public class Maquina extends
Gestor_Persistencia implements Oasis;
{
// atributos de la clase
String cod_maquina; // identificador
String estado;
String marca;
String modelo;
String descripcion;
String contrato;
String revisor;

// Atributo que almacena el estado del
// DTE en el que se encuentra el objeto
String estado_dte;
// Constructor de la clase
public void Maquina
(Object[] _parametros) {...}

// Eventos que implementan las
// evaluaciones
protected boolean Contratar
(string empresamnto)
{
    estado = "CON_CONTRATO";
    revisor = empresamnto;
    }
...
// Eventos para soportar la estrategia
// de ejecución
protected void Check_Trans_Estados
(string event)throws EX_Trans_Estados
{...}
protected void Check_Precondiciones
(string event) throws EX_Precond {...}
protected void Check_Restricciones()
    throws EX_Restricciones {...}
protected void Check_Disparos()
throws EX_Disparos {...}

// Métodos encargados de la activación
// de servicios ofertados
public void Activar_Serv_Contratar
(String p_cod_maquina,
String p_empresamnto) {
    try {
    Recuperar(p_cod_maquina);
    Check_Precondiciones("Contratar");
    Check_Trans_Estados("Contratar");
    Contratar(empresamnto);
    Check_Restricciones();
    Check_Disparos();
    Salvar();
    }
    catch (EX_Precond e) {...};
    catch (EX_Trans_Estados e) {...};
    catch (EX_Restr_Integridad e) {...};
    catch (EX_Disparos e) {...};
}
...
} // fin clase máquina
```

Podemos observar como el código generado se corresponde con el patrón de diseño. Primero tenemos los atributos de la máquina, luego los métodos correspondientes con los servicios de la clase que en este caso son Máquina (método constructor de la clase) y Contratar. A continuación los métodos que implementan el Interfaz Oa-

sis que son Check_Trans_Estados, Check_Precondiciones, Check_Restricciones, Check_Disparos, y finalmente, el método que implementa la activación del servicio contratar que sigue el algoritmo de ejecución de eventos tal y como se explicó en el punto 2.3.2.

4.4.3 Capa de Persistencia

La capa de persistencia estará integrada por la base de datos, es decir, por las tablas relacionales que representan las clases del diagrama de objetos y sus relaciones (herencia y agregación). En el ejemplo, tendremos tablas cuyas tuplas representarán objetos de las clases: máquina, responsable, empresa_mantenedora y contrato_mantenimiento. Además, en esta capa se implementará la clase GestorPersistencia que proporciona los métodos para salvar, borrar y recuperar objetos de la base de datos tal y como se explicó en el punto 3.1.

4.4.4 Ilustración del Ejemplo

A continuación, se describe un escenario ilustrativo para el prototipo generado.

Cuando un cliente carga la página principal (HTML) de la aplicación, se descarga un applet que instancia un objeto de la clase Control_Acceso. Este objeto pide la identificación del usuario tal y como se puede apreciar en la figura 5. Una vez que el usuario es autentificado, se instancia un objeto de la clase Vista_del_Sistema abriéndose un frame que contiene la vista de la sociedad de objetos (servicios a los cuales dicho usuario tiene permiso).

La barra de menú del frame contiene como ítems las clases visibles para el usuario conectado.

A cada una de estas clases se le asocia un menú desplegable que tiene tantos ítems como servicios de esa clase a los que el usuario tiene permiso de ejecución. La figura 6 muestra esta situación.

Cuando el usuario selecciona alguno de estos servicios, se invoca un método de la clase correspondiente (<ActivarNombreEvento> si es un evento o <ActivarNombreTransaccion> si es una transacción) desplegando una caja de diálogo (ver figura 7) para la introducción de los parámetros necesarios para proceder a la activación de ese servicio.

El botón OK de esta caja de diálogo, tiene asociado el código necesario para llamar al método de la clase que implementa el efecto de la ejecución de ese evento. Este método comprobará que existe una transición válida desde el estado actual e intentará ver si se satisfacen las precondiciones. Si se tiene éxito, el cambio de estado del objeto se llevará a cabo de acuerdo a la especificación del modelo funcional.

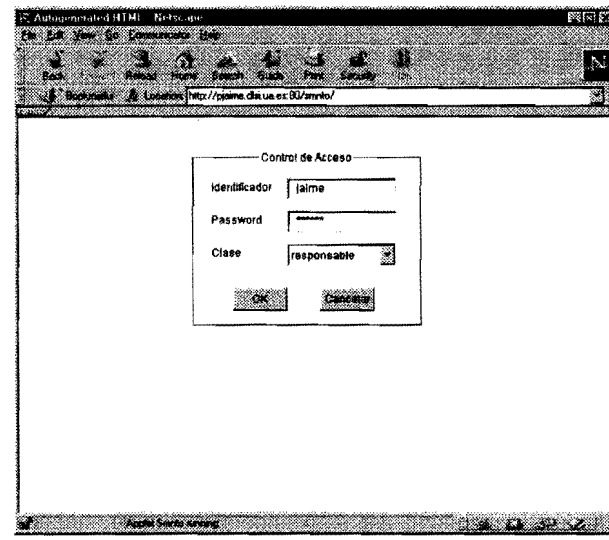

Figura 5: *Control del acceso al sistema*

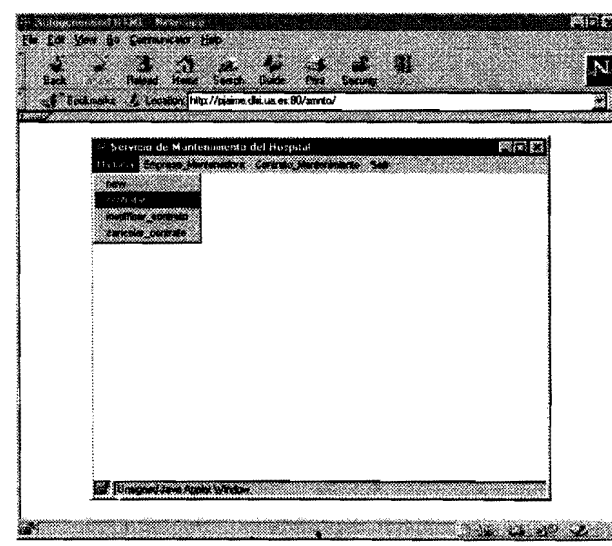

Figura 6: *Vista del sistema*

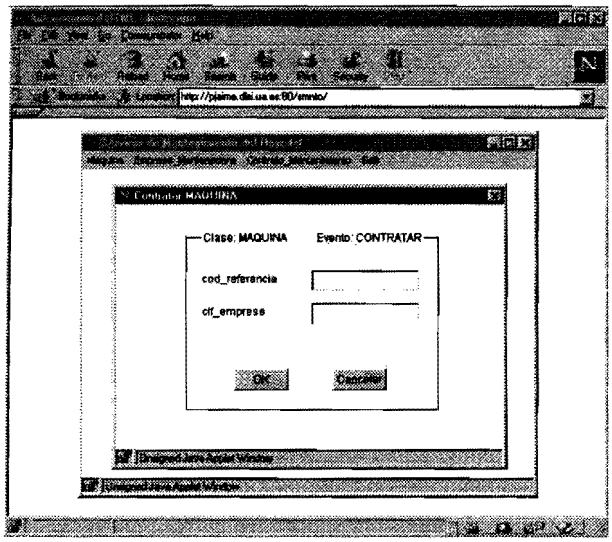

Figura 7: *Activación del servicio 'contratar'*

Finalmente se verificarán las restricciones de integridad y las condiciones de disparo en el nuevo estado alcanzado.

La actualización del estado del objeto se hará persistente en la base de datos a través de los servicios ofertados por la clase GestorPersistencia.

5 Conclusión y Trabajos Futuros

La construcción de aplicaciones Web más complejas dependerá en gran parte de hacer la tecnología de objetos más simple y segura. Para conseguir este objetivo debemos de producir marcos metodológicos bien definidos con una correcta conexión entre entornos de modelado conceptual OO y entornos de desarrollo de software OO. OO-Method proporciona esta conexión. Las características más relevantes son las siguientes:

- Obtención de una implementación operativa del paradigma de programación automática.
- Un modelo orientado a objeto bien definido cuya característica básica es el uso de un lenguaje de especificación como diccionario de datos de alto nivel.

Todo esto realizado dentro de entornos de desarrollo Web haciendo uso de arquitecturas Internet/Intranet y usando Java como lenguaje de desarrollo de software.

Actualmente se están llevando a cabo trabajos de investigación para mejorar la calidad del producto final software generado incluyendo características más avanzadas, tales como interfaces definidos por el usuario, evolución de esquema y mecanismos de optimización de acceso a base de datos.

Referencias

Balzer, R., Cheatham, T.E., and **Green, C.**, "Software technology in the 1990's: using a new paradigm", *IEEE Computer*, 14(5): 66-77, 1983.

Booch, G., *Object Oriented Design with Applications*, Benjamin/Cummings, 1991.

Booch, G., Rumbaugh, J., and **Jacobson, I.**, *UML v1*, Tech. Rept. Rational Software Corp., 1997.

Jacobson, I., Christerson, M., Jonsson, P., and **Overgaard, G.**, *OO Software Engineering, a Use Case Driven Approach*, Addison-Wesley, 1992.

Letelier, P. and **Ramos, I.**, *Un metamodelo estático para especificaciones OASIS y su implementación en una base de datos relacional*, Tech. Rept. DSIC, Universidad Politécnica de Valencia, 1995.

Pastor, O., and **Ramos, I.**, *OASIS 2.1.1: A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach*, 3a ed., Servicio de Publicaciones, Universidad Politécnica de Valencia, 1995.

Pastor, O., Hayes, F., and **Bear, S.**, "OASIS: An Object-Oriented Specification Language", in Loucopoulos, P. (ed), *Proceedings of CAiSE'92 International Conference*, Springer-Verlag, vol. 593, 1992, pp.348-363.

Pastor, O., Insfrán, E., Pelechano, V., Merseguer, J., and **Romero, J.**, "OO-Method: An OO Software Production Environment Combining Conventional and Formal Methods", in Olivé, A. and Pastor, O. (eds), *Proceedings of CAiSE'97 International Conference*, Springer-Verlag, LNCS, vol. 1250, 1997, pp. 145-158.

Platinum-Technology, Inc., *Paradigm Plus: Round-Trip Engineering for JAVA*, Tech. Rept. Project Technology, 1997.

Rational Software, Corp., *Rational Rose User's Manual*, Tech. Rept. Rational Software Corporation, 1995.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and **Lorensen, W.**, *Object Oriented Modeling and Design*, Prentice-Hall, 1991.

Dr. Oscar Pastor *es Profesor Titular en el Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia (DSIC), España. Oscar Pastor es Licenciado en Ciencias Físicas por la Universidad de Valencia y Doctor en Informática por la Universidad Politécnica de Valencia. Es miembro del Grupo de Investigación en Programación Lógica e Ingeniería del Software en el DSIC y responsable de varios proyectos de Investigación y Desarrollo en modelado conceptual orientado a objeto y prototipación automática. Sus actuales líneas de investigación comprenden ingeniería de requisitos, modelado conceptual, generación automática de código, diseño de bases de datos y metodologías orientadas a objetos.*

Vicente Pelechano *es Profesor Titular Interino en el DSIC de la Universidad Politécnica de Valencia. Recibió el grado de Licenciado en Informática por la Universidad Politécnica de Valencia, España. Sus intereses de investigación son orientación a objetos, modelado conceptual, ingeniería de requisitos, sistemas software basados en componentes y lenguajes de especificación. Es miembro del grupo de investigación en Programación Lógica e Ingeniería del Software en el DSIC.*

Emilio Insfrán *es Profesor Asociado en el DSIC de la Universidad Politécnica de Valencia. Sus intereses de investigación son metodologías orientadas a objetos, modelado conceptual, ingeniería de requisitos, lenguajes de especificación, y bases de datos. Recibió el grado de licenciado en Informática por la Universidad Nacional de Asunción, Paraguay y es Master en Informática por la Universidad de Cantabria, España. Es miembro del grupo de investigación en Programación Lógica e Ingeniería del Software en el DSIC y miembro de la IEEE. Actualmente realiza una estancia de investigación en la Universidad de Twente (Holanda) con el Prof.Dr. Roel Wieringa.*

Jaime Gómez *es Profesor Titular en el Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Alicante(DLSI), España. Sus intereses de investigación son orientación a objetos, modelado conceptual, ingeniería de requisitos, tecnologías internet/intranet, sistemas distribuídos y generación de componentes software. Recibió el grado de licenciado en Informática por la Universidad Politécnica de Valencia. Es miembro del grupo de investigación en Programación Lógica e Ingeniería del Software en el DSIC, y miembro del grupo de investigación de Programación Lógica y Sistemas de Información en el DLSI. Actualmente se encuentra finalizando su tesis doctoral en 'Generación Automática de Componentes Software a partir de modelos Conceptuales OO'.*