Esta tesis doctoral contiene un índice que enlaza a cada uno de los capítulos de la misma.
Existen asimismo botones de retorno al índice al principio y final de cada uno de los capítulos.

Ir directamente al índice

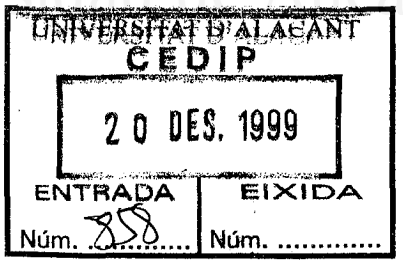
Para una correcta visualización del texto es necesaria la versión de Adobe Acrobat Reader 7.0 o posteriores

Aquesta tesi doctoral conté un índex que enllaça a cadascun dels capítols. Existeixen així mateix botons de retorn a l'index al principi i final de cadascun deis capítols .

Anar directament a l'index

Per a una correcta visualització del text és necessària la versió d' Adobe Acrobat Reader 7.0 o posteriors.

UNIVERSIDAD DE ALICANTE
Departamento de Lenguajes y Sistemas Informáticos



Generación Automática de Componentes Software a partir de Modelos Conceptuales Orientados a Objetos

Diciembre 1999

Memoria presentada para optar al grado de Doctor en Informática

Jaime Gómez Ortega

Dirigida por: **Prof. Dr. Oscar Pastor López**

... a mi Familia.

Resumen de la Tesis

Los métodos de modelado conceptual existentes en la actualidad deben de proporcionar marcos de trabajo formales que permitan definir el proceso de desarrollo de software con el propósito de migrar desde la especificación de requisitos a la implementación de forma automática.

Para alcanzar este objetivo, en esta tesis se aborda el problema del desarrollo de aplicaciones en el contexto de un método de producción automática de software basado en el paradigma orientado al objeto. El método proporciona una notación gráfica basada estrictamente en un lenguaje formal de especificación que determina los constructores de modelado necesarios para obtener una especificación de requisitos.

Partiendo de esa especificación de requisitos, esta tesis propone un modelo que establece una estrategia concreta para reificar cada constructor de modelado usado a nivel del espacio del problema, en su correspondiente representación software sobre el espacio de la solución.

La definición de una arquitectura distribuida basada en componentes sobre la que se soporta el modelo, proporciona el contexto necesario para generar de forma automática componentes software. Estos componentes adecuadamente combinados constituyen una aplicación software que preserva la funcionalidad capturada en la especificación de requisitos y es ejecutable sobre entornos internet/intranet distribuidos.

Agradecimientos

Quisiera expresar mis más profundo y sincero agradecimiento al director de esta tesis, el Dr. Oscar Pastor por su ayuda y soporte a lo largo del presente trabajo. Oscar ha sido especialmente paciente conmigo y buena parte del mérito de este trabajo proviene de su participación. Al Dr. Isidro Ramos, Catedrático de Universidad y líder del grupo de Programación Lógica e Ingeniería del Software del Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia, por darme la oportunidad de unirme a su grupo de investigación y por su inestimable ayuda en la realización del capítulo 5. También quisiera hacer una mención especial a mis compañeros de batalla Emilio Insfrán y Vicente Pelechano por sus valiosos comentarios que se ven reflejados en los resultados este trabajo. Gracias también a Manolo Palomar, Antonio Ferrández, Rafa Muñoz y a todos mis compañeros del grupo de investigación de Programación Lógica y Sistemas de Información de mi departamento por su apoyo y sugerencias.

Finalmente, muchas gracias también por el tiempo y la dedicación de todos aquellos que han participado en el proceso de revisión de forma desinteresada.

Alicante, Diciembre 1999 *Jaime Gómez*

Índice General

1. Introducción 1
1.1 Motivación 1
1.2 Modelado Conceptual 4
1.3 Análisis y Diseño Orientado a Objetos 7
1.4 Métodos Convencionales vs Lenguajes Formales 9
1.5 Propósito de la Tesis y Contribución 11

2. Revisión al Estado del Arte en Generación Automática de Código OO 15
2.1 Un poco de historia 15
2.2 Obtener la especificación formal del sistema 19
2.2.1 OMTROLL 19
2.2.2 OBLOG 20
2.2.3 Formalismos Sintonizables 21
2.2.4 Lenguajes de Modelo Equivalente 22
2.2.5 SOFL 23
2.2.6 TRADE 25
2.3 Generar la implementación del sistema 26
2.3.1 Generación de Estructura 28
2.3.2 Generación de Comportamiento 28
2.3.3 Generación Traslativa 30
2.4 Clasificación 31
2.5 Conclusiones del Capítulo 35

3. Generación Automática de Componentes Software 37
3.1 Introducción 37

3.2 Espacio del Problema 41
3.2.1 Paso I. Capturar un Modelo Conceptual 41
3.2.2 Paso II. Obtener la Especificación Formal OASIS 52
3.3 Espacio de la Solución 60
3.3.1 Paso III. Generar el Código OO 60
3.3.2 Paso IV. Obtener la Especificación de los Componentes 79
3.4 Comparación con Trabajos Relacionados 84
3.5 Conclusiones del Capítulo 88

4. El Sistema JEM 91
4.1 Introducción 91
4.2 Arquitectura 94
4.3 Ejecución de un Modelo Conceptual 96
4.4 Conclusiones del Capítulo 103

5. Equivalencia Semántica 105
5.1 Introducción 105
5.2 Un cálculo operacional para el modelo de ejecución 109
5.2.1 Evaluaciones 110
5.2.2 Derivaciones 113
5.2.3 Precondiciones 113
5.2.4 Restricciones de Integridad 114
5.2.5 Disparos 116
5.2.6 Especificación de Proceso 117
5.3 De la especificación a la implementación 119
5.4 Conclusiones del Capítulo 121

6. Conclusiones Finales 123
6.1 Aportaciones 125
6.2 Trabajos en Progreso 125
6.3 Publicaciones realizadas 129

Referencias 133

A. Caso de Estudio 147

A.1 Descripción 147
A.2 Elaboración del Modelo Conceptual 151
A.2.1 Requisitos del Sistema 152
A.2.2 Modelo de Objetos 153
A.2.3 Modelo Dinámico 172
A.2.4 Modelo Funcional 177
A.3 Especificación OASIS 183
A.4 Aplicación generada 195

B. Especificación Sintáctica de la Gramática OASIS 205
B.1 Introducción 205
B.2 Fórmulas 206
B.3 Enunciados 210
B.4 Unidades de la Especificación 215
B.5 Esquema Conceptual 216

C. El Componente Generador de Código 217

Índice de Figuras

2.1 *Compiladores de Modelos* 27
2.2 *Generación de Estructura* 29
2.3 *Generación de Comportamiento* 29
2.4 *Generación Traslativa* 30
2.5 *Formas de capturar la especificación formal* 32
2.6 *Clasificación según el nivel de generación* 34

3.1 Pasos clave en la propuesta OO-method 41
3.2 *Modelo de objetos del sistema biblioteca* 44
3.3 *DTE para la clase socio.* 46
3.4 *Sintaxis para las transiciones en el DTE* 46
3.5 *Diagrama de Interacción mostrando un disparo.* 47
3.6 *sintaxis de una especificación de disparo* 48
3.7 *Evaluaciones cardinales.* 51
3.8 *Evaluaciones de estado.* 51
3.9 *Evaluaciones de situación.* 52
3.10 Especificación OASIS para la clase socio 54
3.11 Plantilla del Modelo de Ejecución 61
3.12 *Arquitectura multicapa para el modelo de ejecución.* 64
3.13 *Instancia del componente* controlDeAcceso. 65
3.14 *Instancia del componente* vistaDelSistema. 66
3.15 *Instancia del componente* activacionDeServicio. 67
3.16 La clase abstracta OASIS 68
3.17 Patrón de diseño para las clases del dominio 69
3.18 Representación de precondiciones para la clase socio 71
3.19 Representación de las transiciones de estado de la clase socio 72

3.20 Representación de las evaluaciones para la clase socio 74
3.21 Servicios de la clase socio 74
3.22 Representación de restricciones de integridad para la clase socio 75
3.23 Representación de disparos para la clase socio 76
3.24 *Diseño de un mediador* 78
3.25 Código generado para la clase socio 82
3.26 Especificación CORBA IDL para la clase socio class 83
3.27 Especificación Java/RMI para la clase socio 83

4.1 *El entorno de producción y animación de software.* 93
4.2 *Arquitectura de JEM.* 95
4.3 *Flujo básico de un servlet.* 95
4.4 *Opciones del sistema JEM.* 96
4.5 *Descargando los artefactos software.* 97
4.6 *Selección de un modelo conceptual.* 98
4.7 *Configuración de parámetros para la ejecución.* 99
4.8 *Applet descargado por el cliente.* 99
4.9 *Conexión de usuario válido.* 100
4.10 *Vista del sistema.* 101
4.11 *Activación de servicios.* 102

5.1 *Estados de ejecución de un programa* 109
5.2 *Aplicación entre conjuntos* 119
5.3 *Equivalencia Semántica* 120

A.1 *Parte de avería y orden de trabajo.* 149
A.2 *Petición de material.* 150
A.3 *Solicitud de I.L.T.* 151
A.4 *Relación de Especialización.* 155
A.5 *Relación de absentismo laboral.* 158
A.6 *Contratos de mantenimiento de las máquinas.* 162
A.7 *Revisiones de máquinas.* 163
A.8 *Resolución de incidencia.* 166
A.9 *Hacer pedidos.* 168

A.10 *Gestión de incidencias*. 170
A.11 *DTE de la clase área*. 172
A.12 *DTE de la clase personal*. 173
A.13 *DTE de la clase incidencia*. 174
A.14 *DTE de la clase pedido*. 174
A.15 *DTE de la clase máquina*. 175
A.16 *DTE de la clase revisión*. 175
A.17 *Disparo de stock mínimo*. 176
A.18 *Interacción global recibirPiezas*. 177
A.19 *Interacción global usarPiezas*. 177
A.20 *Menú de especialización*. 195
A.21 *Registrar un absentismo laboral*. 196
A.22 *Consulta de absentismo laboral*. 197
A.23 *Contratos de mantenimiento de máquinas*. 198
A.24 *Estado de las revisiones*. 199
A.25 *Insertar líneas de incidencia*. 200
A.26 *Introducir un pedido*. 201
A.27 *Registrar la ocurrencia de una incidencia*. 202
A.28 *Clasificar una incidencia*. 203
A.29 *Solucionar una incidencia*. 203

C.1 *Componente Generador de Código*. 217
C.2 *Configuración del mediador*. 218
C.3 *Opciones avanzadas*. 219

Índice de Tablas

A.1 Requisitos del Sistema Servicio de Mantenimiento 152
A.2 Atributos clase personal 154
A.3 Servicios clase personal 154
A.4 Atributos clase jefe de mantenimiento 154
A.5 Servicios clase jefe de mantenimiento 154
A.6 Atributos clase operario 154
A.7 Servicios clase operario 155
A.8 Atributos clase área 155
A.9 Servicios clase área 155
A.10 Atributos clase absentismoLaboral 156
A.11 Servicios clase absentismoLaboral 156
A.12 Atributos clase tipo_baja 157
A.13 Servicios clase tipo_baja 157
A.14 Atributos clase máquina 159
A.15 Servicios clase máquina 159
A.16 Atributos clase contrato_mantenimiento 160
A.17 Servicios clase contrato_mantenimiento 160
A.18 Atributos clase empresa_externa 160
A.19 Servicios clase empresa_externa 161
A.20 Atributos clase revision 161
A.21 Servicios clase revision 161
A.22 Atributos clase incidencia 164
A.23 Servicios clase incidencia 164
A.24 Transacciones clase incidencia 165
A.25 Atributos clase lineaIncidencia 165
A.26 Servicios clase lineaIncidencia 165

A.27 Atributos clase piezas 165
A.28 Servicios clase piezas 166
A.29 Atributos clase pedido 167
A.30 Servicios clase pedido 167
A.31 Atributos clase lineaPedido 167
A.32 Servicios clase lineaPedido 168
A.33 Atributos clase servicio 169
A.34 Servicios clase servicio 169
A.35 Relaciones de agente del sistema 171
A.36 Atributos variables 178
A.37 Atributos de estado clase incidencia 178
A.38 Atributos de situación clase personal 179
A.39 Atributos de estado clase resp_servicio 179
A.40 Atributos de estado clase Operario 179
A.41 Atributos de estado clase pieza 180
A.42 Atributos cardinales clase pieza 180
A.43 Atributos de situación clase pedido 180
A.44 Atributos estado clase pedido 180
A.45 Atributos de situación clase máquina 181
A.46 Atributos estado clase máquina 181
A.47 Atributos de estado clase Abs_Laboral 181
A.48 Atributos de situación clase revisión 182

1. Introducción

"... es probable que las técnicas de especificación formales, basadas en principios matemáticos, pasen a ser el fundamento de una futura generación de herramientas CASE..."

(PRESSMAN, 1998)

1.1 Motivación

Si intentáramos definir la época histórica en la que nos encontramos actualmente, podríamos etiquetarla como la era de la *revolución de la información.* Debido a avances en campos tales como las telecomunicaciones y la informática, sin darnos apenas cuenta, estamos siendo desbordados por una cantidad ingente de información. Sin embargo, a diferencia de fenómenos similares que durante el transcurso de los años han acontecido en nuestra sociedad (ej. la revolución industrial), esta revolución se caracteriza por un aspecto de gran impacto para las personas dedicadas a la Ciencia de la Computación: la mayoría de la información disponible hoy en día, se encuentra almacenada en ordenadores bajo formatos tales como ficheros o bases de datos.

Por todos estos motivos, las tareas para los informáticos que trabajan en las áreas relacionadas con los sistemas de información, deben de estar muy orientadas a desarrollar teorías, herramientas y técnicas que manejen adecuadamente toda esta información.

En este sentido, las técnicas que históricamente se han estado utilizando para la construcción de sistemas software han dejado de ser adecuadas. La creciente demanda de nuevos servicios, tales como el trabajo cooperativo y distribuido, la recuperación de in-

formación, los servicios de traducción o los almacenes de datos, han dado lugar al resurgimiento dentro del campo de la Ingeniería del Software (IS) de lo que se ha dado en llamar como el modelado de la información (Mylopoulos, 1998).

Ante este panorama, la Ingeniería de Sistemas de Información necesita ser extendida para soportar arquitecturas software que permitan no sólo la construcción de nuevos sistemas de información sino que además puedan ser capaces de adaptar sistemas heredados[1].

El modelado de la información constituye una pieza clave para cualquier técnica que pretenda abordar esta demanda creciente de nuevos y mejores servicios. Para usar la información necesitamos representarla, capturar su semántica y estructura inherente. Estas representaciones no sólo son importantes para 'comunicar información' entre gente, sino para construir sistemas que manejen y exploten la información de forma adecuada.

El modelado de la información es una práctica utilizada desde los primeros sistemas de procesamiento de datos allá por los años 50, donde se usaban estructuras de registros y ficheros para modelar y organizar la información. Desde entonces, han surgido varias propuestas de modelos de información, orientadas a cubrir diferentes áreas de la Ciencia de la Computación y de la Ingeniería de los Sistemas de Información. Estos modelos que *Mylopoulos* (Mylopoulos, 1998) clasifica en tres categorías diferentes, reflejan el avance histórico del estado del arte en el modelado de la información desde las primeras representaciones orientadas a la máquina, hasta los últimos modelos orientados al ser humano mucho más expresivos y adecuados para modelar aplicaciones complejas:

- **Modelos de Información Físicos**: éstos empleaban estructuras de datos convencionales u otros constructores de programación para modelar una aplicación en términos de registros, cadenas, arrays, listas, ... El principal inconveniente de tales

[1] Legacy Systems

modelos era que se forzaba a los programadores o modeladores a enfrentarse con dos tipos de conflictos:

– uno relacionado con la eficiencia computacional y,

– otro referente a la calidad del modelo de aplicación.

El llamado efecto 2000, es un ejemplo relevante en este sentido. Durante años los programadores abreviaron las fechas de los años con dos dígitos asumiendo de forma implícita que los dos dígitos "faltantes" eran el 19.

- **Modelos de Información Lógicos**: a principios de los 70 aparecieron varias propuestas de modelos de datos lógicos que ofrecían estructuras abstractas de símbolos matemáticos (ej. conjuntos, arrays, relaciones), para propósitos de modelado que ocultaban los detalles de implementación al usuario. Los modelos relacional y red para base de datos son buenos ejemplos. Tales modelos liberaban al modelador de los aspectos de implementación, y las tareas se centraban en el modelado del sistema. Por ejemplo, el modelo relacional utilizaba tablas para construir su base de información sin preocuparse cómo éstas iban finalmente a ser implementadas.
- **Modelos de Información Conceptuales**: inmediatamente después de que fueran propuestos los modelos de datos lógicos e incluso antes de que la tecnología relacional conquistara la industria de las bases de datos, surgieron nuevas propuestas de modelos de información que ofrecían mayor expresividad para modelar aplicaciones y estructurar bases de información. Estos modelos (de aquí en adelante, modelos conceptuales) ofrecían términos semánticos para modelar una aplicación, tales como entidad, actividad, agente y objetivo. Además ofrecían formas de organizar la información en términos de lo que conocemos como mecanismos de abstracción inspirados por la ciencia cognitiva (Collins & Smith, 1988), tales como la generalización, la agregación y la clasificación.

Durante años han surgido numerosas propuestas de modelos conceptuales. A continuación se revisan algunos de los más tempranos

que fueron lòs precursores de líneas de investigación fructíferas y que han influenciado el estado de la práctica en la industria. Como se verá, algunos de estos modelos surgieron independientemente unos de otros, y en diferentes áreas de investigación dentro de la Ciencia de la Computación.

1.2 Modelado Conceptual

Modelar una parte del mundo real, lo que llamamos el dominio de la aplicación, ha sido una de las principales inquietudes en varias áreas de la informática, tales como el modelado de datos en el mundo de las Bases de Datos, la representación del conocimiento en el campo de la Inteligencia Artificial o el modelado de requisitos en la Ingeniería del Software. Durante estos años, numerosas notaciones han sido propuestas para tales tareas de modelado. En general, un modelo conceptual comprende una colección de:

- Términos primitivos, que especifican un conjunto básico de constructores para crear símbolos.
- Mecanismos de estructuración, para ensamblar y organizar símbolos.
- Operaciones primitivas, para construir y consultar símbolos.
- Reglas de Integridad, que definen el conjunto consistente de estados en los que se puede encontrar un símbolo o los cambios de estados válidos.

Por ejemplo, el modelo original entidad-relación (ER) de *Peter Chen* (Chen, 1976), ofrece entidades, relaciones y atributos como términos primitivos, soporta una forma limitada de clasificación (porque entidades y relaciones son instancias de tipos de entidades y relaciones), ofrece operaciones primitivas para crear nuevas entidades o tipos de relaciones o instancias de ellas, y soporta restricciones de cardinalidad para las relaciones, tales como 'un cliente puede hacer uno o más pedidos'. Una extensión importante

a este modelo fué el modelo entidad-relación extendido que soporta todas las características de su predecesor y también ofrece dos mecanismos de abstracción, la generalización y la agregación para abordar la composición de entes complejos. El modelo ER fué uno de los primeros modelos de datos semánticos porque asume que el dominio a ser modelado consiste de entidades y relaciones.

En el campo de la Inteligencia Artificial y casi 10 años antes, fueron propuestas las redes semánticas (Quillian, 1968) como 'un modelo simbólico idóneo para la memoria humana'. Las redes semánticas consisten en grafos etiquetados y dirigidos cuyos nodos representan conceptos mientras que los enlaces representan relaciones binarias. La propuesta de *Quillian* ya soportaba la generalización como mecanismo de estructuración. Posteriormente, se introdujeron varias mejoras para construir lenguajes de representación del conocimiento basados en marcos[2]. Las descripciones lógicas, que son una forma popular de construir lenguajes de representación del conocimiento hoy en día, se originaron a partir de esta línea de investigación.

En la disciplina de la Ingeniería del Software, *Ross* propuso la técnica de análisis y diseño estructurado (SADT) como 'un lenguaje para comunicar ideas' (Ross & Schoman, 1977) sobre mediados de los 70. En SADT, el mundo consiste en actividades y datos, que se representan mediante cajas y flechas. Cada actividad consume y produce algunos datos, los datos se representan con flechas que entran y salen de las actividades. Además, cada actividad tiene asociada datos que controlan su ejecución, que ni se consumen ni se producen, y algunos agentes externos (hardware o humanos) que los ejecutan. Diagramas análogos podían ser usados para modelar datos de una forma parecida. Otras técnicas de análisis estructurado que surgieron posteriormente tan populares como (DeMarco, 1979; Yourdon & Constantine, 1979; Gane & Sarson, 1979; Ward & Mellor, 1985), adoptaron las ideas del SADT pero se orientaron más específicamente a aspectos vinculados al modelado de los flujos de información o al modelado de sistemas de tiempo real entre otros.

[2] frames

La Ingeniería de Requisitos nació a mediados de los 70, gracias parcialmente a *Ross* y su propuesta SADT, y gracias parcialmente a otros trabajos como los de (Jackson, 1983), que constataron a través de estudios empíricos la existencia de los denominados 'problemas de requisitos'. Desde sus inicios, la Ingeniería de Requisitos se ha caracterizado por ser un proceso complejo, no determinístico, de conocimiento intensivo, basado en la experiencia y que requiere una actividad creativa e intelectual elevada. Por todo ello, la comunidad de la IS busca la manera de capturar aquellos requisitos que se consideran relevantes para construir un modelo conceptual correcto. La primera decisión a tomar tiene que ver con cual es el paradigma a usar para soportar el proceso de modelado conceptual. En este contexto, tanto la industria como la academia han aceptado el paradigma objetual con considerable energía y entusiasmo (Coad & Yourdon, 1990; Booch, 1991; Rumbaugh *et al.*, 1991; Jacobson, 1993; Snyder, 1993). Ello se ha debido principalmente a su proximidad a los mecanismos cognitivos humanos, a la encapsulación de los aspectos estructurales y de comportamiento proporcionados por la noción de objeto, y por el hecho de que la transición desde el espacio del problema al espacio de la solución se produce de forma más suave puesto que podemos especificar objetos en el modelo conceptual y podemos implementar objetos en la representación final del software.

De forma paralela y debido fundamentalmente a la creciente influencia de la programación orientada a objeto (OO) en la práctica de la programación, se produjo el surgimiento de un nuevo paradigma de análisis de requisitos de sistemas software. Este paradigma adopta las ideas de la programación OO y las mezcla con ideas de modelos de datos semánticos y de representación del conocimiento (redes semánticas), en un marco de modelado que es más potente que los métodos tradicionales de análisis y diseño estructurado. A los métodos que se generaron a partir de este paradigma se les denominó métodos de 'análisis y diseño orientado a objetos'.

1.3 Análisis y Diseño Orientado a Objetos

Los primeros métodos de análisis y diseño orientado a objetos (ADOO), se propusieron hace ya más de 10 años. El método de análisis de sistemas orientado a objetos (OOSA) (Shlaer & Mellor, 1988) adoptó el modelo entidad relación para capturar los aspectos declarativos de un sistema software añadiendo aspectos dinámicos. Esta aproximación, pronto fue seguida por tres propuestas más, el análisis orientado a objetos (OOA) (Coad & Yourdon, 1990), la técnica de modelado OO (OMT) (Rumbaugh *et al.*, 1991), y el diseño OO (Booch, 1991), que soportaban el modelado de los aspectos de estructura, de comportamiento, e interactivos de un sistema software. Hoy en día, existen docenas de técnicas muy parecidas y de herramientas comerciales basadas en la forma de pensamiento OO que soportan el proceso del desarrollo de software desde el análisis de requisitos hasta la implementación. De hecho, la gran promesa del análisis y diseño OO es que todo el proceso de desarrollo del software puede ser racionalizado y simplificado gracias a tener los mismos bloques de construcción (objetos, clases, métodos, mensajes, herencia), que se usan en todas las fases de desarrollo. En este contexto, la propuesta conocida como lenguaje de modelado unificado (UML) (Booch *et al.*, 1997; Booch *et al.*, 1998; Rumbaugh *et al.*, 1998), intenta integrar las características de los modelos de ADOO más destacados, y por consiguiente, mejorar la reusabilidad y consolidar el crecimiento del mercado ADOO.

Pero, ¿por qué se ha hecho tan popular el ADOO? En pocas palabras, porque ha provocado el avance de forma significativa del estado del arte en el modelado de requisitos. El análisis y diseño de sistemas se caracterizó hace 10 años por la existencia de un conjunto de técnicas de modelado separadas (diagramas de flujo de datos, diagramas ER, diagramas de transición de estados), que fueron usados para capturar la información que necesitaba ser modelada, analizada, diseñada y entendida antes de que el sistema software pudiese ser construido. Estos métodos generalmente ofrecían poca ayuda para estructurar especificaciones de requisi-

tos, que aseguraran su legibilidad. En contraste con esta situación, los métodos de ADOO ofrecen un marco coherente que integra un conjunto comprensivo de conceptos de modelado para capturar los aspectos estructurales, de comportamiento e interactivos de un sistema. Además soportan fuertemente dos mecanismos de estructuración, la generalización y la agregación, en términos de los cuales un modelador puede organizar y manejar la información capturada por sus modelos.

Desafortunadamente, muchos aspectos del ADOO son todavía inmaduros (Fichman & Kemerer, 1992; Pressman, 1998; Mylopoulos *et al.*, 1999). A menudo términos como los de clase, tipo, estado, propiedad, objeto, encapsulación y/o herencia, se encuentran sobrecargados semánticamente, lo que provoca que muchas compañías de desarrollo de software continúen experimentando grandes dificultades a la hora de: construir aplicaciones complejas (Maring, 1996), aplicar el proceso de desarrollo (Jacobson *et al.*, 1999), alcanzar la productividad deseada (Boehm, 1999), o asimilar los conceptos relacionados con este paradigma (Kozaczynski & Kuntzmann-Combelles, 1993; Meyer, 1997).

La mayoría de estos problemas derivan de dos hechos fundamentales:

- por un lado, muchos de los métodos basados en ADOO carecen de fundamentos teóricos integrados con modelos conceptuales formales y procedimientos de ingeniería correctos que aseguren la calidad del producto que se obtiene,
- por el otro, la ambigüedad derivada de los distintos conceptos dependiendo de si estamos en el espacio del problema o en el espacio de la solución.

A continuación, se presenta cómo se ha llegado a esta situación y qué posibles vías de solución existen.

1.4 Métodos Convencionales vs Lenguajes Formales

Tradicionalmente, podemos distinguir dos corrientes básicas que tratan el problema de modelar e implementar correctamente un Sistema de información (SI) desde el punto de vista objetual:

- los que podríamos denominar como métodos OO convencionales, que provienen de la experiencia adquirida en los entornos de producción de software industrial y que se basan en el uso no estructurado de una colección de diagramas cuya semántica es a menudo imprecisa, tales como OMT (Rumbaugh *et al.*, 1991), OOSE (Jackson *et al.*, 1994), Booch (Booch, 1991). Recientemente, la propuesta UML (Booch *et al.*, 1997; Booch *et al.*, 1998; Rumbaugh *et al.*, 1998) está tratado de proporcionar un lenguaje unificado de modelado que contemple las características de sus predecesores.
- los lenguajes de especificación OO formales, como OBLOG (Jungclaus *et al.*, 1987), TROLL (Sernadas *et al.*, 1987), ALBERT (Dubois *et al.*, 1994), OASIS (Pastor *et al.*, 1992; Pastor & Ramos, 1995; Letelier *et al.*, 1998), que proporcionan una sólida base matemática y propiedades formales.

Sin embargo, ambas aproximaciones presentan graves deficiencias por separado y ninguna de ellas ha podido dar una respuesta adecuada al problema. Por un lado, los métodos enmarcados en la primera aproximación se caracterizan por heredar lo que conocemos como los problemas de ambientes CASE típicos (Rolland, 1998):

- un conjunto excesivo de modelos con semánticas solapadas.
- transformaciones dificultosas y con pérdidas de expresividad entre modelos, lenguajes y herramientas.
- variabilidad en la calidad de la implementación del sistema así como en la calidad del diseño.

- falta de comprensibilidad debido a complejidades excesivas.

Por otro lado, los lenguajes de especificación formales enmarcados en la segunda aproximación, han sido criticados porque:

- son difíciles de aplicar en situaciones reales.
- requieren de conocimientos matemáticos elevados para escribir, leer y entender especificaciones.
- no existen lenguajes útiles y efectivos que sean fáciles de integrar en los métodos convencionales.
- la mayoría de los métodos formales se centran en la notación, pero no han sido concebidos para ayudar a los diseñadores a aplicar el método en un proceso de desarrollo práctico.

Recientemente, existe un interés creciente en que estas aproximaciones pueden combinarse adecuadamente. Trabajos como los de (Kush *et al.*, 1992), (Clyde *et al.*, 1992), (Jackson *et al.*, 1994), (Liddle *et al.*, 1995), y (Liu *et al.*, 1998), han constatado que la incorporación de los lenguajes formales OO en los métodos OO convencionales facilitan el proceso de un desarrollo de software más correcto y fiable. Por un lado, el uso de los primeros permite aprovechar la experiencia adquirida en los contextos industriales. Por otro, el uso de los segundos puede ayudar a los diseñadores a detectar y eliminar elementos de dudosa utilidad.

La característica principal de todas estas propuestas consiste en que han sido creadas usando marcos de trabajo formales para definir conceptos de manera precisa tanto a nivel del espacio del problema como a nivel del espacio de la solución.

El presente trabajo se enmarca en esta línea considerando que el desarrollo de software de calidad debe de orientarse hacia esta última corriente.

1.5 Propósito de la Tesis y Contribución

El trabajo que se presenta a continuación apuesta por centrar el desarrollo de software en la fase de modelado conceptual. Para capturar las propiedades relevantes del sistema se utiliza una aproximación orientada a objetos basada en un lenguaje de especificación formal. Este lenguaje, debido a las propiedades lógicas de que dispone, va a permitir especificar sistemas con una precisión superior a la que proporciona el uso de métodos convencionales de desarrollo de software. Se verá cómo el lenguaje adecuadamente 'oculto' en algún método gráfico familiar para el modelador, podrá ser utilizado de forma 'transparente' eliminando la complejidad de su uso y permitiendo que se pueda llevar a cabo uno de los objetivos que la IS ha estado intentando perseguir durante estos últimos años: la generación automática de código a partir de modelos conceptuales.

La contribución al estado del arte de esta tesis, toma como punto de partida la propuesta metodológica OO-Method (Pastor *et al.*, 1997c), construida sobre el modelo formal orientado a objeto OASIS (Pastor, 1992; Pastor *et al.*, 1992; Pastor & Ramos, 1995; Letelier *et al.*, 1998), fruto de las labores de investigación desarrolladas por el grupo de Programación Lógica e Ingeniería del Software (PLIS) del Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia. La característica principal de OO-Method es que los esfuerzos de los desarrolladores se centran en el paso de modelado conceptual, donde los requisitos del sistema son capturados de acuerdo con un conjunto finito y predeterminado de lo que denominamos constructores de modelado conceptual, puesto que son la representación de los conceptos relevantes a nivel del espacio del problema.

Esta tesis propone un modelo y una arquitectura para este modelo, que permite obtener de forma automática la implementación completa de un sistema software a partir de la especificación capturada en el paso de modelado conceptual. Y demuestra cómo este modelo, de aquí en adelante modelo de ejecución, proporciona un patrón para generar componentes software. Estos componentes

que produce el modelo de ejecución, adecuadamente distribuidos en una arquitectura software, permiten configurar un prototipo muy cercano al producto final que preserva la funcionalidad capturada en la especificación conceptual.

Por tanto, esta tesis se enmarca en el terreno de la generación automática de código a partir de modelos conceptuales orientados a objetos.

Una de las contribuciones más relevantes de esta tesis (Gómez *et al.*, 1999), es que cada constructor de modelado conceptual, tiene su correspondiente representación final software, y por tanto, la semántica capturada en el paso de modelado conceptual (espacio del problema) se preserva en la implementación generada (espacio de la solución).

La tesis está organizada como sigue:

Capitulo 2: Revisión al estado del arte en generación automática de código OO

Este capítulo introduce las aproximaciones más relevantes propuestas en la comunidad científica para resolver el problema de la generación automática de código a partir de modelos conceptuales, prestando especial atención a los modelos conceptuales orientados a objetos.

Capítulo 3: Generación automática de componentes software

El capítulo 3 comienza con una introducción de las características principales del proceso de modelado conceptual con OO-Method describiendo los diagramas que se usan para capturar las propiedades del sistema y que integran lo que se denomina modelo conceptual. A continuación se muestra como se obtiene automáticamente la especificación formal OASIS, punto de partida para aplicar el modelo de ejecución. El capítulo finaliza describiendo en detalle el modelo de ejecución y la arquitectura que da soporte al modelo para producir los componentes software que conformarán la aplicación resultante.

Capítulo 4: El sistema JEM

El capitulo 4 presenta JEM, una implementación del modelo de ejecución y su arquitectura hecha en Java que es directamente ejecutable en entornos internet/intranet, diseñada con el fin de poner en práctica las ideas propuestas en este trabajo.

Capítulo 5: Equivalencia semántica

En este capítulo se sientan las bases, con ayuda de un formalismo matemático, para mostrar que la implementación generada por el modelo de ejecución es semánticamente equivalente a la especificación formal OASIS inicial. Para ello, se define un cálculo operacional para el modelo de ejecución que establece de forma precisa los efectos de la ejecución de un programa. De esta forma, la equivalencia semántica se puede demostrar verificando que el cálculo así definido establece una relación entre especificaciones e implementaciones.

Capítulo 6: Conclusiones Finales

En este capítulo se presentan las conclusiones derivadas de este trabajo, los trabajos en progreso y líneas futuras que se pretenden continuar investigando, y la producción científica que ha generado esta tesis.

Apéndice A: Caso de estudio

Se presenta la aplicación del método (modelo conceptual, especificación OASIS, modelo de ejecución), sobre un caso real de sistemas de información; el servicio de mantenimiento del hospital general de Alicante.

Apéndice B: Especificación sintáctica de la gramática OASIS

En este apéndice se presentan las distintas construcciones sintácticas de la versión del lenguaje OASIS utilizada en este trabajo.

Apéndice C: El componente generador de código

Volver al índice/Tornar a l'índex

Finalmente, se describe en este apéndice aspectos técnicos del componente que implementa el proceso automático de generación en el contexto de una herramienta de producción de software.

2. Revisión al Estado del Arte en Generación Automática de Código OO

"... 'introduzca la descripción del problema, y nosotros haremos el resto'. Afortunadamente para la programadores, la realidad ha sido menos halagüeña de lo que los departamentos de marketing querían hacernos creer ..."

(MELLOR, 1999)

En este capítulo se revisan algunas de las aproximaciones más interesantes que a lo largo de estos últimos años han sido propuestas en la comunidad científica para resolver el problema de cómo obtener de forma automática la implementación de un sistema a partir de la especificación capturada en el paso de modelado conceptual. Se realiza una revisión cronológica de la historia del desarrollo del software que va desde los 'modelos informales' pasando por los 'modelos formales' hasta los 'modelos mixtos'. Estos últimos han probado ser los más adecuados para el objetivo que se pretende cubrir en esta tesis. De los métodos más representativos basados en estos modelos, se examinan dos aspectos fundamentales; el primero de ellos es cómo permiten obtener una especificación formal del sistema, el segundo, cómo generan a partir de ella la implementación final del mismo. Finalmente se presentan las conclusiones y se sitúa la tesis en el contexto adecuado.

2.1 Un poco de historia

Los primeros intentos para explicar a los clientes los detalles de un sistema fueron realizados utilizando descripciones en lenguaje natural. Estas descripciones fueron enriquecidas posteriormente con diagramas gráficos y algunas guías de diseño que dieron paso a los

denominados 'modelos informales' caracterizados por carecer de fundamentos teóricos en su base. Entre los más representativos se encuentran el análisis estructurado (Yourdon, 1989), el diseño estructurado (Jackson, 1983), y el desarrollo estructurado de datos (Warnier, 1981).

Con estos modelos, la especificación de requisitos llegaba a ser más precisa y entendible, pero sin embargo, seguía presentando varios problemas. El principal motivo residía en el hecho de que las especificaciones informales carecían de la precisión necesaria debido fundamentalmente a redundancias y a descripciones incompletas que con frecuencia aparecían en las especificaciones en lenguaje natural. Además surgían ciertos problemas adicionales en las fases de desarrollo y prueba porque no era fácil localizar problemas durante la implementación, problemas que se debían principalmente a deficiencias en la especificación.

Para resolver este problema el concepto de prototipo fué introducido en los ambientes de desarrollo con el objetivo de mejorar el nivel de entendimiento entre clientes y desarrolladores. En este sentido, la inclusión de prototipos en el ciclo de desarrollo del software fué beneficiosa. Sin embargo, el prototipado también arrastraba sus propios problemas. La adición de un prototipo a un método informal requería dos cambios de paradigma durante la primera fase de un proyecto;

- el primero se producía entre el modelo de análisis y el lenguaje de prototipado,
- el segundo cambio se producía cuando la información obtenida durante la ejecución del prototipo debía retroalimentar los modelos de análisis.

Este trabajo extra aumentaba la pérdida potencial de información en los cambios que se producían de un paradigma a otro.

Para incrementar de forma precisa la comunicación entre el cliente y el desarrollador, investigadores y un pequeño número de desarrolladores empezaron a utilizar técnicas más formales de análisis y especificación. El resultado fué la aparición de los lenguajes de

especificación formales que permitían obtener una representación formal abstracta de los aspectos relevantes del Universo del Discurso (UoD). Es lo que se denominó especificación conceptual (Brodie *et al.*, 1984; Wieringa, 1990) o especificación de requisitos (Zave, 1979).

De esta forma, las técnicas formales soportadas por fundamentos matemáticos, contribuyeron al proceso de desarrollo añadiendo principios de ingeniería a la construcción del software y proporcionando un marco riguroso dentro del cual especificar sistemas. Esta forma de desarrollar software tenía una gran ventaja:

- los lenguajes formales podían ser interpretados y ejecutados sobre entornos de programación lógicos que soportaban de forma natural éstos formalismos, y por consiguiente,
- la generación de un prototipo podía potencialmente realizarse de forma automática.

Sin embargo, estos 'importantes' beneficios se alcanzaban tras pagar un elevado coste, ya que los fundamentos matemáticos en los que se basaban dificultaban el uso de éstos por parte de los desarrolladores.

En el contexto del paradigma objetual (marco de trabajo en el que se presenta esta tesis), se disponen de varios lenguajes formales para especificar sistemas de información. Los hay de distinta naturaleza; entre los más representativos se encuentran OBLOG (Sernadas *et al.*, 1987), TROLL (Jungclaus *et al.*, 1987; Jungclaus *et al.*, 1995), ALBERT (Dubois *et al.*, 1994), OASIS (Pastor *et al.*, 1992; Pastor & Ramos, 1995; Letelier *et al.*, 1998). Todos ellos, fundamentalmente basados en lógica, especificaciones algebraicas, teoría de conjuntos o teoría de categorías.

A lo largo de este tiempo las demandas sobre las especificaciones de modelos conceptuales formales han sido muy variadas. Autores como (Brodie *et al.*, 1984; Parsch, 1990), aconsejaban que las especificaciones debían de ser legibles y entendibles tanto por especialistas del dominio como por desarrolladores de sistemas,

debían de soportar abstracciones e integridad conceptual mediante un preciso lenguaje formal, debían de soportar modificabilidad (en el caso de producirse cambios en los requisitos), y finalmente debían de ser independientes de la implementación, es decir, especificar 'qué tiene que ser representado' en vez de 'cómo debería ser representado'. La mayoría de estas recomendaciones no se tuvieron en cuenta en los métodos 'convencionales' OO (OMT (Rumbaugh *et al.*, 1991), Booch (Booch, 1991), OOSE (Jacobson *et al.*, 1992)), nacidos de forma paralela en la industria del desarrollo de software con el auge de la 'orientación a objetos'.

Como se comentó en el capítulo anterior, durante estos últimos años y respondiendo a estas exigencias ha existido una importante actividad en la integración de los lenguajes de especificación formales OO (Jungclaus *et al.*, 1987; Sernadas *et al.*, 1987; Dubois *et al.*, 1994; Pastor *et al.*, 1992) con los métodos convencionales OO (Rumbaugh *et al.*, 1991; Jacobson *et al.*, 1992; Booch, 1991; Booch *et al.*, 1997). Esta situación ha provocado la aparición de propuestas (Kush *et al.*, 1992; Clyde *et al.*, 1992; Jackson *et al.*, 1994; Liddle *et al.*, 1995; Liu *et al.*, 1998), que sugieren el 'matrimonio' entre ambas aproximaciones y que permiten alcanzar beneficios tales como:

- descubrir y eliminar ambigüedades y elementos de dudosa utilidad que aparecen en los métodos convencionales.
- la formalización de un modelo OO fuerza a dejar claro qué se debe implementar y no dejaría ninguna puerta abierta a decisiones de modelado para los implementadores.
- la combinación de los métodos convencionales con los lenguajes de especificación formales ayuda a la integración de los métodos formales en la industria.
- los lenguajes de especificación formales utilizan sistemas de inferencia lógicos que al combinarlos con los métodos convencionales, hacen que técnicas tan potentes como la animación del sistema, o la generación automática de código puedan estar disponibles en herramientas relacionadas con ambientes CASE.

Como resultado de estos esfuerzos de investigación, la comunidad científica en Ingeniería del Software empieza a creer de forma seria que el conjunto de métodos enmarcados en esta propuesta 'mixta' abre la puerta a unos de los objetivos que los ingenieros de software han estado intentando incluir en los ciclos de vida de sus proyectos durante mucho tiempo; la generación automática de código.

En estos métodos, el problema de obtener de forma automática la implementación de un sistema a partir de los requisitos iniciales, se reduce a resolver dos tareas primordiales:

i. Cómo obtener una especificación formal del sistema a partir de la información capturada a nivel de modelado conceptual.

ii. Cómo generar el código de la aplicación a partir de la especificación formal.

A continuación se presentan la estrategias usadas por algunos de los métodos o aproximaciones más relevantes enmarcados en este grupo para tratar el problema de capturar una especificación formal del sistema. Posteriormente, se examinan qué técnicas utilizan estos métodos para generar la implementación completa del sistema a partir de esa especificación formal. Se finaliza con un estudio que establece una clasificación de las distintas aproximaciones en base a éstos parámetros. De esta forma situaremos el trabajo que se presenta en esta tesis en el contexto adecuado.

2.2 Obtener la especificación formal del sistema

2.2.1 OMTROLL

El objetivo de la aproximación OMTROLL (Wieringa *et al.*, 1993) es asistir a los diseñadores mediante una notación gráfica compatible con OMT (Rumbaugh *et al.*, 1991), con el objetivo de construir

la especificación formal del sistema. El formalismo subyacente a la especificación está basado en el lenguaje de especificación TROLL (Jungclaus *et al.*, 1987; Jungclaus *et al.*, 1995; Grau *et al.*, 1998). La naturaleza formal de TROLL (que usa una lógica temporal distribuida (Ehrich *et al.*, 1998)), ayuda a obtener especificaciones concisas y no ambiguas. OMTROLL, tiene un soporte CASE conocido como TrollWorkBench (Tbench) (Kush *et al.*, 1992; Grau & Kowsari, 1997). Tbench proporciona un proceso de prototipado que permite la generación de un prototipo ejecutable independiente a partir de la especificación conceptual gráfica. El prototipo generado es un programa C^{++} que incluye los aspectos estáticos y dinámicos del sistema y utiliza una base de datos Ingres como repositorio de la especificación. Este entorno CASE se centra en los procesos de validación de especificaciones y comprobación de modelos.

2.2.2 OBLOG

OBLOG (Sernadas *et al.*, 1987; Camara & Andrade, 1999) es otra aproximación OO para el desarrollo de software que representa la convergencia de los esfuerzos de investigación en el contexto del proyecto ESPRIT IS-CORE[1]. La semántica de OBLOG se formaliza en el contexto de la teoría de categorías. OBLOG dispone de una herramienta CASE (Oblog, 1999; Andrade *et al.*, 1998) que comparte el objetivo de migrar desde la especificación a la implementación mediante el uso de un lenguaje de especificación formal (OBLOG en este caso). El proceso de generación automática es visto en OBLOG como un proceso de transformación basado en reglas de reescritura. Las reglas son escritas utilizando un lenguaje específico que debe de ser conocido por el modelador. La habilidad para generar código depende directamente de la codificación de las reglas. El lenguaje para escribir las reglas se basa en gramáticas atributivas (Aho *et al.*, 1986).

[1] Information Systems - Correctness and Reusability

2.2.3 Formalismos Sintonizables

El concepto de formalismo sintonizable propuesto por *Clyde* (Clyde *et al.*, 1992) supone un avance más en el proceso de integración entre los métodos convencionales OO y los lenguajes de especificación formales. Clyde observa que los métodos que soportan formalismos sintonizables permiten satisfacer las necesidades tanto de teóricos como de desarrolladores. De forma intuitiva un modelo presenta un formalismo sintonizable si permite que el usuario del modelo pueda trabajar con diferentes niveles de formalización, desde un nivel totalmente informal hasta el más puro rigor matemático. Para ello, el formalismo debe de soportar las siguientes características:

i. ser lo suficientemente expresivo para el desarrollador,

ii. tener una sintaxis y semántica formalmente definida y,

iii. permitir varios niveles de detalle y completitud.

La propuesta de *Clyde* parte del método OSA[2] (Embley *et al.*, 1992), un método OO similar a los que proporcionan los métodos convencionales OO cuya característica principal es que los componentes de modelado de OSA están formalmente definidos. De esta forma OSA también es similar a otros modelos formales que están basados en algún lenguaje lógico pero con la característica añadida de soportar prácticas de ingeniería informal más cercanas al desarrollador.

Además OSA permite que ciertos componentes de modelado no necesiten estar totalmente descritos conforme se van añadiendo a la instancia del modelo. En la propuesta de *Clyde* se puede por ejemplo describir inicialmente los disparos, acciones, y restricciones en lenguaje natural narrativo. De esta forma estos aspectos son informales, puesto que sólo representan sentencias cuya verdad no puede ser computada de forma automática. En este sentido cuando se anima la instancia del modelo con un componente incompleto, algún oráculo fuera del modelo debe de decidir si es

[2] Object-oriented System Analysis

cierto o no. Cuando se desea ejecutar el modelo sin consultar a ningún oráculo debemos de describir los componentes en OSM-Logic (Embley *et al.*, 1992), el lenguaje procedural asociado a OSA.

2.2.4 Lenguajes de Modelo Equivalente

Otra contribución substancial en este contexto es la propuesta de los lenguajes de modelo equivalente[3]. *Liddle* (Liddle *et al.*, 1995) propone este concepto para solucionar el problema de la pobre integración entre los modelos y los lenguajes en el desarrollo de aplicaciones avanzadas. En su estudio observa que la utilización de modelos semánticos para facilitar el entendimiento en el análisis de las aplicaciones entra en conflicto con los lenguajes usados para implementar los sistemas, puesto que los lenguajes ni están totalmente integrados con el modelo ni son totalmente compatibles con el mismo. Esta falta de integración entre modelos y lenguajes es la causa de innumerables problemas para automatizar el proceso de migración del análisis a la implementación, entre los que se incluyen:

- dificultades y pérdidas de transformación entre modelos, lenguajes y herramientas,
- comunicaciones restringidas,
- barreras para desarrollar herramientas integradas entre el análisis y la implementación, y sobre todo,
- falta de comprensibilidad debido a complejidades innecesarias.

Conceptualmente un lenguaje $\mathcal{L}$ es equivalente a un modelo $\mathcal{M}$, si para cada instancia del modelo $\mathcal{I}_{\mathcal{M}}$ de $\mathcal{M}$, existe un programa $\mathcal{I}_{\mathcal{L}}$ de $\mathcal{L}$ cuya semántica está en relación uno-a-uno con $\mathcal{I}_{\mathcal{M}}$, y recíprocamente, para cada programa $\mathcal{I}$ de $\mathcal{L}$ existe una instancia del modelo $\mathcal{I}_{\mathcal{M}}$ de $\mathcal{M}$ cuya semántica está en relación uno-a-uno con $\mathcal{I}_{\mathcal{L}}$.

[3] Model-equivalent Languages

Por semántica uno-a-uno *Liddle* se refiere a que cada constructor en el programa, tiene su correspondiente constructor en la instancia del modelo y viceversa. De tal forma, que un programa escrito en un lenguaje de modelo equivalente es simplemente una vista alternativa de una instancia del modelo y por consiguiente está totalmente integrado y es completamente compatible con la instancia del modelo.

La ventaja básica de los lenguajes de modelo equivalente es que eliminan la necesidad de transformaciones conforme se va cambiando el foco entre las diferentes visiones del sistema. *Liddle* argumenta que con un lenguaje de estas características se pueden resolver los siguientes problemas:

- las transformaciones en el ciclo de desarrollo no son más que desplazamientos de puntos de vista.
- las definiciones son idénticas para el modelo y para el lenguaje y por consiguiente, las herramientas pueden tratar el lenguaje y el modelo sin cambios de paradigmas.
- la alta calidad en el diseño puede ser llevada a una alta calidad en la implementación y por tanto la comprensión del sistema mejora notablemente puesto que sólo existe un paradigma a lo largo de todo el proceso de desarrollo.

En su trabajo *Liddle* propone un lenguaje de modelo equivalente para un modelo concreto, OSA (Embley *et al.*, 1992) y un lenguaje concreto, Melody (Liddle, 1995).

2.2.5 SOFL

SOFL (Liu *et al.*, 1998), es una aproximación de modelado cuyo propósito es abordar el proceso de desarrollo de software mediante el uso de métodos formales, especificaciones estructuradas y un método OO. Concretamente, SOFL propone un método en donde la especificación de requisitos se captura siguiendo un estilo estructurado, posteriormente esa especificación se transforma en un

diseño objetual y se finaliza con un estilo de implementación de acuerdo al entorno de explotación del sistema (ambiente estructurado u objetual).

En este sentido, SOFL puede ser visto como un proceso que guía la fase de desarrollo junto con un método específico para construir el software. El proceso de desarrollo se divide en dos fases: desarrollo de estructura y desarrollo de comportamiento, en cada una de ellas se capturan diferentes perspectivas de los requisitos.

El desarrollo de estructura se divide en cuatro etapas: análisis preliminar de requisitos, especificación de requisitos, diseño e implementación. El análisis preliminar de requisitos consiste en extraer requisitos informales de alto nivel a partir de las descripciones proporcionadas por el usuario del sistema. La especificación de requisitos se formaliza mediante un proceso iterativo que a partir de una especificación abstracta produce como resultado una especificación concreta del sistema. En la etapa de diseño la especificación formal de requisitos resultante es transformada en un diseño formal a través de refinamientos sucesivos. Finalmente, la implementación se obtiene mediante un proceso que transforma la especificación de diseño en un programa escrito en el lenguaje SOFL. De esta forma, el programa obtenido puede servir indistintamente como prototipo o como diseño detallado a partir del cual se puede construir una implementación más eficiente del sistema utilizando una lenguaje de programación más conveniente.

El desarrollo del comportamiento en SOFL es un proceso diseñado para descubrir los aspectos dinámicos del sistema mediante técnicas de prototipación. Para ello se construye un prototipo que permite comprobar si las necesidades actuales del usuario se satisfacen de forma adecuada. De esta forma es posible descubrir nuevos requisitos que inicialmente no resultaban claros para el usuario.

SOFL utiliza técnicas tales como diagramas de flujo de datos condicionales, especificaciones estructuradas de módulos asociados a éstos diagramas, transformaciones de las especificaciones estructuradas en diseño OO y finalmente transformaciones de diseño OO

a programas, para abordar el proceso de desarrollo de estructura y comportamiento del sistema.

2.2.6 TRADE

TRADE (Wieringa, 1998), es otra propuesta que comparte en gran medida la filosofía de SOFL. A diferencia de SOFL, la contribución más relevante de TRADE es su habilidad para introducir técnicas de forma independiente al método. De esta forma TRADE puede contener técnicas estructuradas, especificaciones OO y métodos de diseño convencionales que, combinados adecuadamente, permiten generar una especificación de diseño de software coherente. En el marco conceptual de TRADE se distingue entre interacciones externas al sistema y componentes internos. Las interacciones son vistas como funciones externas a través de las cuales se especifican los aspectos de comunicación y comportamiento externos del sistema. Algunas de las técnicas usadas en TRADE para especificar interacciones externas son: diagramas de contexto, casos de uso, listas evento-respuesta o árboles de decisión. A partir de una especificación de estas características, es posible diseñar una descomposición del sistema en términos de un conjunto de componentes internos que cumplen con las propiedades de la especificación externa pero que abstraen ciertas propiedades asociadas a aspectos más detallados del sistema como pueden ser los de implementación. *Wieringa* etiqueta este proceso de descomposición como la descomposición esencial del sistema software (Wieringa & Dubois, 1998).

Una de las actividades más importantes a la hora de usar TRADE consiste en establecer de forma precisa las interconexiones necesarias entre las interacciones externas y los componentes internos (métodos OO, lenguajes de especificación formales), que permiten descomponer de forma sistemática los requisitos software de alto nivel del sistema en un conjunto detallado de requisitos que especifican de manera adecuada los componentes software que conformarán el sistema final. En este sentido y en el ámbito del

proyecto 2RARE[4], TRADE ha sido aplicado con éxito utilizando como componente interno de integración el lenguaje de especificación ALBERT (Dubois *et al.*, 1994). Un editor gráfico denominado TCM[5] da soporte a TRADE.

2.3 Generar la implementación del sistema

Históricamente, los sistemas basados en transformaciones (Bauer *et al.*, 1989), han sido usados para construir programas a partir de especificaciones formales. La idea de derivar una implementación desde un lenguaje de especificación formal a partir de su semántica, se intentó por primera vez de forma experimental en el sistema SIS (Mosses, 1979) hace aproximadamente 20 años. Desde entonces, han habido muchas propuestas relacionadas con el cálculo formal de programas y la siempre ambigua tarea de automatizar la generación de código. Un avance importante durante el transcurso de estos años se produjo aproximadamente hace 15 años en el proyecto CIP de Munich (Bauer *et al.*, 1985; Bauer *et al.*, 1987). Concretamente, en este proyecto se abordaba el desarrollo de programas como un proceso formal que transformaba la especificación de un dominio del problema en un programa eficiente para una máquina determinada. Este proceso se realizaba aplicando una serie discreta de pasos, donde cada paso se correspondía con la aplicación de una transformación que preservaba el significado[6] (TPS). De esta forma el programador seleccionaba la TPS adecuada en cada paso a través de una herramienta interactiva que comprobaba su aplicabilidad y la ejecutaba en caso favorable. En este proyecto, tanto el lenguaje usado (CIP-L) como el sistema de transformación (CIP-S) estaban basados en especificaciones algebraicas.

En el contexto del paradigma objetual, recientemente han aparecido un conjunto de técnicas de generación 'basadas en modelos' que

[4] 2 Real Applications for Requirements Engineering
http://www.info.fundp.ac.be/~phe/2rare.html

[5] Toolkit for Conceptual Modeling

[6] Meaning-preserving transformation

permiten producir de forma automática el código de la aplicación a partir de modelos OO gráficos (tanto estructurales y como de comportamiento). Estas técnicas que se presentan a continuación, se basan también en parte en un proceso transformacional que se apoya, en este caso, en el concepto de 'compilador de modelos'.

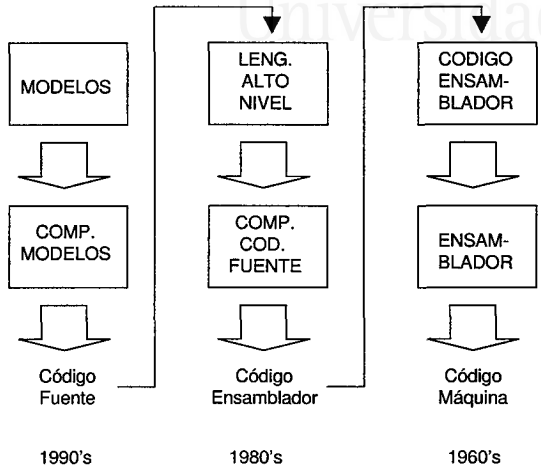


Figura 2.1. *Compiladores de Modelos*

Según (Mellor, 1999), un compilador de modelos representa una evolución natural en el esfuerzo de aumentar el nivel del abstracción durante el desarrollo del software. En este contexto, los modelos conceptuales OO capturados en la etapa de análisis y diseño adoptan el rol de 'técnica de programación', al igual que el que adoptaron el código máquina y el ensamblador en la década de los sesenta, y posteriormente los lenguajes de alto nivel en los ochenta como se refleja en la Figura 2.1.

Básicamente un compilador de modelos consta de dos componentes principales:

- un conjunto de reglas de traducción: que especifican como se construyen los 'patrones de implementación' en el lenguaje de programación destino. Este proceso se lleva cabo mediante la traducción de los 'constructores de modelado' que aparecen en los modelos OO capturados.

- una librería de ejecución: que proporciona las funciones necesarias para ejecutar de forma satisfactoria el código generado en el entorno de implementación escogido (sistema operativo, base de datos, entorno de red) .

En este contexto, en (Bell, 1998) se identifican tres aproximaciones de generación automática de código a partir de modelos conceptuales OO: la generación de estructura, la generación de comportamiento y la generación traslativa. Estas aproximaciones son 'incrementales', es decir, cada una de ellas mejora los defectos que presentan las anteriores. Estas estrategias reflejan el 'estado del arte' en las técnicas de generación de código a partir de la información capturada en el paso de modelado conceptual. A continuación se presentan en detalle.

2.3.1 Generación de Estructura

La generación de estructura comprende técnicas que producen código a partir de los modelos que capturan la estructura del sistema. Esta aproximación ilustrada en la Figura 2.2, permite a los desarrolladores evitar la tediosa tarea de escribir el código para las estructuras que se derivan de un diagrama de clases. Algunas herramientas CASE (Rational-Software, 1995; Platinum-Technology, 1997) que usan métodos OO convencionales del estilo de OMT (Rumbaugh *et al.*, 1991), Booch (Booch, 1991), y OOSE (Jacobson *et al.*, 1992), soportan generación estructural. La mayor crítica que se realiza a esta aproximación es que el código para capturar el comportamiento del sistema debe ser escrito a mano.

2.3.2 Generación de Comportamiento

La generación de comportamiento resuelve el problema de las técnicas anteriores. Algunos métodos OO modelan el comportamiento del sistema mediante máquinas de estados sobre las que añaden código en algún lenguaje para representar las acciones que acontecen en una transición de estado. Además, los métodos que

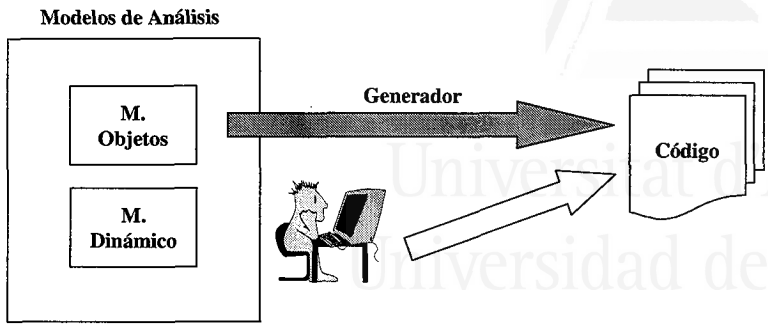


Figura 2.2. *Generación de Estructura*

soportan la generación de código para el comportamiento disponen de un beneficio añadido, y éste es que se puede simular y verificar el comportamiento del sistema antes de que el código sea generado. Entre estos métodos se encuentran SDL[7] (Broy, 1991), los diagramas de estados de *Harel* (Harel & Naamad, 1996) y ROOM[8] (Selic *et al.*, 1992).

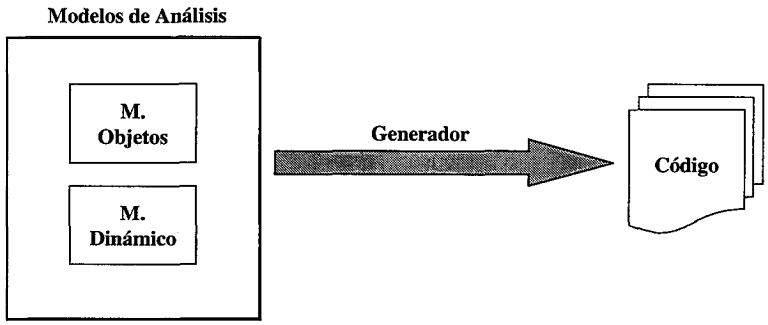


Figura 2.3. *Generación de Comportamiento*

En esta aproximación (ver Figura 2.3), el esquema de generación de código es completamente cerrado. El único punto de entrada al proceso son los modelos de análisis, y por tanto la única opción de cambiar algo es modificando estos modelos. Su mayor crítica la provoca el hecho de que todas las decisiones de diseño estén in-

[7] Specification and Description Language
[8] Real-time Object-Oriented Methodology

corporadas directamente en el generador, y por tanto el producto generado es 'poco reusable' en otros contextos de diseño.

2.3.3 Generación Traslativa

La generación traslativa resuelve estos problemas proporcionando dos puntos de entrada al proceso de generación de código.

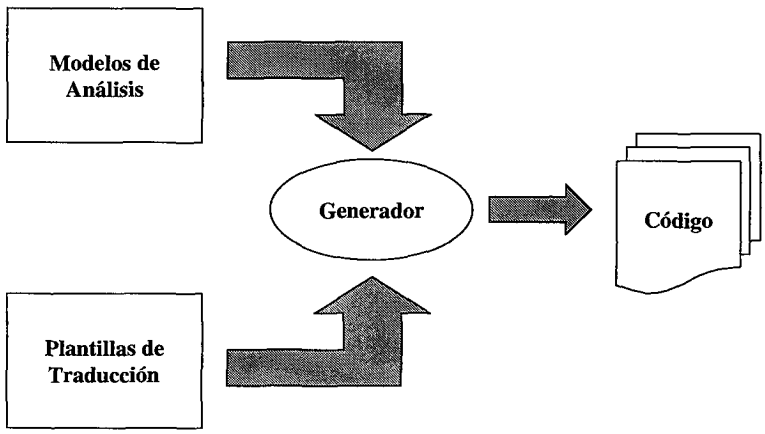


Figura 2.4. *Generación Traslativa*

El primero de ellos sigue siendo el de los modelos de análisis, adecuadamente expresados en un repositorio centralizado del sistema. El segundo punto de entrada es un conjunto de plantillas de traducción independientes de la arquitectura software de la aplicación. El generador produce código sustituyendo los elementos que aparecen en los modelos de análisis de acuerdo a las reglas definidas en las plantillas de traducción. Estas plantillas pueden generar el código destino para el entorno de ejecución más adecuado a las características del sistema (arquitectura). La aproximación traslativa, ilustrada en la Figura 2.4, favorece altamente el reuso porque los modelos de aplicación y arquitectura son completamente independientes.

Dentro de la aproximación traslativa podemos diferenciar claramente dos corrientes:

- la primera de ellas (Bell, 1998) adopta como condición necesaria y suficiente que un método soporta generación traslativa cuando proporciona un lenguaje que permita describir la arquitectura software del sistema generado de forma independiente al código de la aplicación (estructura y comportamiento).
- la segunda de ellas (Shlaer & Mellor, 1997), sostiene que un método soporta generación traslativa cuando en el proceso de generación de código se tienen en cuenta los aspectos de arquitectura software asociados a la aplicación final.

Teniendo en cuenta estas consideraciones, algunos de los métodos enmarcados en esta propuesta son OBLOG (Andrade *et al.*, 1998; Camara & Andrade, 1999), Shlaer & Mellor (Shlaer & Lang, 1996; Shlaer & Mellor, 1997) y Catalysis (D'Souza & Wills, 1998), OO-Method y nuestro trabajo.

2.4 Clasificación

Dentro del paradigma objetual, han surgido en la literatura varias propuestas de integración entre los métodos convencionales y formales de desarrollo de software con el objetivo centrado en la generación automática de código. Estas propuestas globales se pueden clasificar dependiendo de la filosofía que utilizan para realizar el proceso de integración. La mayoría de ellas consideran el proceso en términos de cómo capturar a nivel de modelado conceptual la información necesaria para construir la especificación formal del sistema.

Se puede realizar una clasificación de estas propuestas atendiendo a 3 estrategias básicas. En la Figura 2.5, se presenta esta clasificación en función de dos dimensiones: la dimensión vertical identifica la forma en que la información de modelado es capturada por el método, el grado varía desde una captura totalmente textual a una totalmente gráfica. La dimensión horizontal identifica el grado de formalización de la especificación que se obtiene, y

puede variar desde un grado completamente informal a un grado puramente formal.

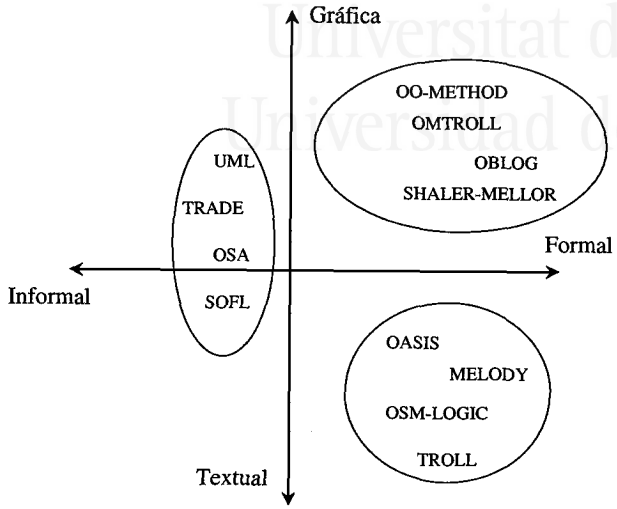


Figura 2.5. *Formas de capturar la especificación formal*

Atendiendo a estos parámetros y dependiendo del grado de experiencia del modelador se diferencian 3 estrategias básicas:

- un grupo formado por métodos basados estrictamente en lenguajes de especificación formales (Oasis, Melody, OSM-Logic, Troll). Estos métodos utilizan una captura textual de los requisitos codificando la información de modelado a través de los constructores del lenguaje. La especificación que se obtiene es completamente formal.
- otro grupo formado por métodos convencionales basados en lenguajes de especificación como OO-Method, OM-Troll, Oblog, Shlaer-Mellor, reducen la complejidad de proceso de captura de requisitos proporcionando constructores gráficos de modelado permitiendo de forma transparente completar la especificación formal del sistema. El proceso de captura es eminentemente gráfico (aunque no de forma completa). La especificación que se obtiene es completamente formal, puesto que todos éstos

métodos están basados en algún lenguaje formal de especificación como se vió anteriormente.

- y finalmente, otro grupo formado métodos de distinta naturaleza tales como UML, Trade, OSA y SOFL, donde la captura de la información de modelado es parcialmente gráfica puesto que determinadas secciones deben de ser capturadas de forma textual. Estos métodos se caracterizan por dejar en manos del modelador la decisión sobre el grado de formalización que se desea obtener de la especificación del sistema. En alguno de ellos, para obtener una especificación completamente formal es necesario completar la especificación del sistema de forma textual utilizando el lenguaje proporcionado por el método (por ejemplo, OSM-Logic para el método OSA).

Por otro lado, si examinamos como estas propuestas tratan el problema de la transición de la especificación al código, observamos que este proceso conlleva la resolución de varios problemas complejos relacionados con los 'desacoplos de impedancia' que surgen entre modelos y lenguajes y entre los que se incluyen:

- conflictos entre variable/tipo/clase.
- conflictos entre datos persistentes/transitorios.
- conflictos entre notación gráfica/textual.
- conflictos de procesamiento imperativo/declarativo.
- conflictos entre interacción objetual/invocación de operaciones.

La amplia experiencia adquirida en el contexto de los sistemas basados en transformaciones (derivación de programas lógicos a partir de especificaciones), resuelve de forma satisfactoria algunos de estos problemas. Sin embargo, estas técnicas no pueden aprovecharse completamente para el desarrollo de software industrial, principalmente por la necesidad de producir programas imperativos y no programas lógicos.

El talón de Aquiles en la aplicación de los métodos presentados en este capítulo es precisamente la transición al código. Sin ge-

neración de código los beneficios del modelado objetual no son aprovechados a lo largo del ciclo de vida del software, puesto que la mayoría de las veces los modelos quedan 'obsoletos'. Esto provoca que el mantenimiento evolutivo se realice directamente sobre el código en vez de sobre los modelos.

Por tanto, a la hora de adaptar los métodos OO para generación de código basada en la información capturada a nivel conceptual debemos de tener en cuenta:

- la facilidad del lenguaje de modelado para representar el problema.
- la suficiencia de los constructores de modelado para generar código.
- la madurez de los traductores para generar código de calidad.

Atendiendo a estas recomendaciones, podemos clasificar las propuestas anteriores en función de los siguientes parámetros: generación de estructura, generación de comportamiento y generación de arquitectura.

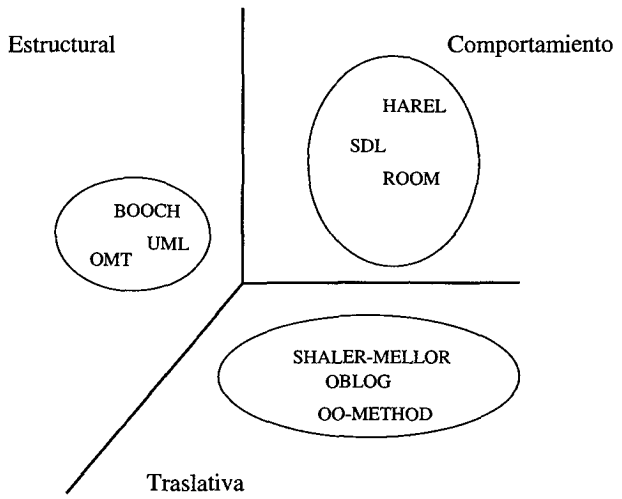


Figura 2.6. *Clasificación según el nivel de generación*

La generación de estructura la resuelven varias herramientas CASE basadas en métodos como OMT y Booch, generalmente produciendo una base de datos (relacional u objetual), que asegure la persistencia de los objetos del sistema.

La generación de la dinámica del sistema la ofrecen propuestas como Harel, SDL, o ROOM, que utilizan distintas estrategias para generar el comportamiento como el uso de lógica condicional o implementación de máquinas de estados.

Sólo un conjunto reducido de propuestas entre las que se encuentran Shlaer-Mellor, Oblog, OO-Method, permiten abordar la generación traslativa. Este proceso incluye la generación de la estructura (base de datos), la generación de la lógica de la aplicación (comportamiento) y la estructuración del código conforme a la arquitectura deseada (cliente/servidor, 3-capas, OLTP[9], ...).

2.5 Conclusiones del Capítulo

En este capítulo se ha presentado una revisión de la investigación desarrollada en métodos OO de generación automática de código. La revisión se ha realizado examinado con detalle dos aspectos básicos; cómo se obtiene una especificación formal del sistema, cómo se genera código a partir de ella. Se ha establecido una clasificación de los métodos atendiendo a éstos parámetros lo que ha permitido detectar cuales deben de ser las cualidades básicas que un método de generación automática de código debe de ofrecer, estas son:

- Uso de constructores gráficos de modelado para capturar los requisitos del sistema.
- Formalización de éstos constructores mediante algún lenguaje formal de especificación.
- Generación traslativa de código (estructura, comportamiento y arquitectura), a partir de la especificación obtenida.

[9] On-Line Transaction Processing

A continuación, en el capítulo siguiente se presenta el modelo que permite proporcionar estas características en el contexto de un método de modelado conceptual OO (el método OO-Method). De esta forma lo que se obtiene es un verdadero entorno de producción automática de software basado en el paradigma objetual. Una vez presentado este modelo, se realiza el estudio comparativo (ver sección 'comparación con trabajos relacionados' del capítulo 3), que contrasta nuestra propuesta con las aquí presentadas.

3. Generación Automática de Componentes Software

"... nuestra capacidad de producción de software no está a la par con la evolución del hardware. Se necesita una revolución industrial en el software. Es probable que esta revolución provenga de las técnicas orientadas a objetos, combinadas con herramientas CASE, generadores de código, programación visual y desarrollo basado en repositorios ..."

(MARTIN & ODELL, 1994)

Este capítulo se centra en la descripción del método OO-Method en el contexto de la producción automática de software basado en componentes. Primeramente se introducen las características principales del proceso de modelado conceptual con OO-Method, presentando la notación gráfica que se usa para capturar las propiedades del sistema y que integran el modelo conceptual. Posteriormente se detalla cómo esta notación, estrictamente basada en un lenguaje formal OO de especificación, determina los conceptos de modelado conceptual necesarios para obtener la especificación del sistema. El capítulo finaliza describiendo minuciosamente un modelo de ejecución abstracto que especifica a nivel de modelo y de arquitectura cómo obtener la representación software correspondiente para cada concepto de modelado conceptual. De esta forma, el producto final software se obtiene de forma automática.

3.1 Introducción

La mayoría del trabajo existente en el campo de desarrollo de software basado en componentes se ha centrado en desarrollar la infraestructura necesaria para conectar piezas independientes de

software (OMG, 1997; Box, 1997; Downing, 1998). En este contexto en el que nuevas tecnologías están emergiendo continuamente, los desarrolladores de aplicaciones necesitan entornos de trabajo cuyas propiedades incluyan:

- métodos para diseñar soluciones basadas en componentes que ayuden a que la organización se centre en las principales piezas de su dominio y en cómo estas piezas interactuarán.
- herramientas que soporten la especificación de componentes usando técnicas que permitan describir la funcionalidad del componente independientemente de la tecnología de implementación escogida.

En el campo de los métodos OO (Wirfs-Brock *et al.*, 1990; Rumbaugh *et al.*, 1991; Jacobson *et al.*, 1992; Booch, 1994; Coleman *et al.*, 1994; Booch *et al.*, 1997), el movimiento hacia el desarrollo de software basado en componentes requiere que las aproximaciones existentes sean reconsideradas. En concreto, cualquier método que soporte el desarrollo de software basado en componentes debe de cumplir los cuatro principios siguientes (Brown, 1998):

i. una separación clara en la especificación del componente entre su diseño e implementación.

ii. un diseño centrado en interfaces.

iii. una captura formal de la semántica del componente.

iv. un proceso de refinamiento riguroso.

En este contexto, la contribución presentada en esta Tesis se basa en la aproximación OO-Method (Pastor *et al.*, 1997c; Pastor *et al.*, 1998; Pastor *et al.*, 1999a), construida básicamente sobre el modelo formal OO OASIS (Pastor *et al.*, 1992; Pastor & Ramos, 1995), cuya principal característica es que los esfuerzos de los desarrolladores se centran en la fase de modelado conceptual. Concretamente los requisitos del sistema se capturan de acuerdo con un conjunto predefinido y finito de lo que denominamos cons-

tructores de modelado conceptual[1]. La implementación OO completa se obtiene de forma automática guiada por un modelo de ejecución (Gómez *et al.*, 1998; Gómez *et al.*, 1999), que establece las correspondientes representaciones (a nivel de estructura y comportamiento) entre los constructores del modelo conceptual y sus representaciones en un entorno de desarrollo de software concreto.

Una de la principales contribuciones de este capítulo es la propuesta de una arquitectura basada en componentes para el modelo de ejecución de OO-Method (Gómez, 1998a; Gómez, 1998b). Esta propuesta proporciona dos beneficios:

i. una estrategia para obtener componentes software a partir del modelo conceptual. Estos componentes pueden combinarse dinámicamente para construir un prototipo software que preserva la funcionalidad capturada en la fase de modelado conceptual.

ii. un entorno integrado para ejecutar la especificación en el espacio de la solución.

El proceso de diseño e implementación de cualquier aplicación desarrollada con OO-Method, implica tomar decisiones que pueden ser descompuestas de acuerdo a 2 dimensiones diferentes (Gómez & Pastor, 1999a):

- espacio del problema: que representa el dominio del problema que está siendo resuelto (alquiler de coches, biblioteca, hospital, ...).
- espacio de la solución: entorno de desarrollo software elegido para la implementación, y arquitectura del sistema elegida para la aplicación (cliente/servidor, 3-capas, sistema OLTP[2], ...)

Idealmente una decisión tomada respecto a una dimensión no debe de interferir con las decisiones tomadas para la otra. Esta independencia permite a los diseñadores cambiar decisiones previas

[1] Representaciones de conceptos relevantes a nivel del espacio del problema
[2] On-Line Transaction Processing

en alguna dimensión sin tener que reconsiderar las decisiones tomadas en la otra, y por consiguiente alcanzar un grado elevado de flexibilidad en el diseño.

OO-Method permite la independencia entre estas dos dimensiones de la siguiente forma:

- en la dimensión del espacio del problema se obtiene una descripción del problema representada haciendo uso de los modelos gráficos proporcionados por OO-Method, que lleva a un modelo del problema especificado con el lenguaje de especificación OASIS. Esta especificación formal actúa como repositorio de alto nivel del sistema.
- la otra dimensión (espacio de la solución), viene especificada por un modelo de ejecución abstracto que define cómo navegar a través del modelo OASIS, extraer la información relevante y expresarla en una sintaxis adecuada. De esta forma, a un mismo modelo se le pueden aplicar diferentes conjuntos de representaciones produciendo diferentes resultados tales como una implementación del sistema para un lenguaje de programación dado y para una arquitectura determinada.

Por tanto, la idea básica en la propuesta OO-Method para soportar el desarrollo de componentes software, reside en separar claramente lo que denominamos el nivel de modelo conceptual y el nivel de modelo de ejecución. El primer nivel representa el espacio del problema y está centrado en qué es el sistema. El segundo nivel representa el espacio de la solución y especifica cómo se va a implementar el sistema.

Cuatro pasos clave hacen que consideremos la aproximación OO-Method adecuada para la producción de componentes software. Estos pasos se exponen en la Figura 3.1.

El espacio del problema comprende las tareas asociadas a la obtención de la especificación formal de los requisitos que debe de satisfacer el sistema. Este es un proceso de dos pasos; capturar un modelo conceptual y obtener una especificación formal OASIS de forma automática. Esta especificación formal es el punto

```
-----------------------------------------------------------
ESPACIO DEL PROBLEMA (nivel de modelo conceptual)
-----------------------------------------------------------
1.- Capturar un modelo conceptual.
2.- Obtener una especificación formal del modelo conceptual
    de forma automática.
-----------------------------------------------------------
ESPACIO DE LA SOLUCION (nivel de modelo de ejecución)
-----------------------------------------------------------
3.- Generar código OO a partir de la especificación formal
4.- Obtener la especificación de los componentes del sistema
    de forma automática.
-----------------------------------------------------------
```

Figura 3.1. Pasos clave en la propuesta OO-method

de partida de un modelo de ejecución que debe de establecer adecuadamente las características dependientes de la implementación asociadas a la correspondiente representación software de la especificación. Por tanto, el modelo de ejecución establece una estrategia concreta para generar código a partir de la especificación formal OASIS sobre el espacio de la solución.

A continuación se presenta con detalle cada una de estas fases.

3.2 Espacio del Problema

3.2.1 Paso I. Capturar un Modelo Conceptual

Dos aspectos principales (Pastor *et al.*, 1999b), se deben establecer de forma precisa cuando se introduce una aproximación de modelado conceptual:

- qué constructores de modelado conceptual son proporcionados por el método.
- qué notación se proporciona para capturar adecuadamente los constructores de modelado conceptual.

Antes de presentar el conjunto específico de constructores de modelado y sus notaciones, debemos insistir en que la filosofía de la

propuesta OO-Method, reside en combinar técnicas de especificación formales con técnicas de modelado OO convencionales ampliamente usadas en contextos industriales. Uno de los objetivos implícitos de esta decisión fué evitar la complejidad que tradicionalmente se ha asociado al uso de métodos formales, haciendo ver a los modeladores que el método es compatible con los estándares de modelado usados en la industria.

Por todos estos motivos se adoptó la estrategia bien conocida de OMT (Rumbaugh *et al.*, 1991), que consiste en dividir el proceso de modelado conceptual en tres vistas complementarias del sistema, que se corresponden con el modelo de objetos, el modelo dinámico y el modelo funcional. Sin embargo, existe una gran diferencia: en OO-Method, cuando el modelador está especificando el sistema, lo que realmente está haciendo es capturar una especificación formal del mismo, de acuerdo con el lenguaje de especificación formal OASIS. Esta característica permite introducir una expresividad bien definida en la especificación, de la que a menudo carecen los métodos convencionales. Obviamente, se ha tenido que introducir en esos diagramas (modelo de objetos, modelo dinámico y modelo funcional) información relevante para tratar características específicas de OASIS. Sin embargo, esto se realiza preservando en todo momento la vista externa que es compatible con la mayoría de las técnicas de modelado conceptual referenciadas. Sólo la información que es relevante para rellenar una definición de clase en OASIS debe ser introducida. De esta forma el formalismo se oculta al modelador cuando se encuentra describiendo el sistema, causándole la sensación de que está usando una notación conocida de modelado conceptual.

Respecto a la notación, el modelado conceptual de OO-Method utiliza diagramas compatibles con el lenguaje de modelado UML (Pastor *et al.*, 1997a; Insfrán *et al.*, 1997). De acuerdo a los argumentos expuestos anteriormente, uno de los intereses principales en el diseño de OO-Method fué evitar que los modeladores tuviesen que aprender otra notación gráfica para modelar un sistema de información. El disponer de estas bases formales permite proporcionar un entorno de modelado donde el conjunto de diagra-

mas que se necesitan para obtener una especificación del sistema está claramente establecido. A continuación se introducen los conceptos de modelado de OO-Method y los diagramas usados para representarlos.

OO-Method captura las propiedades relevantes del Sistema de Información usando tres modelos complementarios:

Modelo de Objetos. El modelo de objetos (MO) es un modelo gráfico donde se especifican las clases del sistema incluyendo sus atributos, servicios y relaciones entre clases (agregación y herencia). Además, se pueden especificar las denominadas relaciones de agente que permiten establecer qué servicios pueden activar los objetos que pertenecen a una determinada clase. El diagrama básico UML correspondiente es el diagrama de clases, en donde la expresividad adicional es introducida utilizando los estereotipos adecuados.

Vamos a introducir un ejemplo de sistema de información muy simplificado de una biblioteca que nos servirá para ir introduciendo progresivamente los distintos conceptos de modelado que aparecen en los diagramas (en el apéndice A se presenta un caso real mucho más completo sobre el que ha sido aplicado el método que a continuación se presenta):

"El sistema de información de una biblioteca debe de permitir dar de alta socios y libros. Los socios podrán tomar prestados libros de la biblioteca. La información que se requiere de un socio es su número de socio, un nombre y el número de libros que tiene prestados. De cada libro se desea conocer su código, título, autor si está o no disponible, y el lugar que ocupa en la biblioteca. Los libros se prestan a los socios como consecuencia de un préstamo. Un préstamo estará caracterizado por el código del libro prestado, el número de socio y la fecha del préstamo. Un socio puede desempeñar un rol de socio no fiable cuando la fecha del préstamo expira. A los socios no fiables no se les permite pedir prestados libros hasta que dejan de serlo. Un lector no podrá tener más de 10 libros prestados, en cuyo caso también pasará a desempeñar un rol de socio no fiable."

La Figura 3.2, muestra un ejemplo de una instancia del MO asociado al sistema de información de la descripción anterior. Las clases de objetos se representan como rectángulos con tres áreas: nombre de clase, atributos y servicios. Las relaciones de herencia se representan mediante flechas que conectan clases. Por ejemplo, la flecha que apunta desde socio no fiable a socio denota que socio no fiable es una especialización de socio. Las relaciones de agregación se representan con un rombo en la clase compuesta del que parten líneas que enlazan con las clases componentes. Por ejemplo, el rombo asociado a la clase préstamo denota una relación de agregación que incluye la cardinalidad (mínima y máxima), donde préstamo es la clase compuesta y libro y socio son las clases componentes. Las líneas discontinuas sirven para especificar las relaciones de agente, o dicho de otro modo, quién puede activar cada servicio de clase (relaciones cliente/servidor). En este ejemplo concreto, los objetos de la clase socio pueden activar los servicios prestar y devolver de la clase libro. Las líneas continuas representan lo que denominamos servicios compartidos, prestar y devolver son servicios compartidos entre las clases libro y socio.

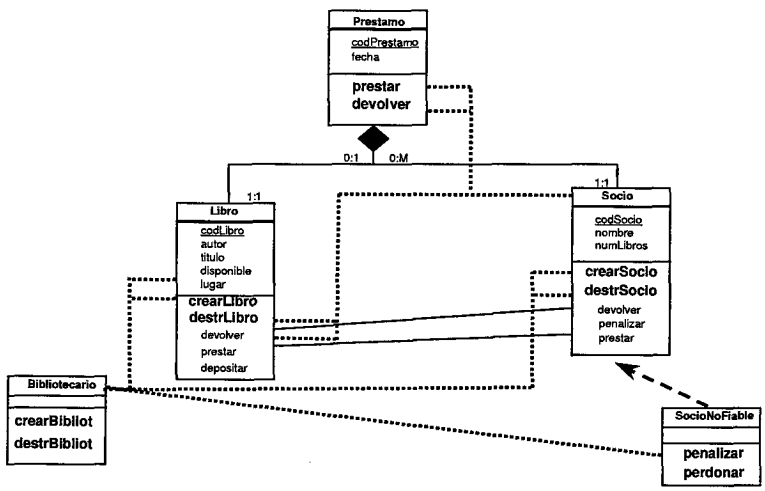


Figura 3.2. *Modelo de objetos del sistema biblioteca.*

Además es necesario especificar información adicional en el MO para completar la descripción formal de cada clase. Concretamente, para cada clase del MO se captura la siguiente información:

item	descripción
atributos	tipo de atributo (constante o variable)
servicios	nombre de los servicios con sus correspondientes argumentos
derivaciones	expresiones para el cálculo de atributos derivados (aquellos cuyo valor es dependiente de otros atributos)
restricciones	fórmulas bien formadas que fijan condiciones que los objetos de una clase deben satisfacer
relaciones	información específica asociada a jerarquías de agregación y herencia

Más precisamente, la información asociada con herencia y agregación es la siguiente:

- para clases agregadas, se especifica si es una asociación o una composición (siguiendo la caracterización de UML), y si la agregación es estática o dinámica.
- para jerarquías de herencia, se especifica si la especialización es permanente o temporal. En el primer caso, se caracteriza la condición correspondiente sobre los atributos contantes; en el segundo, se especifica una condición sobre los atributos variables o el servicio portador que activa el rol del hijo.

Modelo Dinámico. Con el MO se especifica la arquitectura de clases del sistema. Determinadas características básicas tales como qué ciclos de vida se consideran válidos para los objetos o qué tipo de comunicación interobjetual puede establecerse también ha de ser introducida en la especificación del sistema. OO-Method permite la introducción de estas características a través de un modelo dinámico que usa dos clases de diagramas:

DIAGRAMA DE TRANSICION DE ESTADOS. El diagrama de transición de estados (DTE) se usa para describir el comportamiento de forma correcta estableciendo los ciclos de vida válidos

para los objetos de cada clase. Por vida válida se entiende una secuencia correcta de estados que caracteriza un comportamiento correcto de los objetos que pertenecen a una clase.

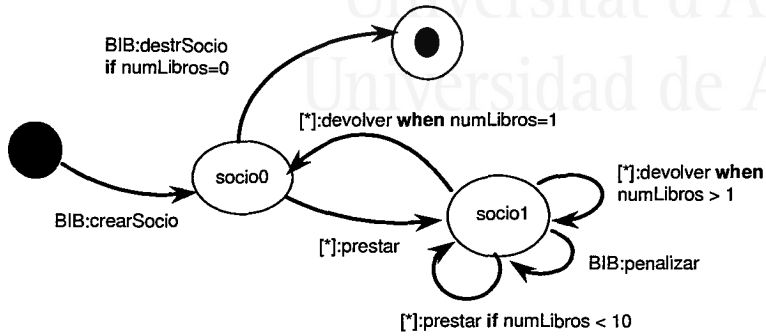


Figura 3.3. *DTE para la clase socio.*

La Figura 3.3, muestra un DTE simple para la clase socio. El diagrama básico UML correspondiente es el diagrama de estados. Cuando un objeto no existe, un círculo negro representa el estado de noexistencia y será la fuente de la transición inicial etiquetada por la acción correspondiente (ej. BIB:crearSocio).

Cada ocurrencia de servicio (crearSocio) se etiqueta por el agente (BIBliotecario) que puede activarlo. En este ejemplo, el * denota que cualquier agente válido del sistema puede activar la transición. Como en este caso el único agente válido para crearSocio son los objetos de la clase BIBLIOTECARIO (según lo especificado en el MO), ambas representaciones son equivalentes.

Las transiciones representan cambios válidos de estado que pueden ser restringidos introduciendo condiciones de acuerdo a la sintaxis mostrada en la Figura 3.4.

```
servicio [if precondición] [when condición de control]
```

Figura 3.4. *Sintaxis para las transiciones en el DTE*

Las precondiciones son condiciones definidas sobre los atributos de los objetos que se deben de cumplir para que un servicio ocurra. Las condiciones de control son condiciones definidas sobre los atributos de los objetos para evitar el posible no determinismo para una acción dada. En el ejemplo si el socio se encuentra en el estado socio0 y se activa el servicio prestar, el objeto pasará al estado socio1. Desde aquí, si acontece el servicio devolver, la transición seleccionada será aquella que satisfaga la condición de control correspondiente (numLibros = 1 or numLibros > 1) en el estado del objeto implicado.

DIAGRAMA DE INTERACCION. El Diagrama de Interacción (DI), especifica la comunicación interobjetual. Definimos dos tipos básicos de interacción: *disparos*, que son servicios de un objeto que son activados de forma espontánea cuando se satisface una condición, e *interacciones globales*, que son transacciones que engloban a servicios de diferentes objetos.

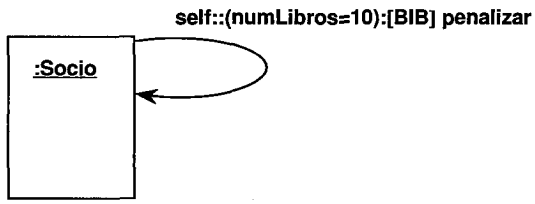


Figura 3.5. *Diagrama de Interacción mostrando un disparo.*

Hay un DTE por cada clase, pero únicamente un DI para todo el sistema en el que se especifican gráficamente las interacciones objetuales. La Figura 3.5, muestra un DI simple para nuestro ejemplo. En este caso el diagrama UML usado para capturar este tipo de expresividad es el diagrama de colaboración. Las cajas etiquetadas con un nombre subrayado identifican a un objeto representante de la clase. En el ejemplo de la Figura 3.5, se especifica un disparo que indica que el servicio penalizar debe de ser activado de forma automática cuando el atributo de la clase socio (numLibros) es igual a 10. La especificación de los disparos sigue la sintaxis que se muestra a continuación:

```
destino :: (condición de disparo) : servicio
```

Figura 3.6. *sintaxis de una especificación de disparo*

El primer componente es el destino (el objeto/s proveedor del servicio a disparar). El destino puede ser el mismo objeto que satisface la condición (disparo a self), un objeto concreto si se especifica el OID del objeto involucrado (disparo a object), o todos los objetos de una clase si se trata de un servicio de difusión[3] a todos los objetos de la clase (disparo a class). El segundo componente es la condición que, al cumplirse en el estado del objeto considerado, originará el disparo de la solicitud del servicio correspondiente, que es el introducido como tercer componente.

Las interacciones globales se especifican gráficamente enlazando los servicios involucrados en la declaración de la interacción. Estos servicios se representan mediante flechas que enlazan los objetos que los proporcionan de forma análoga a como se especifican los disparos.

Modelo Funcional. El modelo funcional (MF) se utiliza para especificar el efecto que tiene la ejecución de un servicio sobre el estado de un objeto. La especificación de ese tipo de propiedades es un problema resuelto de forma imprecisa por la inmensa mayoría de los métodos OO. Algunos de ellos simplemente obvian el asunto, dejando en manos del modelador la libre definición de la semántica de los servicios en cualquier lenguaje sin formato, y otros (en el otro extremo) permiten introducir esta información utilizando la sintaxis de un lenguaje de programación específico (por ej. C^{++}). Propuestas como la de OMT (Rumbaugh *et al.*, 1991), proponen utilizar un modelo funcional basado en los Diagramas de Flujos de Datos (DFDs) del Análisis Estructurado. El uso dado frecuentemente a esta técnica (en cualquiera de sus variantes) en el contexto de un modelo objetual ha sido muy criticado por ser demasiado operacional e impreciso, además de ofrecer una perspectiva del sistema que difiere (en muchos casos) total-

[3] Broadcast

mente de los otros dos modelos (objetos y dinámico). El modelo de objetos y el modelo dinámico particionan el sistema en objetos con un estado y un comportamiento local. Por el contrario, el modelo funcional separa estado y comportamiento particionando el sistema en almacenes de datos y procesos. Un almacén de datos puede contener un fragmento del estado de todas las instancias de una clase, y un proceso puede llevar a cabo parte o todo el proceso de una operación. De todo esto podemos sacar la conclusión de que difícilmente se puede establecer una relación entre el modelo funcional típico y los otros dos modelos.

En general, una forma alternativa de especificar el efecto de un servicio es declarativamente como se propone en OASIS (Pastor & Ramos, 1995; Letelier *et al.*, 1998) mediante el uso de la lógica dinámica (Harel, 1984) o en TROLL (Grau *et al.*, 1998) mediante el uso de la lógica temporal distribuida (Ehrich *et al.*, 1998).

En particular, en el MF de OO-Method se especifica la funcionalidad del sistema sin necesidad de utilizar explícitamente la notación formal de OASIS, dotando a este modelo de un esquema declarativo para recoger toda la información que describe los efectos de los servicios sobre los objetos del sistema. La utilización de este esquema asume el conocimiento de determinados conceptos inherentes a OO-Method, como son la categorización de las evaluaciones y las condiciones de evaluación que determinan el efecto que sobre un atributo tiene la ocurrencia de un servicio.

El MF de OO-Method proporciona una estrategia clara y simple para especificar los cambios de valores de los atributos de un objeto en función de 3 aspectos:

- el efecto de un servicio
- las condiciones de evaluación
- las categorías de evaluaciones

El modelador debe ser capaz de definir con claridad y facilidad cómo afecta la ocurrencia de un servicio al valor de cada uno de los atributos del objeto.

Cada atributo variable del objeto tendrá asociado un conjunto de servicios relevantes, extraídos de los especificados en la signatura de su clase correspondiente, que serán los que podrán hacer que su valor cambie (y, recíprocamente todo servicio estará relacionado con un conjunto de atributos variables a los que modifica).

Las condiciones de evaluación son condiciones sobre los atributos de un objeto que son evaluadas en el estado en el que es activado un servicio relevante. Estas condiciones determinan que el efecto de una acción sobre el valor de dichos atributos sea uno u otro dependiendo del estado del objeto.

Un atributo variable puede ser modificado por varios servicios (al menos uno) y un servicio puede modificar diversos atributos. Para describir la forma en que un servicio puede modificar el valor de un atributo se definen tres categorías: situación, estado y cardinal.

Teniendo en cuenta el modo en el que los servicios modifican el valor de un atributo en una evaluación, podemos hacer una clasificación de las evaluaciones que nos facilite el proceso de descripción de la semántica del cambio de estado.

Las categorías de evaluaciones son:

- Evaluaciones cardinales: el servicio tiene un efecto de incremento o decremento cuantificable a través de una función sobre el valor del atributo evaluado, los argumentos de dichos eventos y los atributos del objeto. El efecto del servicio es dependiente del valor del atributo en el estado anterior a la ocurrencia de éstos. El incremento / decremento en una unidad es un caso especial que puede asumirse por defecto.

La Figura 3.7, ilustra el efecto de los servicios prestar y devolver sobre el atributo numLibros de la clase socio. En este caso la evaluación que se produce sobre el atributo es una evaluación cardinal, es decir, el valor del atributo se verá incrementado o decrementado tras la ejecución de alguno de estos dos servicios.

- Evaluaciones de estado: son aquellas que tienen como propiedad característica que cuando ocurre un servicio relevante, el atri-

CLASE: Socio ATRIBUTO: numLibros

Servicio	Condición de evaluación	Efecto del servicio
prestar		= numLibros + 1
devolver		= numLibros - 1

Figura 3.7. *Evaluaciones cardinales.*

buto al que afecta adquiere un valor que es independiente de los valores que haya tenido anteriormente. Ese nuevo valor del atributo se calculará a partir de algún argumento del evento relevante correspondiente.

CLASE: Libro ATRIBUTO: lugar

Servicio	Cond. evaluación	Efecto del servicio
depositar(ubicación)		= ubicación

Figura 3.8. *Evaluaciones de estado.*

Como ejemplo, en la Figura 3.8 se ilustra una evaluación de estado sobre el atributo lugar de la clase libro. El atributo representa la estantería donde se puede localizar un libro en la biblioteca. El efecto de la ejecución del servicio depositar (que recibe como parámetro la ubicación donde se coloca el libro), provocará que el nuevo valor del atributo lugar se corresponda con el valor del argumento ubicación.

- Evaluaciones de situación: son aquellas en las que el atributo afectado toma valor en un dominio discreto.

  Cada uno de los valores que puede tomar el atributo afectado modela las posibles situaciones alcanzables por el atributo. A través de la ocurrencia de servicios portadores el atributo al-

canza una determinada situación. En este tipo de evaluaciones el nuevo valor del atributo también puede depender en cierto modo del valor que tenía en el estado anterior a la ocurrencia del evento.


```
CLASE: Libro  ATRIBUTO: disponible

------------------------------------------------------------
Servicio    Condición de evaluación    Efecto del servicio
------------------------------------------------------------
prestar                                = FALSE
devolver                               = TRUE
------------------------------------------------------------
```

Figura 3.9. *Evaluaciones de situación.*

Un ejemplo muy evidente de evaluaciones de situación se produce en el caso de atributos de tipo booleano. En la Figura 3.9 se especifica el modelo funcional del atributo disponible de la clase libro. Este atributo contiene información acerca de si un libro está o no disponible en la biblioteca. Los posibles valores que puede tomar el atributo pertenecen a un dominio discreto (TRUE o FALSE), su valor por tanto dependerá en este caso de si el libro ha sido o no prestado.

La categorización de los atributos es una contribución relevante del método OO-Method. Los cambios de valores de los atributos de un objeto se especifican a través de una estrategia simple que de esta forma permite generar una especificación OASIS completa de forma automática. Información más detallada puede consultarse en (Pastor *et al.*, 1997c).

3.2.2 Paso II. Obtener la Especificación Formal OASIS

Tomando como punto de partida los tres modelos anteriores y utilizando una estrategia de traducción precisa, se obtiene de forma automática la correspondiente especificación formal OO OASIS. La especificación OASIS resultante actúa como repositorio de alto

nivel del sistema. Esta es una característica principal en la aproximación OO-Method: cada pieza de información introducida en el paso de modelado conceptual tiene su contrapartida formal representada como un concepto OASIS. De esta forma se puede ver OO-Method como un editor gráfico avanzado para especificaciones OASIS.

El trabajo que se presenta en esta tesis utiliza la versión 2.2 del lenguaje de especificación OASIS. Aunque durante éstos últimos años en el contexto de nuestro grupo de investigación se ha desarrollado una versión superior del lenguaje (versión 3.0) (Letelier *et al.*, 1998), la razón fundamental de usar la versión anterior viene motivada por el hecho de que en el momento de empezar este trabajo era la última disponible.

Para tener una visión completa de la aproximación que aquí se presenta, a continuación se introducen los conceptos formales asociados al lenguaje de especificación OASIS versión 2.2 (Pastor & Ramos, 1995) utilizando como contexto la especificación de la clase socio de nuestro ejemplo.

Conceptos Preliminares. Una especificación OASIS es un conjunto estructurado de definiciones de clase. La figura 3.10, muestra la especificación OASIS resultante para la clase socio que se obtiene automáticamente tras finalizar el paso de modelado conceptual.

Una clase se compone de un nombre de clase (socio), uno o más mecanismos de identificación para instancias de la clase y un tipo o plantilla que comparten todas las instancias. La plantilla recoge todas las propiedades (estructura y comportamiento), compartidas por todos los objetos potenciales de la clase considerada.

Vamos a proporcionar una definición informal de plantilla de clase y dejaremos la definición formal para el capítulo 5 en el que demostraremos aspectos de equivalencia semántica entre especificaciones OASIS e implementaciones imperativas.

```
CONCEPTUAL SCHEMA biblioteca
domains nat,bool,int,date,string

class socio
identification
    by_codSocio: (codSocio);
constant_attributes
    edad : String ;
    codSocio : String ;
    nombre : String ;
variable_attributes
    numLibros : Int ;
private_events
    crearSocio new;
    destrSocio destroy;
    penalizar;
shared_events
    prestar with book;
    devolver with book;
constraints
    static numLibros < 10;
valuations
    [prestar]  numLibros = numLibros + 1;
    [devolver] numLibros = numLibros - 1;
preconditions
    bib:destrSocio () if
    numLibros = 0 ;
triggers
    Self :: penalizar if numLibros = 10;
process
    socio = bib:crearSocio socio0;
    socio0= bib:destrSocio + prestar socio1;
    socio1= if numLibros=1 devolver socio0
                + (if numLibros > 1 devolver
                + if numLibros < 10 prestar) socio1;
end_class

END CONCEPTUAL SCHEMA
```

Figura 3.10. Especificación OASIS para la clase socio

Una plantilla de clase OASIS se caracteriza mediante una estructura que posee un conjunto de atributos, un conjunto de eventos, un conjunto de fórmulas y una especificación de proceso.

Los atributos son pares de la forma (nombre, sort de evaluación), donde el nombre representa el identificador del atributo y el sort de evaluación el nombre del conjunto de los valores (o conjunto soporte) que puede tomar dicho atributo. El conjunto de valores

de los atributos (atributos evaluados), determina el estado de los objetos pertenecientes a la clase que define la plantilla. Los atributos (ver secciones constant_attributes y variable_attributes en la Figura 3.10), pueden tomar valor en el momento de creación del objeto y no cambiar durante la vida del mismo (atributos constantes), o pueden venir definidos por ciertas reglas que expresan cómo varían con la ocurrencia de eventos (atributos variables), o pueden obtener su valor a partir de otros atributos mediante ciertas reglas de derivación (atributos derivados).

Los eventos representan abstracciones de cambios de estado. El conjunto de eventos determina el comportamiento de los objetos pertenecientes a la clase que define la plantilla. Los eventos se caracterizan como propios cuando participan en la vida de las instancias de una única clase de objetos y como compartidos en caso de que participen en las vidas de instancias de varias clases de objetos (ver secciones private and shared events en la Figura 3.10).

Los eventos pueden agruparse formando unidades de comportamiento moleculares denominadas transacciones. Las transacciones tiene dos propiedades importantes: primero, cuando se inicia una transacción se ejecuta completamente o sino no se ejecuta y segundo, los estados por los que pasan los objetos involucrados en la transacción no son observables.

El conjunto de fórmulas de la plantilla responden a un subconjunto de la lógica dinámica (Harel, 1984) y se deglosan en los siguientes tipos de fórmulas dinámicas:

- Evaluaciones; son fórmulas de la forma $\Phi[e]\Phi'$ cuya semántica viene dada por una función ρ que, a partir de un evento e devuelve una función entre mundos posibles.

  $$\rho(e) \in W \rightarrow W \tag{3.1}$$

  Dicho de otra forma, son fórmulas que definen cambios de estado ante la ocurrencia de eventos. En el ejemplo de la especificación de la Figura 3.10, aparecen las siguientes evaluaciones:

$$[prestar]\ numLibros = numLibros + 1; \tag{3.2}$$

$$[devolver]\ numLibros = numLibros - 1; \tag{3.3}$$

Es importante resaltar el hecho de que en el contexto de la lógica dinámica, la fórmula Φ se evalúa en $s \in W$, y Φ' se evalúa en $\rho(e)(s)$, siendo $\rho(e)(s)$ el mundo alcanzado por el objeto después de la ejecución en s de la acción considerada.

- Derivaciones son fórmulas del tipo $\Phi \rightarrow \Phi'$. Permiten definir atributos derivados (Φ') en términos de una condición de derivación dada (declarada en Φ).
- Restricciones de integridad; son fórmulas que deben satisfacerse en todos los mundos. OASIS distingue entre restricciones estáticas y dinámicas.

  Las restricciones estáticas son un caso particular de restricciones dinámicas puesto que están definidas para todo mundo posible y deben de cumplirse siempre. En el ejemplo:

  $$static\ numLibros < 10; \tag{3.4}$$

  Por otra parte, las restricciones dinámicas son aquellas que relacionan mundos diferentes. Requieren el uso de una lógica temporal, con los correspondientes operadores de dicha lógica (ver Capítulo 5).

- Precondiciones, son fórmulas con el esquema $\neg\Phi[e]false$, donde Φ es una fórmula que debe de cumplirse en el mundo anterior a la ejecución del evento e. Sólo en los mundos donde Φ se cumple, se permite que ocurra e. Si $\neg\Phi$ se cumple, la ocurrencia de e no genera ningún estado sucesor válido. En el ejemplo se especifica la siguiente precondición:

  $$\neg(numLibros = 0)\ [bib : destrSocio]\ false; \tag{3.5}$$

  o de forma más conveniente a efectos de que la especificación sea más legible, se puede escribir:

$$bib : destrSocio\ if\ numLibros = 0; \tag{3.6}$$

- Disparos. Son fórmulas de la forma $\Phi[\neg e]false$. Donde $\neg e$ es la negación de la acción[4]. Si se satisface Φ y ocurre otro evento distinto a a, entonces no hay estado sucesor. Esto fuerza a que e ocurra o el sistema permanece en un estado bloqueado. Por ejemplo, de acuerdo al DI presentado en la Figura 3.5 tenemos:

$$numLibros = 10\ [\neg(Self :: penalizar)]\ false; \tag{3.7}$$

que de nuevo se puede escribir de forma equivalente y más convencional para facilitar la legibilidad de la especificación:

$$Self :: penalizar\ if\ numLibros = 10; \tag{3.8}$$

Por consiguiente, los disparos son eventos activados cuando las condiciones declaradas en Φ se cumplen. La diferencia principal entre precondiciones y disparos viene dada por el hecho de que los disparos llevan implícita la obligación de activar el evento tan pronto como la condición sea satisfecha. De esta forma permiten introducir actividad interna en la sociedad de objetos que está siendo modelada.

En cualquiera de estas fórmulas dinámicas, Φ, Φ' son fórmulas bien formadas en la lógica de primer orden usada.

El último elemento que integra una plantilla de clase OASIS es la especificación de proceso. Una especificación de proceso (Ramos *et al.*, 1992), se construye a partir del conjunto de eventos usando ciertos operadores del álgebra de procesos como los operadores de secuencia ($\bullet$), de alternativa ($+$) y de recursión en cola. Estos representan la parte funcional del álgebra de procesos usada para describir las vidas posibles de los objetos. De esta forma, el conjunto de trazas válidas o secuencias de eventos permitidas se hacen corresponder con términos de ese álgebra. La especificación de proceso así construida permite caracterizar las vidas válidas de

[4] Significa que a no ocurre, y no se especifica que ocurre

un objeto mediante una traza correcta de eventos que pertenecen a su signatura.

A continuación se muestra la especificación de proceso para la clase socio.

$$
\begin{aligned}
&socio = bib : crearSocio \bullet socio0;\\
&socio0 = bib : destrSocio + prestar \bullet socio1;\\
&socio1 = if\ numLibros = 1\ devolver \bullet socio0\\
&+ (if\ numLibros > 1\ devolver\\
&+ if\ numLibros < 10\ prestar) \bullet socio1;
\end{aligned}
\tag{3.9}
$$

Los estados de proceso socio, socio0 y socio1 representan clases de equivalencia que caracterizan los estados de los objetos. En este sentido, un estado de proceso contemplará varios estados del objeto. La parte derecha siempre establece secuencias de eventos válidos que pueden acontecerle a los objetos que cumplen la especificación de proceso.

En el anexo B se presenta la sintaxis de la versión del lenguaje OASIS utilizada en este trabajo. Una descripción más detallada de OASIS se puede encontrar en (Pastor *et al.*, 1992), y la descripción completa de su semántica y fundamentos formales en (Pastor & Ramos, 1995; Letelier *et al.*, 1998).

La Especificación como Repositorio del Sistema. Una vez presentados los constructores de modelado de OO-Method y los conceptos formales OASIS asociados a ellos, a continuación se introducen las correspondencias que permiten obtener una representación textual del sistema, esto es la especificación OASIS. El punto de entrada de este proceso es la información gráfica introducida en el modelo conceptual. Como veremos en la sección 3.3 de este capítulo, la especificación formal que se obtiene constituye una documentación sólida del sistema para generar un producto final software que es compatible con los requisitos iniciales representados en el modelo conceptual.

Dicha especificación formal se obtiene automáticamente a través de un proceso de conversión de cada constructor de modelado

usado en el modelo conceptual en su contrapartida formal sobre el lenguaje de especificación OASIS.

De acuerdo a la plantilla de clase presentada en la Figura 3.10, a continuación se identifican el conjunto de conceptos de modelado de OO-Method y sus correspondientes representaciones OASIS.

Las clases del sistema se obtienen del MO. Para cada clase se obtiene:

- su conjunto de atributos (contantes, variables y derivados).
- su conjunto de servicios, incluyendo eventos (privados y compartidos) y transacciones locales.
- las restricciones de integridad especificadas para la clase.
- las expresiones de derivación que corresponden a atributos derivados.

Para el caso de las clases complejas (aquellas definidas utilizando los operadores de agregación y herencia), el MO proporciona las características particulares especificadas para la correspondiente clase compleja. Concretamente, en el caso de la agregación, la cardinalidad (mínima y máxima) entre las clases compuestas y sus componentes proporciona el tipo de agregación. En el caso de la herencia, como se comentó anteriormente, el tipo de especialización (permanente o temporal) se determinará a partir de la condición de especialización.

Con la información proporcionada por el MO, se captura la parte estática del sistema. Para capturar la parte dinámica OO-Method dispone en el MD de dos diagramas: el DTE y el DI. El DTE proporciona un mecanismo para representar:

- las precondiciones de los servicios (aquellas fórmulas que etiquetan las transiciones de estado).
- la definición de proceso de la clase que establece las secuencias válidas de cambios de estado que pueden sufrir los objetos que pertenecen a una clase.

Con lo cual se garantiza que el proceso de diseño de las secciones relacionadas con las precondiciones y la especificación de proceso (secciones preconditions y process en la Figura 3.10), se realice de forma automática a partir de este diagrama.

Del DI se obtienen las últimas dos características necesarias para completar una especificación de clase OASIS:

- relaciones de disparo
- transacciones globales (aquellas que involucran servicios de diferentes clases de objetos).

Por último, el MF proporciona las fórmulas dinámicas asociadas con las evaluaciones en donde se especifica el efecto de los servicios sobre los atributos.

Una vez definido el conjunto de información relevante que puede ser introducido en un modelo conceptual OO-Method, la especificación formal correspondiente proporciona un repositorio del sistema fiable que captura completamente la descripción del sistema según el modelo OASIS. Esto permite abordar el proceso de implementación (modelo de ejecución) desde un punto de partida bien definido. Las piezas de información involucradas tienen un significado concreto puesto que provienen de un conjunto de conceptos de modelado conceptual que además, como hemos comentado anteriormente, tienen una contrapartida formal en el lenguaje OASIS.

3.3 Espacio de la Solución

3.3.1 Paso III. Generar el Código OO

La especificación OASIS obtenida es el punto de partida para un modelo de ejecución (ME) que establece las características dependientes de la implementación asociadas a la representación final del Sistema de Información en el espacio de la solución.

A continuación se describe con detalle el modelo de ejecución y la arquitectura software que va a permitir obtener un producto software en el espacio de la solución.

El Modelo de Ejecución. Para implementar y animar el sistema especificado, el modelo predefine una estrategia de ejecución mediante la cual los usuarios interactúan con el sistema de objetos. Para ello, introduce una nueva forma de interacción que podríamos denominar como la *realidad virtual OO*, en el sentido de que un objeto activo 'se introduce' en la sociedad de objetos e interactúa con otros objetos de dicha sociedad. El modelo de ejecución utiliza la plantilla mostrada en la Figura 3.11 para alcanzar este comportamiento:

```
1. Identificar al usuario
2. Obtener la vista del sistema para ese usuario
3. Permitir la activación de servicios
        3.1 Identificar el objeto servidor
        3.2 Introducir los argumentos del servicio
        3.3 Enviar el mensaje al objeto servidor
        3.4 Comprobar transición del estado
        3.5 Comprobar precondiciones
        3.6 Realizar las evaluaciones
        3.7 Comprobar restricciones de integridad en el nuevo estado
        3.8 Realizar la comprobación de disparos
```

Figura 3.11. Plantilla del Modelo de Ejecución

El proceso comienza conectando al usuario al sistema (*paso 1*) y proporcionando la vista del sistema para ese objeto (*paso 2*) determinando el conjunto de clases y servicios que ese usuario puede activar.

Una vez que el usuario está conectado y tiene una vista del sistema precisa, puede activar cualquier servicio disponible en su vista. Entre estos servicios se encuentran: observaciones (consultas sobre el estado de los objetos), servicios simples y transacciones servidas por otros objetos.

Cualquier activación de servicio (*paso 3*) consiste en dos acciones:

- construir el mensaje y,
- ejecutarlo (si es posible).

Para construir el mensaje, el usuario tiene que proporcionar información para identificar al objeto servidor (*paso 3.1*). La existencia del objeto servidor es una condición implícita para ejecutar cualquier servicio a menos que se trate del servicio new[5], e introducir los argumentos del servicio (*paso 3.2*) pidiendo aquellos necesarios para el servicio que está siendo activado (si se necesitan).

Una vez que el mensaje ha sido enviado (*paso 3.3*), la ejecución del servicio se caracteriza por la ocurrencia de la siguiente secuencia de acciones en el objeto servidor:

- comprobar el DTE (*paso 3.4*) es el proceso que verifica en la especificación OASIS que existe una transición de estado válida para el servicio seleccionado en el estado actual del objeto.
- comprobar las precondiciones (*paso 3.5*) indica que las precondiciones asociadas al servicio que se intenta activar se deben de cumplir.

Si alguna de estas comprobaciones falla, se produce una excepción y se ignora la ejecución del mensaje. En otro caso, el proceso continúa con:

- la realización de las evaluaciones (*paso 3.6*) donde las modificaciones inducidas por la ejecución del servicio (especificadas en la sección *valuations* de la especificación OASIS), tienen lugar sobre el estado actual del objeto.
- para asegurar que la ejecución del servicio deja al objeto en un estado válido, la comprobación de las restricciones de integridad (*paso 3.7*) se verifica para el nuevo estado del objeto.

[5] Formalmente, el servicio new es un servicio de un metaobjeto que representa la clase. El metaobjeto (Canós, 1996; Carsí, 1999) actúa como una factoría de objetos para crear instancias individuales de clase. Este metaobjeto (uno por cada clase) tiene el atributo de población de clase como propiedad principal, el siguiente identificador de objeto y el servicio new

- además, después de un cambio válido de estado hay que comprobar si se cumple alguna regla del conjunto de reglas condición-acción (disparos) que representan la actividad interna del sistema. Si eso ocurre, el servicio asociado se disparará (*paso 3.8*).

Los pasos descritos guían la implementación de cualquier programa a partir de la especificación del sistema capturada en el modelo conceptual.

A continuación se describe como se enlaza la especificación formal capturada en el paso de modelado conceptual con el modelo de ejecución abstracto presentado anteriormente. Primero se propone una arquitectura específica como marco de producción del software generado y posteriormente se presentan las representaciones de los conceptos OASIS en un entorno imperativo OO de desarrollo de software (en este caso el lenguaje de programación Java).

La Arquitectura del Modelo de Ejecución. Una arquitectura común para el desarrollo de SI es la denominada de tres capas (Schulte, 1995). Su característica principal es la separación de la lógica de la aplicación respecto al resto del software, en una capa media que controla la lógica del sistema. De esta forma la capa de presentación queda relativamente libre de procesamiento de aplicación, redirigiendo las peticiones del usuario a la capa intermedia. La capa intermedia se comunica con el sistema de persistencia alojado en la capa de almacenamiento.

La arquitectura propuesta para el ME es una variante extendida de la arquitectura clásica 3-capas. En ella se distinguen ciertas responsabilidades que se asignan a la capa de lógica. En la Figura 3.12 se ilustra la arquitectura del modelo de ejecución. Inicialmente se siguen diferenciando 3 capas básicas. La *capa de presentación* (también llamada capa de interface de usuario), integra los objetos que manejan la presentación de la información entre el usuario y la aplicación. La novedad se presenta en la *capa de lógica* la cual se encuentra subdividida en dos subcapas:

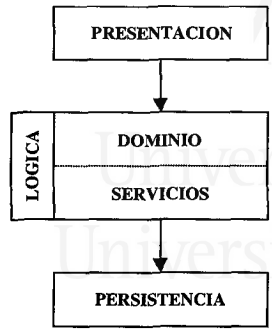


Figura 3.12. *Arquitectura multicapa para el modelo de ejecución.*

- *subcapa objetos del dominio*: contiene objetos que representan conceptos del dominio y que satisfacen los requisitos de acuerdo a la especificación OASIS capturada.
- *subcapa servicios*: son objetos que no pertenecen al dominio del problema pero que proporcionan servicios de interacción con la base de datos. Este tipo de objetos se les denomina comúnmente mediadores[6] (Wiederhold, 1992; Brown & Whitenack, 1996).

Finalmente, la *capa de persistencia* continua representando el sistema seleccionado para dotar de persistencia a los objetos del dominio. A continuación se proporciona una descripción más detallada de cada una de estas capas, y se muestra cómo se representan los conceptos capturados en el espacio del problema (OO-Method/OASIS) sobre el espacio de la solución (utilizaremos como lenguaje de programación para ilustrar estas representaciones el lenguaje Java).

CAPA DE PRESENTACION. Esta capa está integrada por un conjunto de componentes complementarios que no son usados explícitamente en el modelo conceptual pero que ayudan a implementar la interacción entre el usuario y la aplicación, siguiendo la semántica subyacente al modelo de ejecución cuya estructura se presentó en la Figura 3.11. El interface de usuario se construye con 3 componentes principales: controlDeAcceso, vistaDelSistema y activacionDeServicio.

[6] mediators o database brokers

```
public interface IAcceso {
      public boolean validarAcceso (String
      identificadorDeObjeto, String nombreClase);
}

public class controlDeAcceso implements IAcceso {...}
```

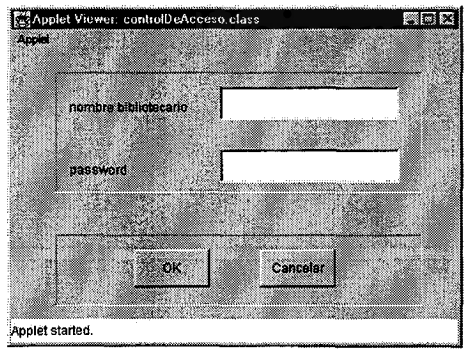


Figura 3.13. *Instancia del componente* controlDeAcceso.

El componente controlDeAcceso (ver figura 3.13), proporciona el interface IAcceso que permite que un usuario pueda identificarse en el sistema. En concreto el interface de este componente proporciona el servicio validarAcceso que captura las propiedades[7] identificadorDeObjeto y nombreClase, necesarias para identificar adecuadamente un objeto como miembro del sistema que está siendo ejecutado.

El componente vistaDelSistema define el interface ISistema que proporciona los servicios necesarios para obtener la vista del sistema particular para un objeto específico. Los servicios asociados al interface proporciona la funcionalidad necesaria para obtener las clases (servicio obtenerListaClases en la Figura 3.14), con las que el usuario puede interactuar y para cada una de ellas qué servicios (ver obtenerListaServicios en la Figura 3.14), puede activar. La representación visual de este componente es un frame que dispone de una barra de menú para representar las clases e items para

[7] Adicionalmente se define otra propiedad denominada password, que nada tiene que ver con la filosofía del modelo de ejecución, pero que permite introducir un mecanismo de seguridad en el sistema, de tal forma que sólo podrán conectarse al sistema objetos (usuarios) válidos

```
public interface ISistema {
  public abstract String[] obtenerListaClases (String claseAgente);
  public abstract String[] obtenerListaServicios (
  String claseServidor, String claseCliente);
}

public class vistaDelSistema implements ISistema {...}
```

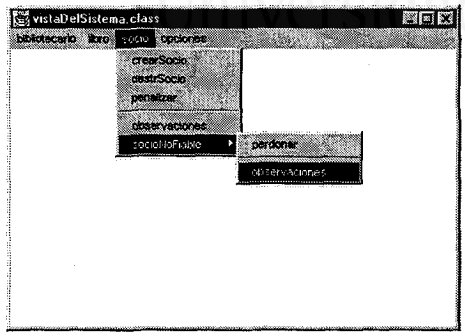


Figura 3.14. *Instancia del componente* vistaDelSistema.

representar los servicios. Es importante resaltar el hecho de que las relaciones de agente especificadas en el MO de OO-Method permiten obtener la información necesaria para inicializar este componente con la vista del sistema concreta para un usuario.

Por último, el componente activacionDeServicio proporciona el interface IServicios que ofrece los servicios necesarios para obtener los argumentos para una activación de servicio concreta (ver obtenerListaDeArgumentos en la figura 3.15).

CAPA DE LOGICA. El diseño de la capa lógica proporciona dos contribuciones que esta tesis realiza al estado del arte:

- una estrategia concreta para traducir la especificación formal del sistema en una implementación imperativa OO.
- un conjunto de servicios para tratar la persistencia de los objetos del dominio.

Como se comentó anteriormente, esta capa se encuentra segmentada en dos subcapas.

```
public interface IServicios {
    public abstract String[] obtenerListaDeArgumentos(
    String claseServidor, String nombreServicio);
}

public class activacionDeServicio implements IServicios {...}
```

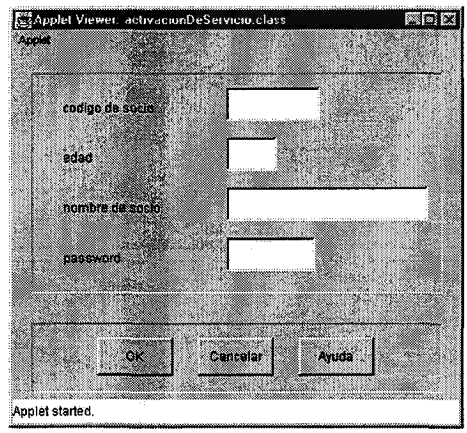


Figura 3.15. *Instancia del componente* activacionDeServicio.

La primera de ellas se llama subcapa de clases del dominio. Está integrada por los objetos que encapsulan la lógica del sistema de acuerdo a la especificación OASIS subyacente. En concreto, cada clase del dominio capturada en el paso de modelado conceptual se representará en esta subcapa como una clase que se genera a partir de la información almacenada en el repositorio del sistema (la especificación OASIS). El proceso de generación se desglosa en 2 fases:

i. la primera de ellas establece cómo las clases del dominio implementan el algoritmo de activación de servicios que propone el modelo de ejecución y que fué presentado en la Figura 3.11,

ii. la segunda tiene que ver en cómo cada sección de la especificación OASIS se representa en el código que se genera para cada clase.

Implementación del algoritmo de activación de servicios. El diseño de las clases del dominio para implementar el algoritmo de activación de servicios se basa en el patrón método de plantilla[8] (Gamma *et al.*, 1994). Una clase abstracta tendrá la responsabilidad de controlar la lógica asociada a la activación de servicios de acuerdo al algoritmo presentado anteriormente. La Figura 3.16 muestra el diseño de esta clase, la clase abstracta OASIS. El método público más importante dentro de ella es activarServicio, un método de plantilla que tiene como argumentos nombreServicio, identificadorDeObjeto y parametros y devuelve un valor booleano que indica si el servicio ha sido activado satisfactoriamente.

```
public abstract class Oasis {
// template method
public abstract boolean activarServicio(String nombreServicio,
String identificadorDeObjeto, String parametros)
{
    mediador.materializar(identificadorDeObjeto);
    comprobarPrecondiciones(nombreServicio);
    comprobarTransicionEstado(nombreServicio);
    valuaciones(parametros);
    comprobarRestricciones();
    comprobarDisparos();
    mediador.dematerializar(identificadorDeObjeto);
    return true;
}
// primitive operations
public abstract void comprobarPrecondiciones(String servicio);
public abstract void comprobarTransicionEstado(String servicio);
public abstract void comprobarRestricciones();
public abstract void comprobarDisparos();
}
```

Figura 3.16. La clase abstracta OASIS

La característica principal del método activarServicio, es que define tanto las partes variantes como las invariantes del algoritmo. Concretamente, los servicios que proporciona la clase mediador para recuperar objetos (materializar) y para almacenarlos (dematerializar) en el sistema de persistencia, son invariantes, es decir el código de éstos métodos es siempre el mismo. Sin embargo

[8] Template method

la definición de los otros servicios involucrados en el algoritmo (comprobarPrecondiciones, etc, ...) pueden variar dependiendo del servicio que está siendo activado. La definición de estos últimos servicios se delega como operaciones primitivas cuya implementación la proporcionan las clases del dominio[9] tal y como se ilustra en la Figura 3.17. Para ello obviamente éstas clases han de derivar de la clase abstracta OASIS.

Por ejemplo, comprobarPrecondiciones, es una operación primitiva del método activarServicio, por consiguiente cada clase del dominio define cómo implementar el cuerpo correspondiente a esa operación[10].

```
public class ClaseDelDominio extends Oasis{
   public void comprobarPrecondiciones(String servicio);
   public void comprobarTransicionEstado(String servicio);
   public void valuaciones(Hashtable parametros);
   public void comprobarRestricciones();
   public void comprobarDisparos();
}
```

Figura 3.17. Patrón de diseño para las clases del dominio

Una vez presentado el patrón de diseño que siguen las clases del dominio para implementar el algoritmo de activación de servicios, se describe en detalle el diseño de estas clases para representar cada sección de una especificación OASIS en la implementación, es decir, cómo se implementan todas y cada una de las operaciones primitivas presentadas en el algoritmo anterior.

Representando la especificación en la implementación. A continuación se introducen estas representaciones tomando como ejemplo el sistema de información biblioteca presentado en la Figura 3.10. El lenguaje de programación utilizado es Java. Para proporcionar un marco claro y conciso sobre el que ilustrar las

[9] Por clases del dominio se quiere expresar las clases concretas que han sido detectadas en el modelo de objetos y que son relevantes para el sistema de información que está siendo modelado

[10] Obviamente, se declara una operación primitiva para cada comprobación en el modelo de ejecución

representaciones, se utilizan 3 vistas diferentes del sistema; las vistas OO-Method y OASIS son vistas en el espacio del problema, la vista lenguaje de programación OO, es la vista en el espacio de la solución.

El objetivo de esta presentación es mostrar de forma estructurada como la representación software de conceptos precisos del espacio del problema es lo que permite automatizar el proceso de generación del código resultante.

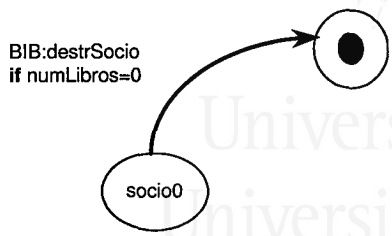
Representación de precondiciones. La Figura 3.18, muestra las tres vistas diferentes para la precondición asociada a la especificación de la clase socio. En la vista OO-Method, una precondición es una fórmula bien formada (fbf) que etiqueta una transición en el DTE de acuerdo a la sintaxis presentada en la Figura 3.4. Desde el punto de vista del lenguaje de especificación formal OASIS, una precondición es una fbf en el marco del subconjunto de la lógica dinámica usada que se representa textualmente. Finalmente, desde el punto de vista del lenguaje de programación, una precondición no es más que una sentencia condicional que debe ser comprobada. Por consiguiente, las precondiciones se representan en un lenguaje de programación OO utilizando comprobaciones condicionales que comprueban si la condición se cumple y en caso negativo se lanza una excepción.

Representación de transiciones de estado. Existen 3 formas diferentes de implementar la verificación de una transición de estado; usar lógica condicional, utilizar el patrón de estado[11] (Gamma *et al.*, 1994), o implementar un intérprete de máquina de estados que realice la verificación sobre un conjunto de reglas de transición.

Trabajos como los de (Gamma *et al.*, 1994) ponen de manifiesto que la aplicabilidad del patrón de estado no es aconsejable cuando se tienen muchos estados en el sistema (en un modelo conceptual OO-Method se tienen varios estados por cada clase). El modelo de ejecución de OO-Method opta en este caso por implementar una máquina de estados simple donde las comprobaciones de transi-

[11] State pattern

OO-METHOD



OASIS

$$bib : destrSocio\ if\ numLibros = 0; \tag{3.10}$$

LENGUAJE PROGRAMACION OO

```
protected void comprobarPrecondiciones(String servicio)
throws error {
  if (serviceName.equals("destrSocio"))
    if (!(numLibros == 0))
    throw new error ("Violación de Precondición.
    El socio tiene libros prestados");
}
```

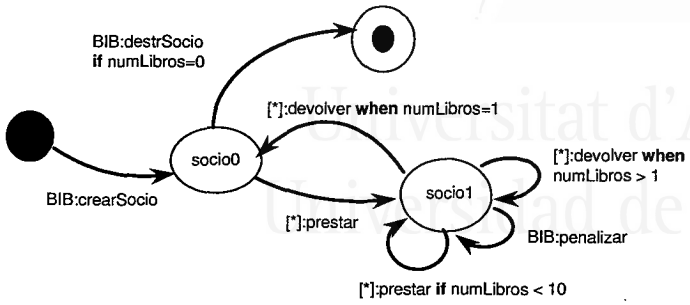
Figura 3.18. Representación de precondiciones para la clase socio

ciones válidas se realizan mediante constructores condicionales, de esta forma se elimina la complejidad de tener que implementar un intérprete real del estilo al utilizado por ejemplo en el modelo TEJA (Camara *et al.*, 1999) en la aproximación OBLOG. Este tipo de intérpretes son mucho más adecuados en entornos de validación de prototipos ya que pese a simplificar considerablemente el proceso de comprobación (se delega en la máquina de estado la responsabilidad de resolver la validez de la transición), finalmente han de ser incluidos en el proceso de generación.

La figura 3.19 muestra como se representa un DTE en el espacio de la solución utilizando este planteamiento.

Concretamente, para cada estado especificado en un DTE (en el ejemplo de la clase socio, estos estados son el de no existencia,

OO-METHOD



OASIS

$$
\begin{aligned}
&socio = bib : crearSocio \ \bullet socio0;\\
&socio0 = bib : destrSocio + prestar \bullet socio1;\\
&socio1 = if\ numLibros = 1\ devolver \bullet socio0\\
&+ (if\ numLibros > 1\ devolver\\
&+ if\ numLibros < 10\ prestar) \bullet socio1;
\end{aligned}
\tag{3.11}
$$

LENGUAJE PROGRAMACION OO

```
protected void comprobarTransicionEstado(string servicio)
throws error {
  if (state.equals("noexistencia"))
    { if (service.equals("crearSocio"))
        {state="socio0";
         return;}
    }
  else if (state.equals("socio0"))
    { if (service.equals("prestar"))
        {state="socio1";
        return;}
      else if (service.equals("destrSocio"))
        { ...}
  else if (state.equals("socio1"))
    { ...}
  throw new error ("Violación de Estado. Transición
  inválida")
}
```

Figura 3.19. Representación de las transiciones de estado de la clase socio

socio0, socio1 y destrucción), se declara una estructura condicional. Esta estructura asegura que existe una transición válida para el servicio que está siendo activado. Si el proceso tiene éxito, el correspondiente cambio de estado se llevará a cabo. En otro caso, el sistema lanzará una excepción. Por ejemplo, si suponemos que el estado actual del objeto socio es socio0 y el servicio que está siendo activado es el servicio prestar, el método comprobarTransicionEstado determinará que es una transacción válida y fijará el estado del objeto socio en socio1.

Representación de las Evaluaciones. Las evaluaciones capturadas en el modelo funcional (FM) han de ser representadas en el código de las clases. Delegaremos en un método de la clase (ver método valuaciones de la figura 3.20) la responsabilidad de comprobar qué servicio está siendo activado.

Para cada servicio perteneciente a la signatura de la clase, se genera un método que implementa el efecto especificado estableciendo modificaciones sobre los valores de los atributos involucrados en la ejecución de ese servicio según la categorización establecida en el modelo funcional.

La ejecución del servicio correspondiente (prestar o devolver), se llevará a cabo de acuerdo con el efecto especificado en el método asociado (ver Figura 3.21). Si suponemos por ejemplo que el servicio activado es prestar entonces el atributo numLibros de la clase socio verá incrementado en 1 su valor.

De esta forma el MF se embebe directamente en estos métodos que mediante asignaciones de nuevos valores sobre los atributos de los objetos establecerán los cambios de estado. Como se comentó anteriormente, estas asignaciones se realizan de acuerdo al efecto especificado en el propio modelo.

Representación de restricciones de integridad. Estas expresan ciertas propiedades que se deben de cumplir a lo largo de la traza de estados de un objeto. Como se comentó anteriormente, estas propiedades se describen en OASIS utilizando cierta lógica temporal.

OO-METHOD

```
CLASE: Socio  ATRIBUTO: numLibros

-----------------------------------------------------------
Servicio     Condición de evaluación     Efecto del servicio
-----------------------------------------------------------
prestar                                  = numLibros + 1
devolver                                 = numLibros - 1
-----------------------------------------------------------
```

OASIS

$$[prestar]\ numLibros = numLibros + 1; \tag{3.12}$$

$$[devolver]\ numLibros = numLibros - 1; \tag{3.13}$$

LENGUAJE DE PROGRAMACION OO

```
protected void valuaciones(String servicio,
Hashtable parametros) {
    if (serviceName.equals("prestar"))
        prestar(parametros);
    if (serviceName.equals("devolver")
        devolver(parametros);
}
```

Figura 3.20. Representación de las evaluaciones para la clase socio

```
protected void prestar(Hashtable parametros) {
    numLibros = numLibros + 1;
    }
protected void devolver(Hashtable parametros) {
    numLibros = numLibros - 1;
    }
```

Figura 3.21. Servicios de la clase socio

En este contexto, podemos considerar como un caso particular las restricciones de integridad estáticas ya que establecen propiedades que se deben de cumplir en todos los estados pertenecientes al ciclo de vida de un objeto. De esta forma y a efectos de implementación cada clase del dominio implementa un método que se responsabiliza de comprobar si se cumplen las propiedades asociadas a la restricción (ver Figura 3.22). En el ejemplo puede verse como el método comprobarRestricciones comprueba mediante el uso de una expresión condicional si el valor del atributo numLibros es menor que 10.

OO-METHOD

```
CLASE: Socio
-------------------------------------------------
Tipo Restricción            Fórmula
-------------------------------------------------
static                      numLibros < 10
-------------------------------------------------
```

OASIS

static numLibros < 10; (3.14)

LENGUAJE DE PROGRAMACION OO

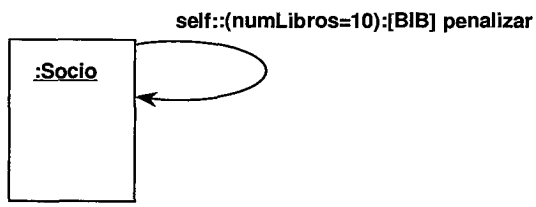
```
protected void constraints() throws error
{
if (!(numLibros < 10))
throw new error ("Violación de Restricción.
Límite de préstamos excedido";
}
```

Figura 3.22. Representación de restricciones de integridad para la clase socio

Las restricciones de integridad dinámicas son más complejas de implementar puesto que deben evaluarse a lo largo de toda la vida de los objetos. A efectos de implementación esta representación compleja se basa en el trabajo presentado en (Lipeck & Saake, 1987) que retomaremos en el capítulo 5.

Representación de disparos. El efecto de la acción de un disparo ha sido tradicionalmente estudiado en varias líneas de investigación vinculadas a la Ingeniería del Software y especialmente en el contexto de las bases de datos activas. Una propuesta común para modelar el conocimiento de los sistemas activos es el mecanismo de reglas evento-condición-acción (ECA) (Paton & Díaz, 1999). Las reglas se componen de un evento que las dispara, una condición que describe una situación concreta y una acción a realizar si la condición se satisface. En este contexto se han presentado varias aproximaciones descritas en (Elizondo, 1998) para incluir el mecanismo de reglas en los sistemas de bases de datos tales como extensiones sobre el sistema de gestión de base de datos (SGBD), aplicaciones en el contexto de la base de datos (BD) o aplicaciones externas a la BD. La aplicación de cada propuesta depende del tipo de funcionalidad que se desea soportar.

OO-METHOD



OASIS

$$Self :: penalizar\ if\ numLibros = 10; \tag{3.15}$$

LENGUAJE DE PROGRAMACION OO

```
protected void comprobarDisparos() throws error
{  if (numLibros = 10)
try { activarServicio("penalizar");
} catch (EX_triggers e) {throws e};
}
```

Figura 3.23. Representación de disparos para la clase socio

Teniendo en cuenta el mecanismo de las reglas ECA, se define un disparo OASIS como una regla activa donde el evento es el servicio que está siendo activado, la condición es la fbf que debe ser evaluada, y la acción es la acción OASIS que debe ser activada si la condición se satisface. Además se considera que el efecto de un disparo actúa en el contexto de la sociedad de objetos que está siendo especificada.

En base a estas limitaciones, se propone una estrategia simple para implementar los disparos. En nuestro caso, esta estrategia tiene sentido que sea simple puesto que el principal objetivo planteado en esta tesis es la generación automática de código. Los disparos se representan (ver Figura 3.23) como un método (comprobarDisparos) que comprueba si la condición asociada al disparo (numLibros = 10) se cumple, en ese caso el servicio correspondiente (penalizar) se activa.

De esta forma se genera el código para las clases del dominio partiendo de la especificación formal OASIS y siguiendo una estrategia concreta. Es importante resaltar el hecho de que esta estrategia para generar código que se ha presentado es independiente del lenguaje de programación destino. Ello es posible porque la implementación final se obtiene de forma automática programando las correspondientes representaciones entre los constructores de modelado conceptual y su representación en un entorno de desarrollo de software determinado (en este caso Java).

Una vez obtenida la implementación OO completa del sistema, a continuación se presenta cómo el ME proporciona los servicios de persistencia que necesitan las clases del dominio. La subcapa servicios es la responsable de esta tarea.

Esta subcapa está integrada por un conjunto de objetos (mediadores), que no pertenecen al dominio del problema pero que proporcionan los servicios de persistencia requeridos por las clases del dominio. Para los propósitos de este trabajo se han diseñado dos mediadores. Ambos siguen el diseño presentado en la Figura 3.24 (orientado a proporcionar la persistencia de los objetos sobre bases de datos relacionales). El primero de ellos denominado me-

diador de base de datos, es el encargado de asegurar que las clases del dominio tengan facilidades de persistencia. Para ello el mediador proporciona servicios de materialización y dematerialización de objetos. El segundo de ellos se denomina mediador del repositorio, y permite que los objetos que integran la capa de interface de usuario se puedan personalizar adecuadamente para el sistema de información que esté siendo generado. Para ello el mediador del repositorio proporciona un conjunto de servicios que permiten realizar consultas sobre el repositorio. Como veremos en el capítulo 4, mediante la invocación de estos servicios se puede personalizar en tiempo de ejecución los componentes que conforman el interface de usuario a partir de la información que proporciona la especificación formal almacenada en el repositorio del sistema.

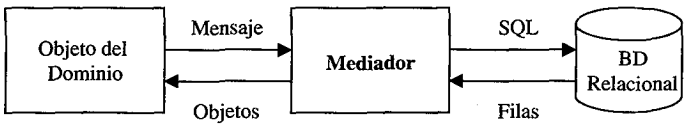


Figura 3.24. *Diseño de un mediador*

CAPA DE PERSISTENCIA. En este nivel residen los datos tanto los de la aplicación como los del repositorio. Los primeros representan el modelo de datos del sistema que está siendo ejecutado. Los segundos representan la metainformación de la especificación y realmente constituyen el diccionario de datos del sistema. Es importante resaltar el hecho de que ambas bases de datos son generadas de forma automática a partir del paso de modelado conceptual. Estrategias de cómo realizar esta tarea de forma automática sobre entornos de almacenamiento relacionales es un proceso bien conocido de traducción como los presentados en (Pastor *et al.*, 1997b; Rumbaugh *et al.*, 1991). Extensiones para bases de datos objeto-relacionales también pueden desarrollarse siguiendo una estrategia similar.

3.3.2 Paso IV. Obtener la Especificación de los Componentes

Una vez obtenida la implementación completa de las clases de dominio de acuerdo a la estrategia planteada por el modelo de ejecución, a continuación se describe cómo se completa el proceso de generación de componentes.

Sin embargo antes de describir este proceso es necesario introducir el concepto de componente en el contexto de esta tesis.

Realmente existe una considerable confusión acerca del significado exacto de qué es un objeto y qué es un componente lo que provoca que a menudo se utilicen ambos términos de forma indistinta. Quizás una de las diferencias más notables entre ambos conceptos es que un componente sólo puede existir en un entorno previamente definido según un modelo de componentes. Un modelo de componentes (Szyperski, 1998), es un conjunto de servicios definidos que son soportados por el modelo, más un conjunto de reglas que deben de ser cumplidas por el componente para aprovechar esos servicios. Entre éstas reglas existen dos que son comunes a todo modelo:

- un componente puede ser usado por otro (cliente).
- el cliente no necesita ser conocido por el componente.

La mayoría de los modelos de componentes disponibles en la actualidad proporcionan descripciones sintácticas para definir el conjunto relevante de servicios del modelo, así como la especificación de los protocolos adecuados que deben de usarse para interactuar sobre el mismo. Aunque la mayoría del trabajo de investigación desarrollado en este terreno se ha centrado en el estudio de la interoperabilidad sintáctica de los componentes, recientemente existe un interés creciente en estudiar aspectos vinculados a la interoperatibidad semántica. Esta última corriente ha provocado la aparición de nuevas propuestas de modelos de componentes (Vallecillo *et al.*, 1999; Vallecillo, 1999), que siendo compatibles

con los ya existentes traten aspectos semánticos vinculados a la interoperabilidad del software.

Algunos de estos modelos se pueden aplicar para diseñar procesamiento basado en el cliente[12], otros para diseñar procesamiento basado en el servidor[13] y otros indistintamente.

En cualquier caso, un componente debe proporcionar una o más especificaciones de interface (dependiendo del modelo), que describen el tipo de servicios que ese componente ofrece al entorno que lo usa.

En el contexto de este trabajo vamos a establecer una correspondencia directa entre componentes y clases del dominio. En este sentido, existirán en el espacio de la solución tantos componentes como clases hayan sido especificadas en la fase de modelado conceptual y posteriormente generadas por el modelo de ejecución. La información necesaria para construir los interfaces de los componentes, se obtiene a partir de la información almacenada en el repositorio del sistema (la especificación OASIS).

De esta forma podemos ver el modelo de ejecución, como un entorno integrado de generación de componentes software. La funcionalidad de los mismos queda determinada por la especificación formal OASIS que utiliza el propio modelo para generar tanto la implementación como la especificación.

Teniendo en cuenta estas consideraciones, vamos a describir el proceso de generación de interfaces. Presentaremos la estrategia de generación y la aplicaremos usando el ejemplo del sistema de información de la biblioteca, sobre tres de los modelos de componentes más representativos; CORBA (OMG, 1997) propuesto por el Object Management Group, JavaBeans/RMI (Javasoft, 1996; Downing, 1998) propuesto por Sun MicroSystems y COM (Microsoft, 1996) propuesto por Microsoft.

[12] cliente-side processing
[13] server-side processing

En los tres modelos, las interfaces públicas de un componente se definen mediante un lenguaje OO textual conocido como el IDL[14]. Dicho lenguaje es independiente del usado en la implementación del componente (en nuestro caso el código generado para las clases de dominio), y describe a nivel sintáctico los métodos públicos que implementa (junto con el tipo de los correspondientes parámetros de entrada y salida), y finalmente, las excepciones que pueden generarse durante la invocación de esos servicios. A partir de esta descripción IDL, los compiladores de IDL se encargan de generar los módulos correspondientes que permitirán invocar los servicios de los componentes creados sobre un lenguaje de programación destino.

La estrategia de generación (Gómez *et al.*, 1999; Gómez & Pastor, 1999b; Devesa *et al.*, 1999), se basa en la aplicación de los siguientes pasos. El punto de partida como se comentó anteriormente es el repositorio del sistema, esto es, la especificación OASIS.

i. Paso I: Partiendo de la información del repositorio, se genera un módulo para cada esquema conceptual. Ese modulo contiene la especificación completa en notación IDL de los componentes pertenecientes a ese sistema.

ii. Paso II: Cada módulo tiene tantas especificaciones de interface como clases del dominio hayan sido especificadas a nivel de modelado conceptual.

iii. Paso III: Cada especificación de interface describe a nivel sintáctico los métodos públicos que implementa el componente junto con el tipo de sus parámetros de entrada y salida. Los métodos protegidos que implementan el algoritmo de activación de servicios no son descritos en el interface y quedan encapsulados en la implementación del componente.

A continuación se muestra la aplicación de esta estrategia sobre los modelos de componentes seleccionados. Partiremos del ejemplo que se ha usado para describir el modelo de ejecución. La Figura

[14] Interface Definition Languaje

```
package biblioteca;
public class socio  extends OASIS {
    // attributos
    String    codSocio;
    String    nombre;
    int       numLibros;

    // servicios
    public void socio(String codigo, String nombre) {...};
    public boolean prestar(String l){...};
    public boolean devolver(String l){...};

    // fórmulas OASIS
    protected void comprobarPrecondiciones(String servicio){...};
    protected void comprobarTransicionEstado(String servicio){...};
    protected void valuaciones(Hashtable parametros){...};
    protected void comprobarRestricciones(){...};
    protected void comprobarDisparos(){...};
}
```

Figura 3.25. Código generado para la clase socio

3.25 muestra parcialmente el código generado para la clase socio. La clase tiene su conjunto de atributos, su conjunto de servicios y finalmente un conjunto de métodos protegidos que representan las fórmulas OASIS que deben ser evaluadas durante la activación de un servicio.

Integración con CORBA. Se genera un módulo CORBA (MODULE biblioteca), para el esquema conceptual de la biblioteca (aplicación Paso I). El modulo detalla los interfaces para cada componente (aplicación Paso II en este caso, INTERFACE iSocio). El interface describe sintácticamente el conjunto de métodos públicos (servicios) ofrecidos por el componente (estos servicios son socio, prestar, devolver). La Figura 3.26 muestra la estructura de la especificación IDL generada.

Integración con JavaBeans/RMI. Uno de los principales beneficios en el uso de este modelo de componentes es que para especificar los interfaces de las clases se utiliza el propio lenguaje. Consecuentemente no es necesario utilizar un lenguaje externo de IDL. Evidentemente, tenemos la restricción de trabajar con componentes desarrollados con el propio lenguaje.

```
MODULE biblioteca{
    INTERFACE iSocio{
        void socio (in string c, in string m);
        boolean prestar(in string libro);
        boolean devolver(in string libro);
    }
    ...
}
```

Figura 3.26. Especificación CORBA IDL para la clase socio class

La figura 3.27 muestra la definición del paquete java biblioteca (aplicación Paso I y II) que contiene las descripciones de interfaces de los componentes (en nuestro caso para socio). La sentencia import importa el paquete Java RMI. El interface iSocio es un interface Java estándar con dos características interesantes:

- extiende del interface RMI denominado Remote, esto hace que el interface declarado esté disponible para invocación remota.
- todos sus métodos lanzan RemoteException, excepción que se usa para notificar posibles fallos de red y de paso de mensajes.

```
package biblioteca;

import java.rmi.*;
public interface iSocio extends Remote {
    void socio(String c, String m)throws RemoteException;
    boolean prestar(String l)throws RemoteException;
    boolean devolver(String l) throws RemoteException;
}
```

Figura 3.27. Especificación Java/RMI para la clase socio

El interface ofrece únicamente los servicios de la clase (socio, prestar y devolver) para que puedan ser invocados de forma remota (aplicación paso III). Las fórmulas OASIS quedan completamente encapsuladas en la invocación de estos servicios.

Integración con COM. Microsoft ha desarrollado una especificación (COM[15]) para crear componentes y construir aplicaciones

[15] Component Object Model

con ellos. Independientemente del lenguaje que se utilice en la implementación de un componente, COM proporciona un estándar binario de interoperatibidad que facilita el desarrollo de aplicaciones mediante el ensamblaje de éstos. Por este motivo, la estrategia usada para generar componentes COM queda totalmente encapsulada en los propios entornos de desarrollo de Microsoft. De tal forma que a partir de un código que describe la funcionalidad del componente, en nuestro caso la implementación generada por cada clase del dominio, el propio entorno genera la especificación COM correspondiente.

3.4 Comparación con Trabajos Relacionados

Aproximaciones similares que también generan código a partir de la fase de modelado conceptual han sido comentadas en el capítulo 2, entre ellas destacan por su grado de madurez las últimas versiones de los métodos Shlaer & Mellor (Shlaer & Lang, 1996; Shlaer & Mellor, 1997) y OBLOG (Camara & Andrade, 1999). Ambas utilizan una aproximación traslativa para generar una aplicación en función de un conjunto de reglas codificadas a través de un lenguaje propietario. De esta forma se consigue dar al modelador un entorno preimplementado que incluye aspectos de arquitectura. Sin embargo, puede ocurrir que ese entorno no se ajuste completamente a las necesidades del modelador ya que por ejemplo puede necesitar de ciertas restricciones de arquitectura que no han sido a priori contempladas anteriormente. En este caso si modelador quiere establecer modificaciones sobre aspectos de arquitectura es necesario que conozca el lenguaje utilizado para codificar las reglas que gestionan la arquitectura software (lenguaje de arquetipos).

En contraste, el método OO-Method a través del modelo de ejecución que se ha presentado en este capítulo, proporciona una forma nueva de ver el proceso de generación de código utilizando una aproximación basada en el concepto de detección de constructores de modelado. Esta filosofía se fundamenta en que cada

concepto de modelado detectado en el espacio del problema tiene una representación software concreta. La representación software de cada concepto de modelado en el espacio del problema que establece las características dependientes de la implementación en el espacio de la solución, es lo que permite automatizar el proceso de generación del código resultante.

Según esta filosofía, cualquier sistema software puede ser generado automáticamente a partir de la fase de modelado. Todo aquello que no se pueda generar es porque no se detectó el constructor de modelado asociado junto con su correspondiente representación software. De esta forma bastaría con incluirlo en el modelo para soportar esa nueva situación de modelado.

En consecuencia, el modelo de ejecución presentado hace de OO-Method un método abierto en el sentido de que se le van añadiendo 'nuevos constructores' conforme va madurando el método.

En este sentido, el modelo que se presenta en esta tesis creemos que es más potente que los incorporados en otras propuestas que también han tenido gran aceptación en la comunidad científica como OMTroll (Wieringa *et al.*, 1993; Jungclaus *et al.*, 1987; Jungclaus *et al.*, 1995), que permite generar un prototipo para validar especificaciones y realizar comprobaciones de modelos. Con el método OO-Method el producto que se obtiene es 'muy cercano' al producto final. Las características de los formalismos sintonizables (Clyde *et al.*, 1992) y los lenguajes de modelo equivalente (Liddle *et al.*, 1995), también aparecen de forma natural en el método OO-Method. En el primer caso, la expresividad del lenguaje OASIS está preservada completamente a través del uso de notaciones gráficas convencionales con las que está familiarizado el modelador, en el segundo caso la equivalencia semántica entre la especificación y el código generado puede inicialmente intentar demostrarse a partir de la definición de un cálculo operacional para el modelo de ejecución que se presentará en el capítulo 5 y que es respetuoso con los fundamentos formales del lenguaje OASIS.

En el método SOFL (Liu *et al.*, 1998) se apuesta por integrar el uso de métodos formales en el ciclo de desarrollo del software

mediante mecanismos que permitan enlazar técnicas OO y estructuradas con métodos formales. En nuestra opinión el intento de combinar éstas técnicas en un únic̣o método hace difícil establecer de forma clara la semántica asociada al método. Además obliga a que el propio método disponga de un conjunto de herramientas que realicen comprobaciones de consistencia entre las diferentes técnicas usadas. Otro aspecto que no soporta SOFL y que es relevante para el trabajo que aquí se presenta es que no proporciona un proceso completamente automático de transformación de la especificación al código.

Por otro lado, el método OO-Method a través de sus dos modelos básicos (conceptual y ejecución) se complementa con aproximaciones más ambiciosas de Ingeniería de Requisitos Software como TRADE (Wieringa, 1998). En el contexto de TRADE, OO-Method puede ser visto como un 'componente interno' que proporciona un conjunto preciso de técnicas con una semántica formal establecida por el modelo de objetos subyacente (OASIS) para generar componentes software. De esta forma, el punto de partida del proceso de desarrollo software puede ser abstraído a un nivel superior al de la fase de modelado conceptual.

Para cerrar este apartado, es importante considerar los trabajos que en ambientes industriales están actualmente más extendidos como métodos que, de alguna forma, son presentados como generadores de componentes software a partir de modelos conceptuales 'a la UML'. Entre esas aproximaciones, se seleccionan dos por su amplia extensión comercial: Rational Rose (Booch *et al.*, 1998; Rumbaugh *et al.*, 1998; Quatrani, 1998) y Catalysis (D'Souza & Wills, 1998).

Rational Rose proporciona un ambiente de especificación genérica de diagramas UML. Soporta asímismo en particular el uso del lenguaje OCL[16] (Warmer & Kleppe, 1998) como mecanismo de declaración de propiedades de los distintos diagramas. En cuanto a su capacidad de generación de componentes software, la herramienta dispone de un lenguaje de scripts para generar código

[16] Object Constraint Language

utilizando esquemas de traducción, con la desventaja que el modelador debe conocer ese lenguaje, no trivial. Además, no dispone de una semántica definida con precisión asociada a la expresividad de los distintos diagramas soportados, lo que hace que la susodicha generación se reduzca a la definición de plantillas de clases (aprovechando la declaración de atributos y operaciones efectuada en un Diagrama de Clases). Podemos concluir que Rational Rose proporciona un entorno de generación que puede perfectamente ser calificado de pobre, debido a que esa generación de plantillas de clases es a todas luces insuficiente desde un punto de vista operacional. El código final resultante (por ejemplo, el cuerpo de los métodos) debe ser introducido explícitamente en notación propia del entorno de desarrollo elegido.

Catalysis[17] proporciona un método de desarrollo de software basado en un proceso iterativo de refinamiento de los componentes que conforman un sistema. A nivel del espacio del problema se proporcionan distintos tipos de diagramas que permiten modelar aspectos tales como; interacciones externas del componente dentro del dominio del sistema que se está modelando (diagrama de colaboración), la descripción del comportamiento preciso de cada operación proporcionada por el interface (diagrama de interface), o la posibilidad de especificar el 'contrato' de un componente con sus clientes potenciales (diagrama de tipos). A partir de la especificación que se obtiene y mediante el uso de ciertas herramientas que se pueden catalogar como de desarrollo rápido de aplicaciones (entre ellas COOL:Spex, COOL:Gen, COOL:Jex), es posible obtener una implementación de los componentes especificados. En definitiva, la conclusión extraída es que se trata de un ambiente de generación con prestaciones similares a las señaladas para Rational, y que sólo genera código utilizable si dicho código ha sido explícitamente introducido en el ámbito del espacio del problema (en el modelo conceptual resultante) usando notación del espacio de la solución (lenguaje de programación seleccionado).

[17] Existen en el mercado algunas herramientas CASE basadas en este método, entre las más representativas se encuentra la de Sterling Software (Brown & Jaeger, 1998)

3.5 Conclusiones del Capítulo

Durante estos últimos años la construcción de aplicaciones mediante el uso de componentes software ha emergido como una forma productiva y ampliamente aceptada de desarrollo de aplicaciones. En este contexto, los métodos convencionales han de proporcionar un proceso de desarrollo de software bien definido de tal forma que estos componentes puedan ser derivados automáticamente a partir de los requisitos del sistema. El propósito de esta tesis es tratar este proceso de producción en el contexto de la aproximación de modelado conceptual presentada (OO-Method). OO-Method puede ser visto como un entorno CARE (**C**omputer-**A**ided **R**equirements **E**ngineering), donde se realiza todo el proceso de desarrollo de software. Este entorno se centra en cómo capturar adecuadamente los requisitos del sistema. El modelo conceptual resultante especifica qué es el sistema (espacio del problema). Un modelo de ejecución abstracto proporcionado por el método guía la representación de los requisitos en un entorno de desarrollo de software destino (espacio de la solución).

El modelo de ejecución abstracto está basado en los constructores de modelado conceptual proporcionados por el método, de tal forma que se proporciona una representación software bien definida de cada uno de esos constructores sobre el espacio de la solución. Un modelo de ejecución concreto basado en una arquitectura de componentes ha sido presentado para abordar las particularidades de los sistemas abiertos y distribuidos basados en componentes. La implementación de estas representaciones desde conceptos en el espacio del problema a artefactos software en el espacio de la solución abre la puerta a la generación automática de componentes software. Estos componentes constituyen juntos un producto software que es semánticamente equivalente a la especificación capturada en el paso de modelado conceptual como se verá en el capítulo 5.

Resumiendo, la contribuciones más relevantes de este capítulo son las siguientes:

- una presentación detallada de la aproximación OO-Method como una propuesta exitosa para cubrir completamente el proceso de producción de software desde un punto de vista OO que aproveche las ventajas que proporcionan los métodos formales y los convencionales.
- un proceso para soportar la producción de componentes software dentro de la aproximación metodológica.
- una estrategia específica para generar código basada en una separación clara de la especificación del componente de la implementación del mismo, con idea de habilitar un diseño de la aplicación independiente de la tecnología usada.
- un marco de trabajo bien definido para ejecutar la especificación en el espacio de la solución.

Estas ideas han sido aplicadas en el contexto de una herramienta para el desarrollo y comprobación de aplicaciones software basadas en componentes que ha sido denominada JEM[18]. JEM es una implementación basada en Java para el modelo de ejecución de OO-Method. El propósito básico de JEM es animar modelos conceptuales elaborados con el método OO-Method, sobre entornos distribuidos internet/intranet. JEM satisface los requisitos respecto al modelo y la arquitectura propuesta en este capítulo para el modelo de ejecución.

En el capítulo siguiente se presenta con detalle el sistema JEM.

[18] **J**ava-based implementation for the OO-Method **E**xecution **M**odel

4. El Sistema JEM

"... diseñar un buen mecanismo de comunicación que permita entender el entorno y las necesidades del usuario es una de las piezas claves para desarrollar un buen software..."

(FOWLER & SCOTT, 1997)

Este capítulo describe el sistema JEM, una herramienta basada en web para animar modelos conceptuales sobre entornos internet/intranet. Los modelos conceptuales son elaborados según el método OO-Method, siguiendo las fases presentadas en el capítulo anterior; modelo conceptual, especificación OASIS, implementación OO, componentes software. Estos artefactos software son publicados en el sistema JEM produciendo un prototipo listo para ser comprobado en la Web. El prototipo permite seleccionar e invocar interacciones entre objetos para validar los requisitos del sistema capturados en el paso de modelado conceptual.

4.1 Introducción

Tal y como se comentó en la introducción y conviene recordar aquí, quizás el elemento más crítico en el desarrollo de un sistema software sea el entender y documentar de forma adecuada los requisitos del sistema que se pretende desarrollar. En este sentido, la construcción de especificaciones de requisitos que sean precisas, formales y fácilmente entendibles es uno de los componentes más importantes dentro del proceso de producción de software. Lamentablemente esta tarea es bastante difícil de realizar en la práctica, puesto que un conjunto de ideas informales deben de ser convertidas a un modelo formal a través de un mecanismo

de comunicación caracterizado por ser incompleto e impreciso. El asegurar de forma fiable el comportamiento deseado de un sistema a partir de su especificación de requisitos depende principalmente de dos factores fundamentales (Berzins *et al.*, 1993):

- validación: que la especificación original se corresponda con las necesidades reales del cliente.
- verificación: que la implementación derivada sea "consistente"con la especificación.

En este contexto los prototipos ejecutables de software (Balzer *et al.*, 1984), ayudan a guiar la reformulación de la especificación en caso de interpretaciones erróneas y también son útiles para confirmar que una especificación propuesta es representada correctamente en el dominio de la solución.

Por todos estos motivos se ha considerado necesario el dotar al método OO-Method de un entorno que permita evaluar la funcionalidad de los 'artefactos software' que produce. Para ello, OO-Method complementa su entorno de producción de software (CASE OO-Method), con un entorno de ejecución (JEM), que permite animar[1] modelos conceptuales.

El proceso de producción y animación de software se ilustra en la Figura 4.1. Inicialmente el diseñador OO-Method utiliza la herramienta CASE (Pastor *et al.*, 1997c) para desarrollar una instancia de un modelo conceptual utilizando los constructores de modelado proporcionados por el método. Una vez completada la especificación del modelo y generada la representación correspondiente, los artefactos software que se obtienen son: un modelo de datos que representa la parte estática del sistema (se corresponderá con una especificación de una base de datos relacional para gestionar la persistencia de los objetos), y unos componentes que representarán la parte dinámica del sistema (encapsularán el comportamiento del mismo). Sin embargo, ¿cómo podemos asegurar que la aplicación que se obtiene ensamblando estos componentes

[1] Se entiende por 'animar' la ejecución de la aplicación software generada automáticamente para un modelo conceptual

cumple con la especificación de requisitos capturada? Es necesario por tanto un proceso que permita contrastar la funcionalidad del sistema resultante respecto a los requisitos del mismo. En concreto, el problema se reduce a verificar que el proceso de generación automática de código produce unos componentes que cumplen la especificación de requisitos inicial del sistema.

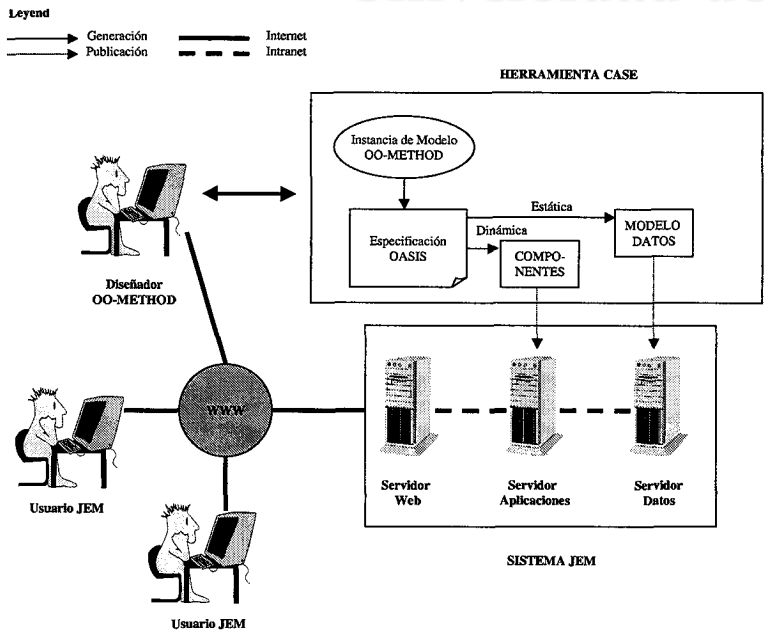


Figura 4.1. *El entorno de producción y animación de software.*

El cumplir con la especificación de requisitos inicial del sistema significa que tanto la 'estructura' como el 'comportamiento' de los componentes software generados está determinado por la información que se especificó a nivel de modelado conceptual.

En este caso diremos que la aplicación obtenida es 'semánticamente equivalente' a la especificación o dicho de otro modo, la semántica de la especificación del sistema se preserva en la implementación del mismo.

La herramienta que se ha construido para soportar este proceso es JEM (Gómez & Pastor, 1999a). JEM (ver Figura 4.1), hace

uso de las ventajas proporcionadas por la aparición de nuevas tecnologías como internet/intranet para proporcionar un entorno adecuado que permita evaluar la funcionalidad o comportamiento del sistema generado. Una de las consecuencias más directas que ha provocado este trabajo ha sido validar que la estrategia de generación de código propuesta por el modelo de ejecución permite embeber en la implementación toda la funcionalidad (sin pérdidas semánticas), especificada en el modelo conceptual. A continuación se describe la arquitectura del sistema JEM.

4.2 Arquitectura

Conceptualmente JEM es un servidor de aplicaciones embebido en una estructura de red centralizada por un servidor Web[2]. JEM posibilita que el diseñador OO-Method pueda 'descargar' a través del servidor Web el modelo de datos y los componentes que han sido generados (ver Figura 4.1), y de forma transparente los distribuye de acuerdo a una arquitectura software de tres capas idéntica a la proporcionada por el modelo de ejecución. A partir de ese momento, JEM da de alta el nuevo modelo conceptual en el sistema habilitándolo para que cualquier cliente que disponga de un navegador internet pueda seleccionarlo y ejecutar su aplicación software correspondiente.

La Figura 4.2 ilustra la arquitectura de JEM. Los elementos básicos de esta arquitectura son un servidor web, un servidor de aplicaciones y un servidor de datos.

El servidor web actúa de interface entre el cliente y el sistema, se encuentra basado en tecnología de servlets. Concretamente los servlets (clases java que se ejecutan en el servidor), responden a las peticiones que bajo protocolo HTTP[3] realizan los clientes desde un navegador internet. Los servlets generan dinámicamente código HTML[4] que se envía al cliente con el resultado de la petición.

[2] Web-centric application server
[3] HyperText Transfer Protocol
[4] HyperText Mark-up Language

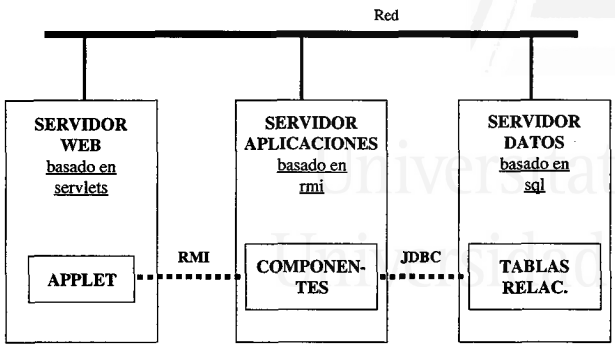


Figura 4.2. *Arquitectura de JEM.*

El objetivo de éstos servlets es proporcionar un mecanismo de comunicación entre el usuario y el sistema.

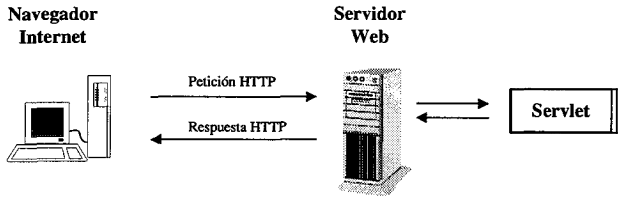


Figura 4.3. *Flujo básico de un servlet.*

Utilizando este mecanismo de comunicación simple que se ilustra en la Figura 4.3, el objetivo último es configurar los parámetros de un miniprograma (applet), que se envía al cliente y que sirve de punto de entrada para la ejecución de un modelo conceptual.

El servidor de aplicaciones es el responsable de almacenar los componentes software generados por cada modelo. Estos componentes encapsulan la lógica de un sistema de acuerdo a la especificación OASIS que se capturó en el proceso de modelado conceptual correspondiente. Los servicios de estos componentes puede ser invocados remotamente desde el cliente utilizando el protocolo correspondiente.

Por último, los aspectos vinculados a la persistencia de los datos que manejan estos componentes son controlados por un servidor de datos (ver Figura 4.2). En su estado actual JEM utiliza un

servidor de datos relacional que gestiona la persistencia de los objetos del sistema sobre tablas relacionales.

4.3 Ejecución de un Modelo Conceptual

Una sesión con JEM comienza a partir del momento que un cliente se conecta a una dirección internet[5].

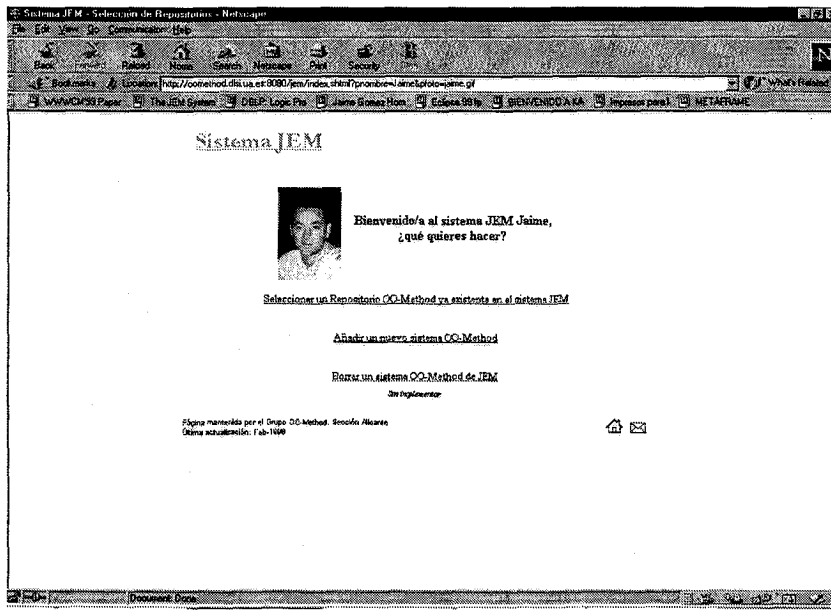


Figura 4.4. *Opciones del sistema JEM.*

El sistema pide inicialmente un control de acceso para validar que el usuario conectado está autorizado. Si el usuario es autorizado (ver Figura 4.4), puede activar una de las tres opciones disponibles: animar un modelo conceptual, registrar un modelo conceptual o eliminar un modelo conceptual.

Para registrar un modelo conceptual es necesario 'descargar' los artefactos software producidos por el entorno de producción (he-

[5] http://oomethod.dlsi.ua.es:8080/jem

rramienta CASE OO-Method). La Figura 4.5, muestra esta situación; se registra el modelo conceptual de la biblioteca 'descargando' en el sistema el modelo de datos (biblioteca.mdb) y la lógica (biblioteca.jar). Adicionalmente también se 'descarga' el repositorio del sistema (repBiblioteca.mdb), que contiene la especificación del modelo conceptual y que va a permitir generar en tiempo de ejecución el interface de usuario de la aplicación.

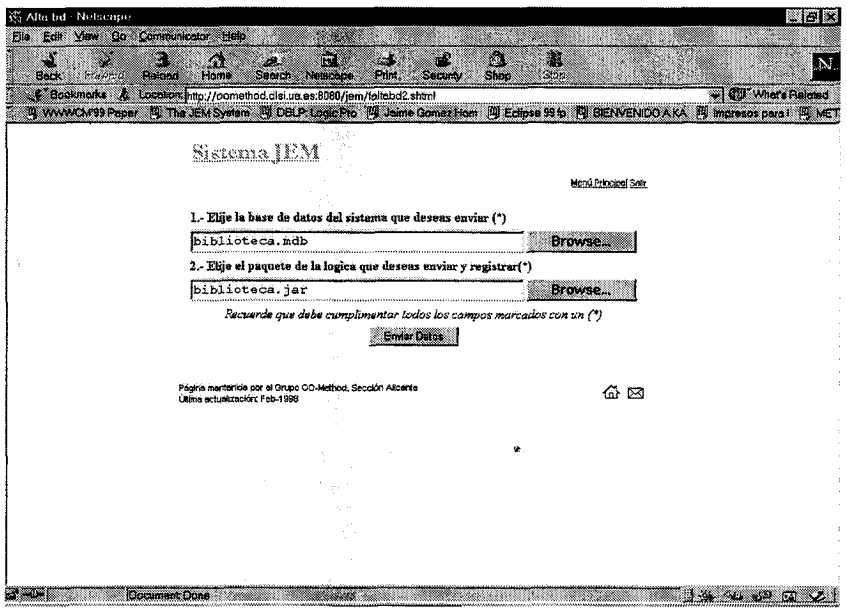


Figura 4.5. *Descargando los artefactos software.*

Una vez que se ha registrado un modelo conceptual, el sistema JEM actualiza su lista de modelos conceptuales disponibles. Para poder animar un modelo primeramente hay que seleccionarlo desde la página de selección de modelos conceptuales (ver Figura 4.6), y posteriormente configurar ciertos parámetros relacionados con la ejecución de la aplicación.

La página que permite la introducción de esta información se ilustra en la Figura 4.7. Concretamente en estos parámetros se proporciona información que tiene que ver con el espacio del problema como por ejemplo el nombre del repositorio que contiene

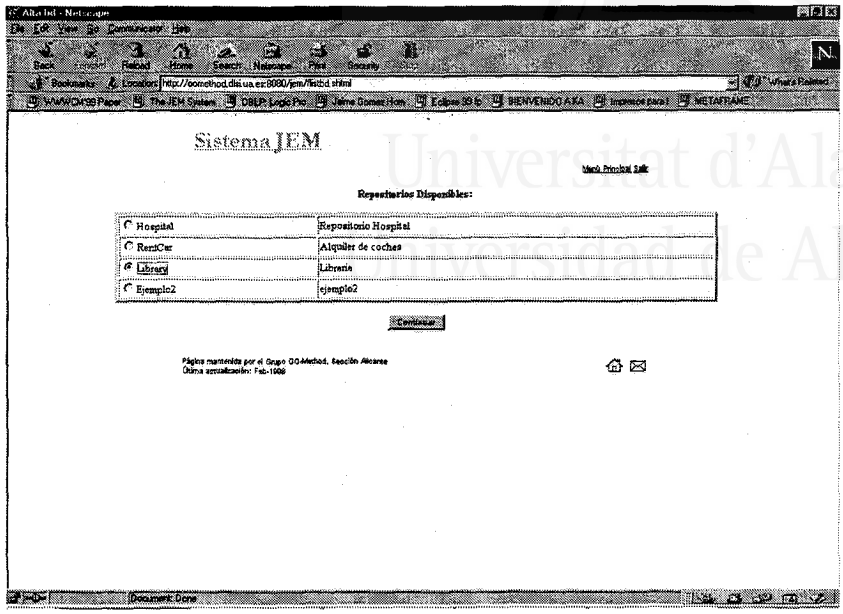


Figura 4.6. *Selección de un modelo conceptual.*

la especificación OASIS (repBiblioteca) y el nombre del modelo conceptual a prototipar (biblioteca), e información que tiene que ver con el espacio de la solución como por ejemplo, la dirección internet del servidor de aplicaciones (oomethod.dlsi.ua.es), la versión de la máquina virtual Java que puede ejecutar el cliente, o la resolución en puntos de la pantalla. La información que el usuario de JEM introduce en éstos parámetros permiten configurar un applet que proporcionará el punto de entrada al entorno de ejecución del modelo conceptual.

La Figura 4.8 muestra la estructura del applet. El applet se encapsula dentro de una página HTML generada por el sistema JEM y se envía al cliente tras la confirmación de la introducción de los parámetros. Los parámetros del applet se inicializan con los valores introducidos por el usuario. De esta forma el applet una vez que se ha sido descargado en el cliente cuando entra en ejecución sabe con qué modelo conceptual debe trabajar (Modelo_Conceptual), qué repositorio OASIS necesita consultar para configurar el interface de usuario de la aplicación (repositorio_OASIS), cuál es el

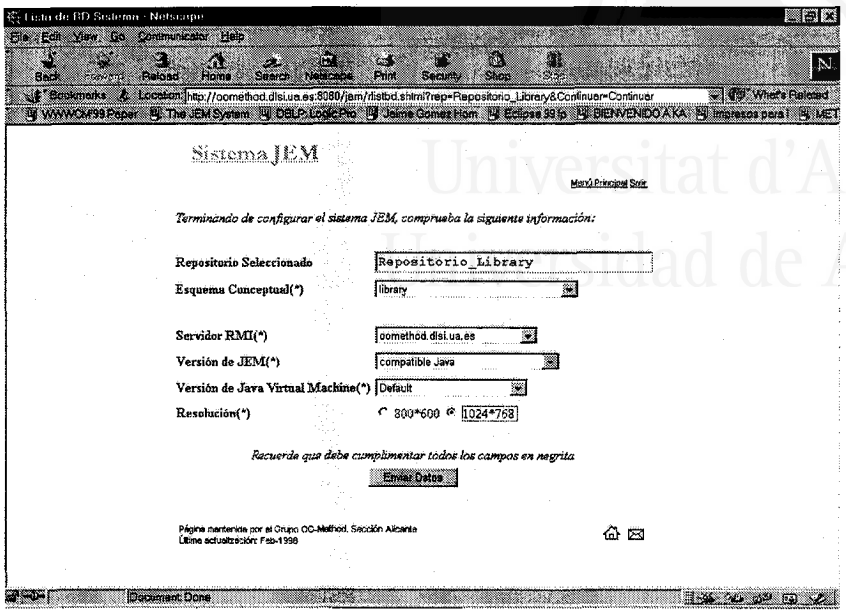


Figura 4.7. *Configuración de parámetros para la ejecución.*

nombre del servidor de aplicaciones donde residen los componentes pertenecientes a la lógica de la aplicación (servidor_RMI), y finalmente qué resolución de la pantalla debe utilizar para construir el interface de la aplicación (resolucion_X y resolucion_Y).

```
<applet code='Prototipo.class' archive='jem.jar'
        width='430' height='270'>
  <PARAM NAME='Modelo_Conceptual'     VALUE='biblioteca'>
  <PARAM NAME='repositorio_OASIS'     VALUE='repBiblioteca'>
  <PARAM NAME='servidor_RMI'          VALUE='oomethod.dlsi.ua.es'>
  <PARAM NAME='resolucion_X'          VALUE='1024'>
  <PARAM NAME='resolucion_Y'          VALUE='768'>
</applet>
```

Figura 4.8. *Applet descargado por el cliente.*

El proceso de descarga del applet consiste en la recepción en el cliente del fichero jem.jar, un fichero comprimido en formato jar[6] que contiene el conjunto mínimo de componentes necesarios para

[6] java archive

arrancar el proceso de ejecución. La clase Prototipo.class (empaquetada en este fichero), es el punto de entrada al proceso de animación.

Durante el proceso de animación, todas las actividades vinculadas a la gestión del interface de la aplicación, son manejadas por el cliente con la consecuente liberación de carga para el servidor de aplicaciones. Como se comentó anteriormente, el interface de usuario se interpreta en tiempo de ejecución en base a la información almacenada en el repositorio del sistema (en el ejemplo, la especificación OASIS almacenada en repBiblioteca).

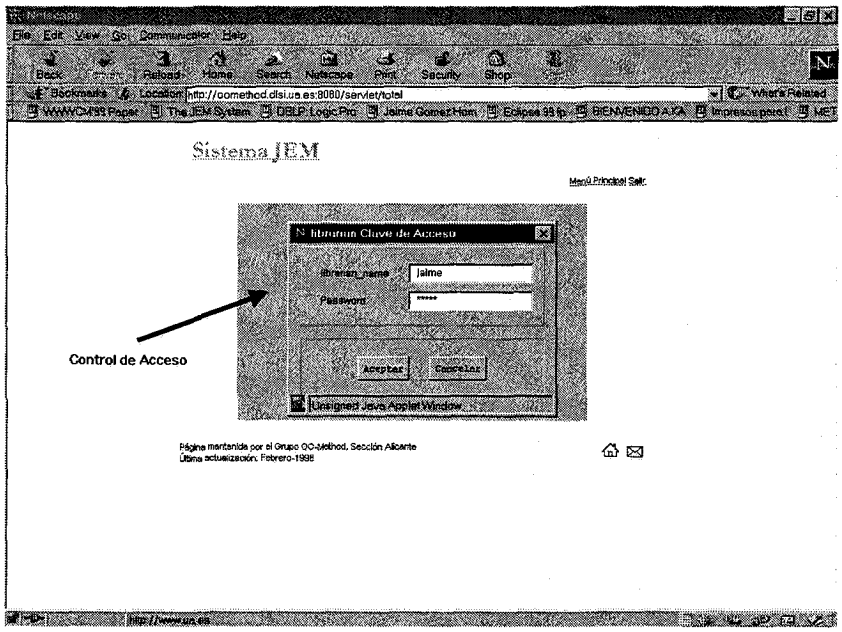


Figura 4.9. *Conexión de usuario válido.*

La primera actividad del applet es consultar este repositorio y almacenar en la memoria del cliente la información necesaria para construir el interface. Entre esta información se incluyen aspectos tales como la lista de clases, lista de atributos y servicios por clase, lista de argumentos de los servicios, etc...

Siguiendo la filosofía del modelo de ejecución presentado en el capítulo 3, el proceso de animación comienza con un control de acceso a través del cual el usuario que quiere prototipar el sistema ha de registrarse en el mismo como un usuario válido (ver Figura 4.9).

Si el usuario es validado el proceso de prototipación se realiza en base a la selección y activación de los servicios que aparecen en la vista del sistema para el objeto conectado. La vista del sistema se ilustra en la Figura 4.10 y está compuesta por la lista de clases y la lista de servicios accesibles por ese usuario.

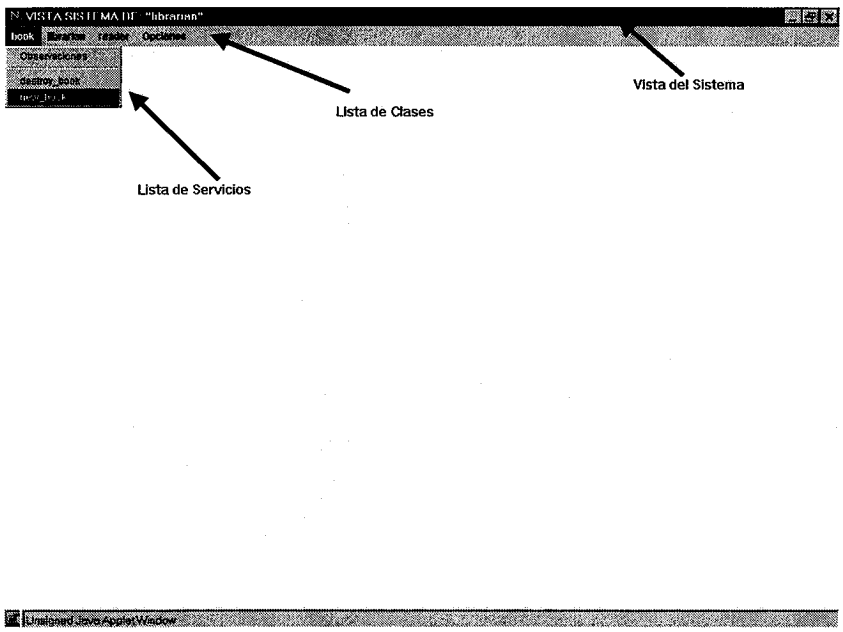


Figura 4.10. *Vista del sistema.*

Cuando un usuario selecciona algún servicio perteneciente a su vista, JEM genera automáticamente el diálogo que contiene la lista de argumentos que el usuario ha de introducir para que la activación de ese servicio se lleve a cabo (ver Figura 4.11). En el momento que el usuario valide ese diálogo (pulse el botón de

Aceptar), se producirá la invocación remota del servicio correspondiente sobre el componente involucrado.

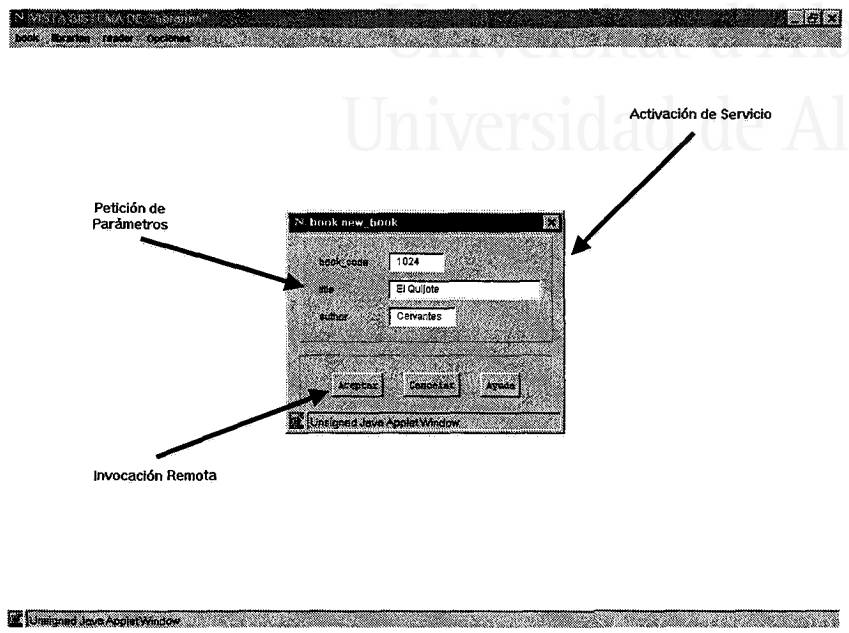


Figura 4.11. *Activación de servicios.*

De esta forma el proceso de prototipado consiste en observar de forma interactiva los resultados de la ejecución del prototipo con el objetivo de (si es necesario), refinar el modelo conceptual. El modelo se refina extendiendo, modificando o introduciendo nuevos requisitos del sistema en la fase de modelado conceptual. El proceso de refinamiento será necesario cuando se aprecien errores u omisiones en la ejecución del prototipo. Si este es el caso el diseñador ejecuta el entorno de producción de software (CASE OO-Method), y realiza las modificaciones oportunas sobre los diagramas de modelado conceptual.

Los cambios realizados automáticamente son actualizados en el repositorio OASIS. Si estos cambios afectan a la funcionalidad de los componentes, éstos deben volver a ser generados y publicados en el sistema JEM. En cualquier otro caso el sistema de-

tecta los cambios sobre el repositorio de forma automática. De esta forma la sesión de prototipado se personaliza de acuerdo a la nueva especificación y el diseñador puede continuar testeando el sistema. JEM está accesible a través de la dirección internet http://oomethod.dlsi.ua.es:8080/jem.

4.4 Conclusiones del Capítulo

Es un hecho comúnmente aceptado que los prototipos de aplicaciones software mejoran la comunicación con los usuarios puesto que proporcionan un modelo ejecutable del sistema durante el proceso de desarrollo del software. En este sentido la confirmación por parte del usuario de que el sistema software que se está diseñando se corresponde con sus necesidades reduce de forma significativa el coste del ciclo de vida puesto que los cambios realizados en las primeras fases del desarrollo son mucho más baratos que los realizados cuando el producto software ya ha sido entregado.

Tradicionalmente el propósito del prototipado del software ha sido el de proporcionar un mecanismo para estabilizar los requisitos del sistema antes de invertir en el esfuerzo de construir la implementación del mismo. En este sentido, en nuestro grupo de investigación se han realizado estudios profundos para dotar al propio lenguaje OASIS de esta funcionalidad (Letelier, 1999).

Teniendo en cuenta estas consideraciones, el método propuesto por el modelo de ejecución presentado en esta tesis, proporciona una contribución interesante de destacar. Ésta parte de la base de diferenciar dos fases principales en el método; la fase de producción de software y la fase de prototipado. El propósito de la fase de producción de software es generar una implementación eficiente de los requisitos iniciales del sistema mientras que el propósito de la fase de prototipado es estabilizar los requisitos software observando la ejecución de la implementación generada. De esta forma, el ciclo de desarrollo del software especificación/prototipado/implementación, se invierte en un ciclo especificación/implementación/prototipado controlado por el modelo de

ejecución. La necesidad de modificar los requisitos se detecta en la fase de prototipación a la que le sigue una nueva fase de producción de software y así sucesivamente hasta que se obtiene el producto final deseado.

Otra contribución importante de JEM es que proporciona en la fase de prototipado un único interface de usuario integrado por un conjunto de componentes que de forma automática y dinámica interpretan en cada ejecución asociada a un modelo conceptual, la información almacenada en el repositorio correspondiente.

Siguiendo estos fundamentos en este capítulo se ha presentado JEM una herramienta que permite llevar a cabo la fase de prototipado y que se complementa con la herramienta que se usa en la fase de generación de código (CASE OO-Method). JEM proporciona un entorno avanzado para ejecutar modelos conceptuales desarrollados con el método OO-Method.

JEM permite que el mantenimiento o evolución de una aplicación software se realice a través del mantenimiento sobre su modelo conceptual correspondiente y no sobre el código generado lo que proporciona una gran ventaja en el proceso de prototipado del software.

5. Equivalencia Semántica

"... el modelo estándar de desarrollo de software sostiene que cada paso en el desarrollo debería ser una realización válida *de la especificación. Por* válida *quiero significar que el comportamiento especificado en la implementación sea un subconjunto del que ha sido definido en la especificación ..."*

(BERZINS *et al.*, 1993)

En este capítulo se sientan las bases, con ayuda de un formalismo matemático, para mostrar que la implementación generada por el modelo de ejecución presentado en el capítulo 3 es semánticamente equivalente a la especificación formal OASIS capturada en el paso de modelado conceptual. Para ello se define un cálculo operacional sobre el modelo de ejecución que establece de forma precisa los efectos de la ejecución de un programa P en términos de dos operaciones básicas; actualizaciones y consultas. El cálculo utiliza una semántica de asignaciones para describir como se pasa de un estado de programa P_i al siguiente P_{i+1}. De esta forma, la equivalencia semántica se podrá demostrar verificando que el cálculo así definido establece una relación entre una representación inicial del sistema en el espacio del problema como una teoría de la lógica dinámica (especificación OASIS), y su correspondiente representación en el espacio de la solución como un programa imperativo (implementación OO).

5.1 Introducción

Tal y como se presentó en el capítulo 3 una especificación OASIS es un conjunto estructurado de definiciones de clase. Estas

definiciones responden a un criterio de agrupamiento basado en propiedades de estructura y comportamiento que son compartidas por un conjunto de objetos.

Una plantilla de clase OASIS se formaliza en el contexto de un subconjunto de la lógica dinámica, una variante de la lógica modal que fué propuesta por David Harel en (Harel, 1984) con el propósito de proporcionar un formalismo natural para razonar acerca de los programas. Aspectos tales como asegurar y probar que cierto programa satisface unas determinadas especificaciones son abordables en dicha lógica. Tal y como se presenta en (Canós, 1996), las fórmulas de la lógica dinámica usada son de la forma,

$$\phi \ [p] \ \phi' \tag{5.1}$$

Donde ϕ y ϕ' son fórmulas bien formadas (fbf's) de cierta lógica de primer orden, p representa un programa o parte de él y [ ] es el denominado operador de necesidad. El significado de la fórmula puede enunciarse de la manera siguiente: para todo mundo ω que es modelo de ϕ (lo representamos como $\models_\omega \phi$), tras la ejecución del programa p el sistema puede pasar a otros mundos ω' que son modelos de ϕ' ($\models_{\omega'} \phi'$).

En este contexto, una plantilla de clase OASIS es una teoría de la lógica dinámica cuya semántica declarativa (semántica de mundos posibles), se interpreta en el marco de una estructura de Kripke simple (Kripke, 1980) de la forma,

$$\mathcal{K} = (\mathcal{W}, \omega_0, \tau, \rho) \tag{5.2}$$

donde $\mathcal{W}$ es el conjunto de todos los mundos posibles[1] que un objeto puede alcanzar, ω_0 el mundo inicial y,

[1] Un mundo es una estructura de interpretación de la lógica de estado. Un estado viene dado por una fbf en la lógica de primer orden de estado. Por ejemplo: siendo $a_i \in A$ y $v_i \in |sort(a_i)|$, un estado representado en lógica ecuacional sería:

$$\bigwedge_{i=1,n} a_i = v_i \tag{5.3}$$

$$\tau : \gamma \rightarrow 2^W \qquad \rho : X \rightarrow 2^{W \times W} \tag{5.4}$$

La función τ se denomina función de satisfabilidad y asigna a cada elemento de γ, el conjunto de mundos en los cuales se satisface, donde γ representa el conjunto de fbf de la lógica de estado. La función ρ se denomina función de alcanzabilidad y asigna a cada elemento del conjunto X de acciones de la signatura del objeto, una relación binaria de alcanzabilidad entre mundos que representa la semántica declarativa del lenguaje. En este sentido, como se afirma en (Letelier *et al.*, 1998), si $\mu \in X$, y $w, w' \in \mathcal{W}$, entonces $(w, w') \in \rho(\mu)$ si y sólo si la ocurrencia de μ conduce al objeto desde el mundo w al mundo w'.

A continuación vamos a mostrar cómo con una semántica de este tipo podemos describir los efectos de la ejecución de un programa. Para ello partiremos de la definición formal de plantilla de clase OASIS definida en (Canós, 1996) y adaptada para nuestros propósitos.

Definición 5.1.1 (Plantilla de clase). *Una* plantilla de clase *OASIS es una tupla* (A, X, Φ, Π)*, donde*

- $A = A_c \cup A_v \cup A_d$ *es un conjunto de atributos.*
- $X = X_p \cup X_c$ *es un conjunto de eventos.*
- $\Phi = \Phi_v \cup \Phi_d \cup \Phi_r \cup \Phi_p \cup \Phi_t$ *es un conjunto de fórmulas de la lógica dinámica.*
- Π *es un término del álgebra de procesos.*

Tal y como se comentó en el capítulo 3 y conviene recordar aquí, el conjunto A está formado por tres subconjuntos de atributos; A_c representa el subconjunto de los atributos constantes. Estos toman valor en el momento de creación del objeto y su característica principal es que su valor no cambia durante la vida del mismo. A_v representa el subconjunto de los atributos variables, su valor viene definido por reglas de valuación que expresan cómo varían con la ocurrencia de eventos. Finalmente, A_d representa el subconjunto

de los atributos derivados, cuyo valor se obtiene a partir de otros atributos mediante ciertas reglas de derivación.

X está formado por dos subconjuntos de eventos; X_p que representa el subconjunto de eventos propios y X_c que representa el subconjunto de eventos compartidos.

El conjunto Φ está formado por los siguientes tipos de fórmulas dinámicas:

- Φ_v (evaluaciones): son fórmulas que definen cambios de estado ante la ocurrencia de eventos.
- Φ_d (derivaciones): son fórmulas que expresan cómo obtener el valor de los atributos derivados.
- Φ_r (restricciones de integridad): son fórmulas usadas para representar restricciones sobre los estados o secuencia de estados de los objetos.
- Φ_p (precondiciones): son aquellas fórmulas que expresan condiciones que determinan cuando un evento puede ser activado.
- Φ_d (disparos): son fórmulas que permiten especificar actividad interna basada en el estado del objeto.

En cualquiera de estas fórmulas dinámicas, ϕ, ϕ' son fórmulas bien formadas (fbf's), en una lógica de primer orden. Estas fbf's se refieren a un estado del sistema caracterizado por un conjunto de valores ligados a atributos de objetos.

Finalmente, Π es un término del álgebra de procesos que describe cuáles son las secuencias aceptables de eventos.

Tomando como punto de partida esta definición formal de plantilla de clase, a continuación vamos a definir un cálculo operacional para el modelo de ejecución que nos permita razonar acerca de la ejecución de los programas y de la semántica asociada a los mismos. Con la definición de este cálculo pretendemos sentar las bases que permitan, posteriormente, intentar demostrar la prueba de la equivalencia semántica.

5.2 Un cálculo operacional para el modelo de ejecución

Un programa toma como punto de partida un conjunto de datos iniciales, los computa y produce unos resultados. Cuando se está escribiendo un programa resulta frecuente el uso de variables de forma que éstas suelen adoptar diferentes valores durante la ejecución del mismo. En un instante de la ejecución de un programa estará en un estado de ejecución que viene determinado por el valor actual de sus variables. De esta forma (ver Figura 5.1), un programa puede verse como un proceso de transformación entre tales pares de estados de ejecución.

$$\underbrace{(v_1^1, v_2^1, ..., v_n^1)}_{estado_1}, \underbrace{(v_1^2, v_2^2, ..., v_n^2)}_{estado_2}, ..., \underbrace{(v_1^n, v_2^n, ..., v_n^n)}_{estado_n}$$

Figura 5.1. *Estados de ejecución de un programa*

En la figura anterior, $v_i^1, v_i^2, ..., v_i^n$ son los sucesivos valores de la variable v_i

En el contexto de la lógica dinámica clásica, cada estado de un objeto OASIS puede representarse como una fórmula en la lógica de estado, por ejemplo, a=v (lógica funcional), o atributo(a,v) (lógica clausal), donde a es un atributo del objeto y v su valor en dicho estado. Utilizando un planteamiento similar, en un contexto imperativo podemos modelar el estado de un objeto como una conjunción de pares atributo,valor donde,

$$\sigma_i \in \bigwedge_{i=1,n} a_i = v_i \qquad con\ a_i \in A\ y\ v_i \in |SORT(a_i)| \qquad (5.5)$$

Siendo σ_i el estado que expresa la fórmula ϕ en el mundo ω_i donde sólo tenemos en cuenta el conjunto de los atributos relevantes que se ven afectados por el acontecer de un evento. En este caso decimos que ω_i es modelo de σ_i y lo representamos como $\models_{\omega_i} \sigma_i$ pudiendo tomar, por ejemplo,

$$\omega_i = \{(a_i, v_i)\} \quad con\ a_i \in A\ y\ v_i \in |SORT(a_i)|, i = 1, n \quad (5.6)$$

y diremos que $\models_{\omega_i} \sigma_i$, si y sólo si,

$$(a_i, v_i) \in \omega_i\ \forall\ i = 1, n \quad (5.7)$$

(a_i, v_i) representa el conjunto de pares atributo,valor relevantes en el mundo ω_i. De esta forma la correspondencia entre estado y mundo puede entenderse en nuestro caso como una relación de satisfacción implementada como una relación de pertenencia. Según esto, un mundo modela un estado si ese estado (el conjunto de pares (a_i, v_i) implicados en el estado), pertenece a ese mundo o dicho de otra forma,

$$\models_{\omega_i} a_i = v_i \qquad \Leftrightarrow \qquad (a_i, v_i) \in \omega_i \quad (5.8)$$

Obviamente se cumple que,

$$\omega_i \supseteq \bigcup_i (a_i, v_i), i = 1, n \quad (5.9)$$

Basándonos en esta noción de mundo vamos a proporcionar un mecanismo matemático para razonar acerca de la equivalencia semántica entre una especificación OASIS y la correspondiente implementación generada por el modelo de ejecución. El cálculo operacional se define sobre los conjuntos Φ y Π de la plantilla de clase OASIS, siendo ϕ y ϕ' fbf's en lógica de predicados de primer orden y $e \in X$ (siendo X el conjunto de eventos en la signatura del objeto).

5.2.1 Evaluaciones

La evaluaciones son fórmulas de la forma $\phi\ [e]\ \phi'$ que definen cómo se producen los cambios de mundos ante la ocurrencia de eventos. La semántica de estas fórmulas viene dada por una función ρ que, a partir de un evento, devuelve una relación entre mundos posibles.

$$\rho(e) \in \mathcal{W} \rightarrow \mathcal{W} \tag{5.10}$$

Dicho de otra forma, siendo un mundo posible para un objeto modelo de uno o varios estados, la función ρ determina qué transiciones entre mundos son válidas al ejecutar el evento e. La fórmula ϕ es evaluada en ω, y ϕ' es evaluada en ω', siendo ω' el mundo modelo del estado del objeto después de la ejecución del evento considerado.

Definición 5.2.1. *Sea ψ una fórmula de evaluación tal que $\psi = \phi\ [e]\ \phi'$, sea σ_i el estado que expresa la fórmula ϕ en el mundo ω_i tal que $\sigma_i \rightarrow \phi$ y $a_i, a'_i \in A$*

$$\phi = \bigwedge_{i=1,m} a_i = v_i \qquad y \qquad \phi' = \bigwedge_{i=1,n} a'_i = v'_i \tag{5.11}$$

Entonces el estado σ_{i+1} que se obtiene tras la ejecución del evento e se calcula mediante la siguiente fórmula:

$$\sigma_{i+1} = \sigma_i\{a'_i\ /_R\ v'_i\} \cup \phi' \qquad con\ a'_i = v'_i \in \phi' \tag{5.12}$$

donde interpretamos,

$$\phi' = \bigwedge_{i=1,n} a'_i = v'_i \qquad como \qquad a'_i := v'_i \qquad i = 1, n \tag{5.13}$$

con lo que garantizamos ϕ' ya que:

$$[a'_i := v'_i]\ a'_i = v'_i \qquad i = 1, n \tag{5.14}$$

Ejemplo 5.2.1. Supongamos una evaluación de la siguiente forma: "si el estado de una cuenta bancaria está caracterizado en un determinado momento por un saldo igual a n, si acontece el evento ingresar(m) el saldo de la cuenta se ve incrementado a m+n."

Si representamos esta evaluación como una fbf de la lógica dinámica obtenemos,

$$\underbrace{saldo = n}_{\phi}\ \underbrace{[ingresar(m)]}_{e}\ \underbrace{saldo = n + m}_{\phi'} \tag{5.15}$$

Aplicando el cálculo definido anteriormente tenemos que,

$$\sigma_i = ... \wedge saldo = n \wedge ... \tag{5.16}$$

El estado σ_i expresa (o implica) la fórmula dinámica ϕ en el mundo ω_i, y su valor se calcula mediante una sustitución por la derecha del valor asociado al atributo relevante en la fórmula, en este caso el saldo. Tras la ejecución del evento ingresar(m), el valor en el estado siguiente (σ_{i+1}) pasa a ser,

$$\sigma_{i+1} = ... \wedge \underbrace{\underbrace{saldo = n}_{borrado} \wedge ... \wedge \underbrace{saldo = n + m}_{introduccion}}_{actualizacion} \wedge ... \tag{5.17}$$

El estado σ_{i+1} se alcanza tras la actualización correspondiente dada en la fórmula ϕ'. La actualización significa que el par (atributo,valor) relevante en la ejecución del evento ingresar(m) (en este caso saldo=n), se reemplaza por el nuevo par (atributo,valor) relevante en el estado alcanzado (saldo=n+m).

De esta forma la traza de ejecución del programa queda caracterizada a nivel de estado como,

$$\{..., \underbrace{(saldo = n), ...}_{\sigma_i}\}, \{..., \underbrace{(saldo = n + m)}_{\sigma_{i+1}}, ...\} \tag{5.18}$$

y a nivel de mundo, o estructura de interpretación como,

$$\omega_i = \{..., (saldo, n), ...\} \tag{5.19}$$

$$\omega_{i+1} = \{..., (saldo, n + m), ...\} \tag{5.20}$$

5.2.2 Derivaciones

El cálculo presentado en la definición 5.2.1 también es válido para calcular el valor de los atributos derivados. Las derivaciones son fórmulas del tipo $\phi \rightarrow \phi'$. Dichas fórmulas permiten definir atributos derivados en términos de una condición de derivación definida en ϕ'. La derivaciones difieren de las fórmulas de evaluación en que mientras las primeras se evalúan en un único mundo, las segundas necesitan de más de uno debido a que describen un cierto cambio de estado. En este caso σ_i representa los estados anterior y posterior a la aplicación de la fórmula de derivación y se evalúa sobre un mismo mundo ω_i.

Ejemplo 5.2.2. Supongamos una derivación de la siguiente forma: "la cuota mensual del préstamo a pagar se calcula dividiendo el precio del coche entre los meses en que se desea pagar y multiplicando por 0.06."

La fórmula de derivación asociada al atributo cuota es,

$$\underbrace{cuota = (coche.precio/meses) * 0.06}_{\phi'} \tag{5.21}$$

Aplicando el cálculo de la definición 5.2.1 tenemos que,

$$\sigma_i = ... \wedge cuota = (coche.precio/meses) * 0.06 \wedge ... \tag{5.22}$$

Siendo $cuota = (coche.precio/meses) * 0.06 \in \phi'$

5.2.3 Precondiciones

Son aquellas fórmulas con el esquema $\neg\phi\ [e]\ false$, donde ϕ es una fórmula que debe cumplirse en el mundo anterior a la ejecución del evento e.

Definición 5.2.2. *Sea ω_i el mundo actual tal que $\omega_i \in \tau(\neg\phi)$ entonces se cumple que,*

$$\nexists\ \omega_{i+1} \in \mathcal{W}/(\omega_i, \omega_{i+1}) \in \rho(e) \qquad con \qquad e \in X \tag{5.23}$$

Es decir, si ω_i representa el mundo actual en el que nos encontramos tal que ese mundo pertenece al conjunto de mundos posibles que no satisfacen la fórmula ϕ, entonces no existe ningún mundo que pueda ser alcanzado tras la ocurrencia del evento e. Con esto estamos impidiendo que ciertos eventos ocurran cuando un objeto se encuentra en un conjunto determinado de estados, en definitiva, en aquellos en donde la fórmula ϕ no se cumple.

Teniendo en cuenta estos fundamentos podemos definir dentro del cálculo la siguiente propiedad que debe de cumplir toda precondición.

Propiedad 5.2.1. $\sigma_i \rightarrow \neg\phi, \models_{\omega_i} \sigma_i \Rightarrow \nexists\, \omega_{i+1} \in \mathcal{W} / \models_{\omega_{i+1}} \sigma_{i+1}$

Esta propiedad refleja el hecho de que si σ_i es un estado válido en el mundo ω_i entonces no existe ningún mundo posible tal que se cumpla que el estado σ_{i+1} obtenido aplicando el cálculo de la definición 5.2.1, satisface a ese mundo. El acontecer del evento asociado a la evaluación implicada está prohibido en w_i. No decimos nada de los mundos w_k tales que $w_k \in \tau(\phi)$. Asumimos la semántica por defecto de que lo que no está prohibido, está permitido. Si además tenemos una evaluación tal que $\phi[e]\phi'$ que define $\rho(e)$, podrá ejecutarse e trasladando a ω' que cumple $\models_{\omega'} \phi'$ y $(w_k, w') \in \rho(e)$.

5.2.4 Restricciones de Integridad

Las restricciones de integridad expresan propiedades que un objeto debe de satisfacer a lo largo de los estados que componen su ciclo de vida (traza de estados). La descripción formal de estas propiedades se realiza a través de fórmulas temporales que pertenecen a una cierta lógica temporal (Lipeck & Saake, 1987), concretamente una extensión de la lógica de primer orden con utilización de operadores temporales.

Intuitivamente, las restricciones son fórmulas que deben ser satisfechas en todos los mundos. A efectos de especificación se distingue entre restricciones estáticas y restricciones dinámicas. Real-

mente las primeras son un caso particular de las segundas, concretamente aquellas que responden a una fórmula temporal de tipo always ϕ. Esto significa que las restricciones estáticas deben de ser satisfechas en el conjunto de mundos posibles alcanzables durante la vida de un objeto.

Definición 5.2.3. *Sea ϕ una fórmula de restricción estática. Sea ω_i un mundo posible tal que $\omega_i \in \tau(\phi)$, entonces se cumple que,*

$$\forall\ \omega_{i+1} \in \mathcal{W}\ /\ (\omega_i, \omega_{i+1}) \in \rho(e) \Rightarrow \omega_{i+1} \in \tau(\phi) \tag{5.24}$$

La definición anterior permite establecer la siguiente propiedad que deben de cumplir las restricciones de integridad estáticas en el cálculo operacional para el modelo de ejecución.

Propiedad 5.2.2. $\sigma_i \rightarrow \phi,\ \models_{\omega_i} \sigma_i \Rightarrow \forall\ \omega_{i+1} \in \mathcal{W}/\ (\omega_i, \omega_{i+1}) \in \rho(e),\ \models_{\omega_{i+1}} \sigma_{i+1}, \sigma_{i+1} \rightarrow \phi$

Restricciones dinámicas. Las fórmulas temporales se construyen a partir de la lógica de primer orden aplicando de forma iterativa las conectivas lógicas y los operadores temporales de la lógica temporal.

En (Lipeck & Saake, 1987) se demuestra que cada operador temporal puede representarse como una fórmula equivalente en donde se distingue una fórmula atemporal que puede ser comprobada en el primer estado de la vida de un objeto y una fórmula temporal precedida por el operador de transición next que será comprobado en los siguientes estados del ciclo de vida del objeto. Estas equivalencias son de la forma:

- always p $\Leftrightarrow$ p $\wedge$ next always p
- sometime p $\Leftrightarrow$ p $\vee$ next sometime p
- always p before q $\Leftrightarrow$ q $\vee$ (p $\wedge$ next (always p before q))
- always p until q $\Leftrightarrow$ (p $\wedge$ q) $\vee$ (p $\wedge$ next (always p until q))
- sometime p before q $\Leftrightarrow$ (p $\wedge \neg$q) $\vee$ ($\neg$q $\wedge$ next (sometime p before q))

- sometime p until q $\Leftrightarrow$ (p $\vee$ ($\neg$q $\wedge$ next (sometime p until q))
- always p since q $\Leftrightarrow$ (p $\wedge$ q $\wedge$ next (always p)) $\vee$ ($\neg$q $\wedge$ next (always p since q))
- always p after q $\Leftrightarrow$ (q $\wedge$ next (always p)) $\vee$ ($\neg$q $\wedge$ next (always p after q))
- sometime p since q $\Leftrightarrow$ (p $\wedge$ q) $\vee$ ($\neg$p $\wedge$ q $\wedge$ next (sometime p)) $\vee$ ($\neg$q $\wedge$ next (sometime p since q))
- sometime p after q $\Leftrightarrow$ (q $\wedge$ next (sometime p)) $\vee$ ($\neg$q $\wedge$ next (sometime p after q))

A partir de las equivalencias anteriores se pueden construir grafos de transición (Pelechano *et al.*, 1997) que hacen posible la comprobación de las restricciones de integridad dinámicas (rid's).

Intuitivamente, los arcos del grafo se etiquetan con una fórmula atemporal y los nodos destino con una fórmula temporal, de manera que cada vez que ocurra un cambio de estado, la comprobación de las rid's se reduce a la comprobación de una condición estática representada por la fórmula atemporal.

En (Lipeck & Feng, 1989) se demuestra como cada etiqueta de un nodo de un grafo de transición (fórmula temporal), es válida en una secuencia de estados si existe un arco saliente cuya etiqueta asociada es válida en el primer estado de esa secuencia y la etiqueta asociada al nodo destino es válida en la subsecuencia de estados restantes.

5.2.5 Disparos

Son fórmulas de la forma, ϕ $[\neg e]$ $false$. $\neg e$ es la negación del evento. Quiere decir que e no ocurre, y que no está especificado qué ocurre. La diferencia principal entre precondiciones y disparos viene dada por el hecho de que en los disparos hay una obligación de activar un evento tan pronto como la condición sea satisfecha. Los disparos permiten de esta forma introducir actividad interna en la sociedad de objetos que está siendo modelada.

Definición 5.2.4. *Sea ω_i el mundo actual tal que $\omega_i \in \tau(\phi)$. Entonces se cumple que,*

$$\exists!\ \omega_{i+1} \in \mathcal{W}/(\omega_i, \omega_{i+1}) \in \rho(e) \qquad con \qquad e \in X \tag{5.25}$$

y además,

$$\nexists\ e' \in X/(w_i, w_{i+1}) \in \rho(e'), \qquad e \neq e' \tag{5.26}$$

En este caso, si ω_i representa el mundo actual en el que nos encontramos tal que ese mundo pertenece al conjunto de mundos posibles que satisfacen la fórmula ϕ, entonces existe un único mundo que puede ser alcanzado tras la ocurrencia del evento e. Con esto estamos forzando a que el evento e ocurra cada vez que el estado del objeto sea tal que la fórmula ϕ se cumpla.

Podemos por tanto definir la siguiente propiedad para los disparos.

Propiedad 5.2.3. $\sigma_i \rightarrow \phi, \models_{\omega_i} \sigma_i \Rightarrow \exists!\ \omega_{i+1} \in \mathcal{W}/ \models_{\omega_{i+1}} \sigma_{i+1}\ y,$
$\nexists\ e' \in X/(w_i, w_{i+1}) \in \rho(e')$

Esta propiedad expresa el hecho de que si σ_i es un estado válido en el mundo ω_i y $\sigma_i \rightarrow \phi$, entonces existe un único mundo ω_{i+1} que cumple que el estado σ_{i+1} obtenido tras la aplicación del cálculo de la definición 5.2.1, es válido en ese mundo.

5.2.6 Especificación de Proceso

En OASIS (versión 2.2) el lenguaje utilizado para especificar procesos es un subconjunto de CCS[2] (Milner, 1989) cuya semántica está basada en la noción de sistema de transición etiquetado y propiedades formales establecidas en CCS.

De acuerdo con el planteamiento y notación propuestos en CCS, tal y como se presenta en (Letelier *et al.*, 1998), la especificación

[2] Calculus of Concurrent Systems

de un proceso puede ser interpretada como un sistema de transición etiquetado representado por un conjunto de constantes de agente utilizados para definir el proceso, un conjunto de acciones pertenecientes a la signatura de la clase que se usan en la especificación del proceso y un conjunto de relaciones de transición. A cada constante de agente se le asocia una ecuación que se construye utilizando el conjunto de operadores de CCS.

En una plantilla de clase OASIS, para cada clase P existe implícitamente un atributo variable de sort $\mathcal{K}_P$ con nombre igual a la clase. Durante la evolución de un programa, un objeto se encontrará en un estado determinado en la especificación de proceso. A este estado lo denominaremos estado del proceso.

Los diferentes estados del proceso son clases de equivalencia en el sentido de que son estados que captan cierta semántica del dominio, es decir, varios estados de un objeto pueden pertenecer al mismo estado del proceso.

Veamos como se tratan las especificaciones de proceso en el cálculo operacional para el modelo de ejecución.

Definición 5.2.5. *Sea $\mathcal{K}_P$ el conjunto de estados de proceso utilizados para definir el proceso de la clase P. Sea $\omega_i \in \tau(\phi)$ el mundo actual, y sea $\models_{\omega_i} \sigma_i$. Diremos que σ_i pertenece al estado de proceso P_i si σ_i pertenece a la clase de equivalencia de P_i.*

Las relaciones de equivalencia suelen ser requisitos que captan semántica del Universo del Discurso. Por ejemplo, en el caso del sistema de información de la biblioteca podríamos diferenciar entre un estado de proceso P_1 etiquetado como 'con libros' que satisface un requisito del tipo numLibros > 0 y otro estado de proceso P_2 etiquetado como 'sin libros' que satisface el requisito numLibros $= 0$.

$$\sigma_i \in P_i \Rightarrow [\sigma_i]^{P_i} \tag{5.27}$$

La definición anterior permite caracterizar la siguiente propiedad para comprobar si una transición de estado inducida por una evaluación es válida en la especificación de proceso.

Propiedad 5.2.4. $[\sigma_i]^{P_i} \Rightarrow \exists\ P_{i+1} \in \mathcal{K}_P\ /\ [\sigma_{i+1}]^{P_{i+1}}$

Esta propiedad establece que siendo σ_i el estado actual de un objeto, y siendo σ_{i+1} el estado que se obtiene aplicando el cálculo de la definición 5.2.1, la correspondiente transición de estado es válida para una especificación de proceso si σ_{i+1} pertenece a la clase de equivalencia de algún estado de proceso $P_{i+1} \in \mathcal{K}_P$.

5.3 De la especificación a la implementación

El cálculo operacional definido en la sección anterior caracteriza el modelo de ejecución y proporciona un mecanismo para poder efectuar el proceso de traducción. Los elementos relevantes de este proceso son los siguientes:

- un cálculo concreto (ecuación 5.12) que establece, dada una evaluación OASIS, cómo se produce el cambio de estado en la implementación. Esto es, como se pasa de un estado σ_i al siguiente σ_{i+1}.
- una serie de propiedades (propiedades 5.2.1, 5.2.2, 5.2.3, 5.2.4) que se deben de cumplir en el cálculo.

De esta forma, el cálculo presentado en este capítulo puede ser visto como una aplicación (h) entre conjuntos (ver Figura 5.2).

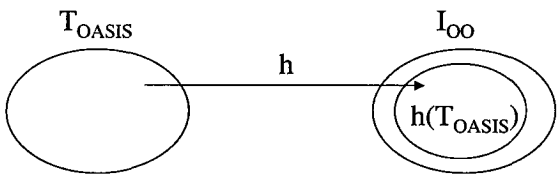


Figura 5.2. *Aplicación entre conjuntos*

Donde los conjuntos involucrados son:

- un conjunto a nivel del espacio del problema que viene representado por una especificación OASIS cuya sintaxis y semántica se formaliza en el contexto de la lógica dinámica (T_{oasis}).
- un conjunto a nivel del espacio de la solución que viene representado por una implementación imperativa y orientada al objeto en un lenguaje de programación (I_{oo}).

Para demostrar formalmente la equivalencia semántica entre ambos conjuntos se debe de:

- terminar de especificar el conjunto I_{oo} con ayuda de algún formalismo,
- establecer un conjunto de leyes de composición interna entre sus elementos y finalmente,
- demostrar que la aplicación h que ha sido definida mediante el cálculo operacional presentado en este capítulo es un homomorfismo.

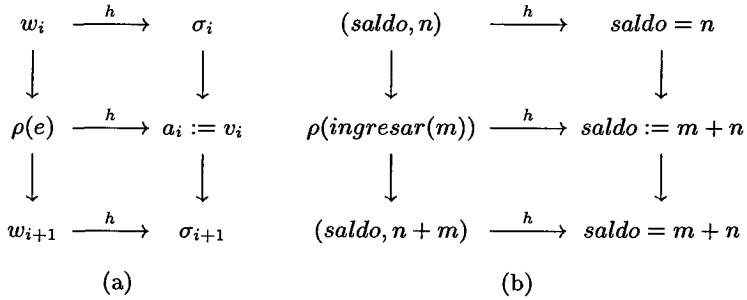


Figura 5.3. *Equivalencia Semántica*

Si se consigue demostrar esto último, entonces, el esquema conmutativo de la Figura 5.3(a) se cumpliría. Tal y como se ilustra en la Figura 5.3(b), si partimos de un elemento válido del conjunto T_{oasis} como por ejemplo (saldo,n) y le aplicamos un operador válido de ese conjunto (por ejemplo, ρ(ingresar(m))), lo que obtenemos es un nuevo elemento que denota el estado resultante

((saldo,n+m)). Si a ese elemento obtenido le aplicamos el cálculo que hemos descrito en este capítulo (representado por h en el gráfico), lo que se obtiene es un nuevo elemento pero dentro del conjunto I_{oo} (este elemento sería saldo=m+n). El diagrama conmutativo implica que ese mismo elemento podría ser obtenido si se procede primero a aplicar el cálculo a un elemento de T_{oasis} y posteriormente aplicaramos el operador correspondiente sobre el conjunto I_{oo}.

5.4 Conclusiones del Capítulo

Uno de los objetivos más codiciados por cualquier método de generación de código es proporcionar una base teórica sobre la que posteriormente se pueda garantizar la equivalencia semántica entre la especificación formal obtenida tras la fase de modelado conceptual y la implementación generada por el método.

Hasta el momento, no conocemos de ningún método de generación de código (en contextos de producción de software industrial), que se haya preocupado por dar una respuesta a este problema. Pese a que hay mucho trabajo realizado en el campo de la derivación formal de programas lógicos estas estrategias no se usan a la hora de aplicarlas sobre entornos imperativos donde perdemos las propiedades formales de los lenguajes lógicos.

El trabajo presentado en este capítulo pretende sentar las bases para en un futuro dar respuesta a esta exigencia en el contexto del método OO-Method. La clave está en dotar al modelo de ejecución, responsable del proceso de generación de código, de un cálculo operacional que permita representar qué efecto producen las evaluaciones especificadas en el espacio del problema. El cálculo presentado permite especificar en el espacio de la solución cómo se producen los cambios de estado de los objetos durante la invocación de servicios (operaciones) así como establecer qué propiedades deben de cumplirse en los nuevos estados alcanzados.

Es importante destacar que con este capítulo no se ha pretendido realizar la demostración formal de la equivalencia semántica entre especificaciones e implementaciones (ese trabajo por sí solo puede ser catalogado como otro trabajo de tesis), sino abrir el camino para que esta demostración nada trivial pueda, en un futuro, ser finalizada.

6. Conclusiones Finales

Los avances tecnológicos que durante estos últimos años se han producido en el campo de las nuevas tecnologías de la información han provocado que el estilo y la arquitectura de las aplicaciones software cambie significativamente. En este sentido, la construcción de tales aplicaciones requiere de nuevos métodos y herramientas que soporten aspectos tales como computación distribuida, arquitecturas internet/intranet multicapa, interoperabilidad del software y desarrollo basado en componentes.

El propósito de este trabajo de tesis ha sido tratar estas nuevas necesidades en el contexto de un método de desarrollo de software basado en el paradigma objetual y que centra los esfuerzos de desarrollo en la fase de modelado conceptual. Se han presentado un conjunto de constructores básicos de modelado que, definidos rigurosamente según un modelo de objetos formal (el lenguaje de especificación OASIS), permiten obtener una especificación de los requisitos del sistema altamente expresiva a partir de la cual es posible generar de forma automática componentes software. Estos componentes adecuadamente combinados, constituyen una aplicación software directamente ejecutable sobre entornos internet/intranet distribuidos.

Se ha definido un modelo de ejecución abstracto que establece una estrategia concreta para guiar el proceso de generación en base a tres pilares fundamentales:

- cada concepto de modelado en el espacio del problema tiene su correspondiente representación software en el espacio de la solución.

- la representación software está basada en separar claramente la especificación del componente de la implementación del mismo.
- la propuesta de una arquitectura para el modelo de ejecución que proporcione el contexto para ejecutar la aplicación obtenida.

El modelo de ejecución así definido ha sido implementado en forma de un componente software que se integra en el contexto de una herramienta CASE (CASE/OO-Method) y que se usa en la fase de generación de código.

Se ha diseñado e implementado una herramienta (JEM) que permite comprobar la aplicaciones generadas sobre entornos internet/intranet distribuidos y validar el proceso de generación de componentes. JEM se complementa con la CASE/OO-Method proporcionando un entorno avanzado de producción automática de software basada en modelos conceptuales.

Una de las características deseables que deberían contemplar los métodos de generación automática de código, es proporcionar algún mecanismo para comprobar la consistencia del código que se genera respecto a los requisitos iniciales del sistema. Hasta donde nosotros sabemos, no conocemos de ningún método que satisfaga completamente este requisito tan importante. En este sentido, otra contribución de este trabajo ha sido la definición de un cálculo operacional para el modelo de ejecución que, estableciendo de manera precisa los efectos durante la ejecución de un programa, permite sentar las bases para intentar demostrar que existe una relación homomórfica entre la especificación de requisitos que se captura en el paso de modelado conceptual y la implementación que genera el modelo de ejecución. Lo que nos permitiría asegurar que la implementación generada es semánticamente equivalente a la especificación formal del sistema.

A continuación se proporciona un resumen de las principales aportaciones de esta tesis.

6.1 Aportaciones

- Una extensión del método OO-Method para soportar la producción de componentes software.
- Una estrategia para generar componentes software a partir de modelos conceptuales orientados a objetos basada en establecer correspondencias entre conceptos de modelado y representaciones software.
- La propuesta de una arquitectura software basada en componentes que permite ejecutar el producto software que se obtiene en el espacio de la solución.
- La presentación de un cálculo que permita sentar las bases para intentar la demostración de la equivalencia semántica entre especificaciones de requisitos y la correspondiente implementación obtenida.
- El diseño e implementación de dos herramientas para mejorar el ciclo de desarrollo de software sobre ambientes CASE:
  i. un generador de código basado en el modelo de ejecución presentado y su arquitectura asociada.
  ii. la herramienta JEM, que permite validar el proceso de generación de componentes sobre entornos internet/intranet distribuidos.

6.2 Trabajos en Progreso

Los problemas inherentes a la generación de código basada en modelos conceptuales no están ni mucho menos resueltos. Aunque con esta tesis se ha pretendido dar un impulso en este terreno, todavía queda mucho por hacer. Los trabajos que nos proponemos abordar en un futuro tratan de dar pasos en cuatro líneas fundamentales de actuación:

- Lenguajes de Descripción de Arquitecturas Software

El concepto de arquetipo está llamado a convertirse en un componente esencial de los métodos de generación basados en modelos. Un arquetipo es un modelo de arquitectura independiente del modelo de aplicación que puede ser reusado. En cierto modo, la arquitectura propuesta en esta tesis para el modelo de ejecución representa el arquetipo sobre el que estamos interesados en generar nuestras aplicaciones (arquitecturas web de tres capas basada en componentes). Los arquetipos se construyen mediante lenguajes de descripción de arquitecturas software como por ejemplo LEDA (Canal *et al.*, 1999). Estos lenguajes describen conectores y puertos a través de los cuales se ofrecen o piden servicios. En el contexto de nuestro grupo de investigación, estamos interesados en ampliar las posibilidades de OASIS como lenguaje de descripción de arquitecturas software. Evidentemente, este estudio llevaría a extender las características del lenguaje para que soportara la posibilidad de describir arquetipos. El beneficio es claro, bajo un mismo marco formal (OASIS) podemos especificar aspectos vinculados tanto a los modelos de aplicación como a los modelos de arquitectura. Otras aproximaciones como OBLOG están diseñando su lenguaje de arquetipos de forma independiente al lenguaje que utilizan para especificar los requisitos del sistema, con el consiguiente peligro implícito de proporcionar al modelador demasiadas notaciones difíciles de manejar.

- Gestión del proceso de reutilización de componentes

  Aunque los principales esfuerzos de esta investigación se han centrado en la generación automática de componentes, otro aspecto clave a considerar, consiste en el estudio y propuesta de marcos teóricos que permitan implantar el concepto de reutilización como práctica estándar en el proceso de desarrollo del software. En este sentido, se hace necesario el desarrollo de nuevas propuestas que proporcionen técnicas para manejar adecuadamente los componentes generados desde dos perspectivas:

  – métricas que permitan establecer estrategias de clasificación de componentes.

– técnicas de búsqueda y recuperación a partir de repositorios de componentes software.

Dentro de este contexto, en nuestro grupo de investigación se están realizado estudios (Anaya, 1999) que permitan la gestión de componentes reusables en el entorno OASIS.

- Modelado Conceptual del Web

  La necesidad de desarrollar un nuevo tipo de aplicación conocidas como aplicaciones Internet/Intranet, ha provocado que recientemente, exista un interés creciente en cómo generar Sistemas de Información que combinen de forma correcta dos aspectos básicos: estrategias de navegación en un espacio heterogéneo de información, y operaciones de consulta o actualización de la base de datos heterogénea correspondiente.

  El estudio de los problemas asociados al diseño de este tipo de aplicaciones complejas en ambientes web, que incluyan comportamiento dinámico, y que combinen adecuadamente el diseño e implementación de patrones de navegación precisos con un sofisticado comportamiento computacional, va a ser objeto de un intenso trabajo de investigación y desarrollo durante los próximos años.

  En este sentido los métodos convencionales OO presentan una grave carencia generalizada: el no tratamiento de abstracciones que hagan posible la especificación de aplicaciones hipermediales consecuentes con la metáfora del hipertexto. En particular, el diseño navegacional, característica esencial de las aplicaciones basadas en web, no se tiene presente.

  Actualmente desde nuestro grupo de investigación se trabaja en extensiones al método OO-Method que soporten este nuevo tipo de requisitos. Los avances más importantes se han producido en la elaboración y propuesta de un nuevo diagrama que se integra en la fase de modelado conceptual del método denominado diagrama de acceso navegacional (DAN) que hereda muchos conceptos de la aproximación HDM-Lite (Fraternali & Paolini, 1998).

- Modelado Conceptual de Bases de Datos Multidimensionales

  Otro aspecto relevante que últimamente aparece muy vinculado a los sistemas de información, es el de disponer de forma organizada de toda la información 'hetereogénea' que se manipula en un sistema organizacional de forma que sea posible realizar análisis sobre esa información y consecuentemente se puedan adoptar estrategias y decisiones sobre las reglas de negocio que gobiernan el sistema organizacional.

  En este contexto, las bases de datos multidimensionales junto con los almacenes de datos y las técnicas de análisis OLAP[1] son líneas de trabajo sobre las que se están dirigiendo grandes esfuerzos de investigación.

  Concretamente en nuestro grupo de investigación estamos interesados en explorar las posibilidades que ofrece el método OO-Method para soportar el modelado conceptual de bases de datos multidimensionales. En este sentido, se están realizado trabajos en dos líneas básicas:

  - la detección de patrones de modelado que permitan capturar aspectos claves del modelado multidimensional tales como medidas derivadas, atributos de dimensión o jerarquías de clasificación múltiple.
  - extensiones al lenguaje de especificación OASIS para expresar tales conceptos.

  Fruto de estos esfuerzos han surgido algunas publicaciones como la detección de patrones de modelado para representar operaciones OLAP básicas (Trujillo *et al.*, 1999), o la propuesta de una versión extendida de OASIS para la especificación de bases de datos multidimensionales (Trujillo *et al.*, 2000).

Por último, destacar también que se están realizando esfuerzos dentro de nuestro grupo de investigación para tratar el aspecto de temporalidad (Meneses *et al.*, 1999) dentro de los propios modelos conceptuales.

[1] On-Line Analytical Processing

6.3 Publicaciones realizadas

Los resultados presentados en esta tesis han sido contrastados en foros relevantes de investigación como publicaciones en revista, y congresos internacionales de alto nivel.

A continuación se presentan los trabajos publicados durante el transcurso de esta tesis. El resumen de publicaciones es el siguiente: 2 artículos en revista internacional (uno de ellos en período de revisión), 9 artículos en congresos internacionales y 2 artículos en congresos nacionales.

Los artículos se agrupan por tipo de publicación y clasificados por orden de relevancia:

i. Artículos en revista internacional

- O. Pastor, E. Insfrán, V. Pelechano, J. Gómez. "Generación Automática de Prototipos en entornos Internet/Intranet a partir de Modelos Conceptuales OO". Revista Computación y Sistemas Volumen 3, Número 1. Pags 38-49. ISSN 1405-5546. July-September 1999.
- O. Pastor, J. Gómez, E. Insfrán, V. Pelechano. "The OO-Method Approach for Information Systems Modeling: From Object-Oriented Conceptual Modeling to Automated Programming". Bajo período de revisión en Information Systems.

ii. Artículos en congresos internacionales

- O. Pastor, V. Pelechano, E. Insfrán, J. Gómez. "From Object-Oriented Conceptual Modeling to Automated Programming in Java". 17th International Conference on Conceptual Modeling (**ER'98**). Singapur. 16-19 Noviembre 1998. Lecture Notes in Computer Science vol. 1507, pags 183-196. Springer-Verlag.
- J. Gómez, O. Pastor, E. Insfrán, V. Pelechano. "From Object-Oriented Conceptual Modeling to Component-Based Development". 10th International Conference on Database and Expert System Applications (**DEXA'99**). Lecture Notes in

Computer Science vol 1677, pags 332-341. Springer-Verlag, 1999.

- J. Gómez. "Component-based Architectures to Generate Software Components From Object-oriented Conceptual Models". In **ECOOP'98** Workshop Reader. Lecture Notes in Computer Science vol 1543, pags 21-22. Springer-Verlag. (Extended abstract).
- J. Gómez, O. Pastor. "Conceptual Design and Development of Applications based on Object Interoperability". In proceedings of ECOOP'99 workshop on Object Interoperability, Pags 77-86. To be published in "**ECOOP'99** Workshop Reader". Lecture Notes in Computer Science. Springer-Verlag. (Extended Abstract).
- J. Gómez, E. Insfrán, V. Pelechano, O. Pastor. "The Execution Model: a Component-based Architecture to Generate Software Components from Conceptual Models". In proceedings of **CAiSE'98** workshop on Component-based Information Systems Engineering, Pags 87-93.
- J. Gómez. "Decreasing the Gap between Formal Specifications Languages and Component-based Development". In proceedings of **ECOOP'98** workshop for PhD. students in Object-Oriented Systems. Bruselas, Bélgica. 20-24 Julio 1998.
- J. C. Trujillo, M. Palomar, J. Gómez. "Detecting Patterns and OLAP Operations in the GOLD Model". To be published in the proceedings of the ACM Second International Workshop on Data Warehousing and OLAP in conjunction with the 8^{th} International Conference on Information and Knowledge Management (**CIKM'99**). Kansas City, Missouri 1999.
- J. C. Trujillo, M. Palomar, J. Gómez. "The GOLD Definition Language (GDL): An Object Oriented Formal Specification Language for Multidimensional Databases". Accepted for publication in the ACM proceedings of the 15^{th} ACM

Symposium on Applied Computing (**SAC'2000**). Villa Olmo, Como, Italy. March 19-21, 2000.

- J. Devesa, J. Gómez, O. Pastor. "OO-Method Distribuido y Compatible CORBA (OOMDC)". En actas del II Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (**IDEAS'99**). Pags 84-96.

iii. Artículos en congresos nacionales

- O. Pastor, C. Heuser, J. Gómez, E. Insfrán. "Ingeniería de Ambientes Software: Combinando Expresividades Temporales y Objetuales en la Fase de Modelización Conceptual". En actas de las Jornadas nacionales en Ingeniería del Software (**JIS'97**), Pags 53,67. 1997.
- E. Insfrán, V. Pelechano, J. Gómez, O. Pastor. " Un Estudio Comparativo de la Expresividad de Relaciones entre Clases en OO-Method y UML". Actas de las III Jornadas **MENHIR**. Pags 13-24. 1998.

Referencias

Aho, Alfred V., Sethi, Ravi, & Ullman, Jeffrey D. 1986. *Compilers: Principles, Techniques and Tools.* Reading, MA: Addison Wesley.

Anaya, R. 1999 (Octubre). *Desarrollo y Gestión de Componentes Reusables en el Entorno OASIS.* Ph.D. thesis, Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia.

Andrade, L. F., Gouveia, J. C., & Xardonne, P. J. 1998. Architectural Concerns in Automating Code Generation. *In: OOPSLA midyear conference.*

Balzer, R., Cheatham, T. E., & Green, C. 1984 (Feb.). Software technology in the 1990's: using a new paradigm. *Pages 3–9 of:* Potts, C. (ed), *Proceedings of the* 1^{st} *International Software Process Workshop.*

Bauer, F., Moller, B., Partsch, H., & Pepper, P. 1989. Formal Program Construction by Transformations - Computer-Aided, Intuition-Guided Programming. *IEEE Transactions on Software Engineering*, **15**(February), 165–180.

Bauer, F. L., Berghammer, R., Broy, M., Dosch, W., Geiselbrechtinger, F., Gnatz, R., Hangel, E., Hesse, W., Krieg-Brückner, B., Laut, A., Matzner, T., Möller, B., Nickl, F., Partsch, H., Pepper, P., Samelson, K., Wirsing, M., & Wössner, H. 1985. *The Munich Project CIP, Vol. 1: The Wide Spectrum Language CIP-L.* Springer LNCS 183.

Bauer, F. L., Ehler, H., Horsch, R., Möller, B., Partsch, H., Paukner, O., & Pepper, P. 1987. *The Munich project CIP. Volume*

II: The Transformation System CIP-S. Springer LNCS 292.

Bell, R. 1998. Code Generation from Object Models. *Embedded Systems Programming*, 1–9.

Berzins, V., Yehudai, L., & Yehudai, A. 1993. Using Transformations in Specification-Based Prototyping. *IEEE Transactions on Software Engineering*, **19**(5), 436–452.

Boehm, Barry. 1999. Management: Managing Software Productivity and Reuse. *Computer*, **32**(9), 111–113.

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

Booch, G. 1994. *Object Oriented Analysis and Design with Applications. Second Edition.* Addison-Wesley.

Booch, G., Rumbaugh, J., & Jacobson, I. 1997. *UML v1.* Tech. rept. Rational Software Corporation.

Booch, Grady, Rumbaugh, James, & Jacobson, Ivar. 1998. *The Unified Modeling Language User Guide.* Addison-Wesley.

Box, D. 1997. *Creating Components with DCOM and C++.* Addison-Wesley.

Brodie, M. L., Mylopoulos, J., & Schmidt, J. W. 1984. *On Conceptual Modelling - Perspectives from Artificial Intelligence, Databases and Programming Languages.* Springer-Verlag.

Brown, A. W. 1998. From Component-based Infrastructure to Component-based Development. *Pages 87–93 of:* Brown, A. W., & Wallnau, K. (eds), *Proceedings of 2nd. International Workshop on Component-based Software Engineering.*

Brown, A. W., & Jaeger, K. 1998. *The Future of Enterprise Application Development with Components and Patterns.* Tech. rept. Sterling Software.

Brown, K., & Whitenack, B. 1996. *Crossing Chasms. Pattern Languages of Program Design vol. 2.* Addison-Wesley.

Broy, Manfred. 1991. Towards a Formal Foundation of the Specification and Description Language SDL. *Formal Aspects of Computing*, **3**(1), 21–57.

Camara, J. A., & Andrade, L. F. 1999. OBLOG: From Specification to Code Generation. *Pages 360–372 of:* Pastor, O., González, C., Trejos, I., & Insfrán, E. (eds), *II Workshop Iberoamericano en Ingeniería de Requisitos y Ambientes Software (IDEAS'99). Costa-Rica.*

Camara, J. A., Gouveia, J. C., & Andrade, L. F. 1999. *Object-Oriented Implementation using State Machines.* Technical report. OBLOG Software.

Canal, C., Pimentel, E., & Troya, J. M. 1999. Specification and Refinement of Dynamic Software Architectures. *Pages 107–125 of:* Donohoe, P. (ed), *Software Architecture (Proc. of WICSA'99), Kluwer Academic Publishers.*

Canós, J. H. Hilario. 1996 (Octubre). *OASIS: Un Lenguaje Unico para Bases de Datos Orientadas a Objetos.* Ph.D. thesis, Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia.

Carsí, JA. 1999 (Junio). *OASIS como Marco Conceptual para la Evolución del Software.* Ph.D. thesis, Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia.

Chen, P.P. 1976. The Entity-Relationship Model: Towards a Unified View of Data. *ACM Transactions on Database Systems*, **1**(1).

Clyde, SW., Embley, DW., & Woodfield, SN. 1992. Tunable Formalism in Object-Oriented Systems Analysis: Meeting the Needs of Both Theoreticians and Practitioners. *Pages 452–465 of: Proceedings of the OOPSLA '92 Conference on Object-oriented Programming Systems, Languages and Applications.*

Coad, P., & Yourdon, E. 1990. *Object-Oriented Analysis.* Yourdon Press. Englewood Cliffs, NJ.

Coleman, D., Arnold, P., Bodoff, S., Dollin, S., Gilchrist, H., Hayes, F., & Jeremes, P. 1994. *Object-Oriented Development: The Fusion Method.* Prentice-Hall.

Collins, A., & Smith, E. 1988. *Readings in Cognitive Science: A Perspective from Psychology and Artificial Intelligence.* Morgan-Kaufmann.

DeMarco, T. 1979. *Structured Analysis and Systems Specification.* Prentice-Hall.

Devesa, J., Gómez, J., & Pastor, O. 1999. OO-Method distribuido y compatible CORBA (OOMDC). *Pages 84–96 of: II Workshop Iberoamericano en Ingeniería de Requisitos y Ambientes Software (IDEAS'99). Costa-Rica.*

Downing, T. B. 1998. *Java RMI: Remote Method Invocation.* IDG Books.

D'Souza, Desmond, & Wills, Alan. 1998. *Objects, Components and Frameworks With UML: The Catalysis Approach.* Addison-Wesley.

Dubois, E., Petit, M., & Wu, S. 1994. ALBERT: A Formal Agent-Oriented Requirements Language for Distributed Composite Systems. *Pages 25–39 of: Proceedings of CAiSE'94 Workshop on Formal Methods for Information Systems Dynamics.*

Ehrich, H. D., Caleiro, C., Sernadas, A., & Denker, G. 1998. *Logics for Specifying Concurrent Information Systems.* In J. Chomicki and G. Saake (eds.) Logics for Database and Information Systems, pages 167-198. Kluwer Academic.

Elizondo, A. J. 1998 (Junio). *Reglas Activas: soporte y manejo en las bases de datos orientadas a objetos.* Ph.D. thesis, Departamento de Lenguajes y Sistemas Informáticos y Computación. Universidad del País Vasco.

Embley, DW., Kurtz, BD., & Woodfield, SN. 1992. *Object-Oriented System Analysis: A Model-Driven Approach.*

Prentice-Hall.

Fichman, R G., & Kemerer, C F. 1992. Object-Oriented and Conventional Analysis and Design Methodologies. *Computer*, **25**(10), 22–39.

Fowler, M., & Scott, K. 1997. *UML Distilled: Applying the Standard Object Modeling Language.* The Addison-Wesley Object Technology Series. Addison Wesley Longman, Inc.

Fraternali, P., & Paolini, P. 1998. A Conceptual Model and a Tool Environment for Developing more Scalable, Dynamic, and Customizable Web Applications. *Pages 421–435 of: Advances in Database Technology - EDBT'98*, vol. 1377. Lecture Notes in Computer Science.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. 1994. *Desing Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Gane, C., & Sarson, TL. 1979. *Structured Systems Analysis: Tools and Techniques.* Englewood Cliffs, New Jersey: Prentice-Hall.

Gómez, J. 1998a. Component-based Architectures to Generate Software Components from Object-Oriented Conceptual Models. *Pages 21–22 of: ECOOP'98 Workshop Reader.* Lecture Notes in computer Science, vol. 1543. Springer.

Gómez, J. 1998b. Decreasing the Gap between Formal Specification Languages and Component-based Development. *Pages 50–61 of: Proceedings of* 8^{th} *ECOOP'98 Workshop for PhD Students in Object-Oriented Systems.* University of Aarhus.

Gómez, J., & Pastor, O. 1999a. Conceptual Design and Development of Applications based on Object Interoperability. *In:* (Vallecillo *et al.*, 1999).

Gómez, J., & Pastor, O. 1999b. *Improving Conceptual Modeling Methodologies to Develop Distributed Object Applications.* Tech. rept. DLSI. Universidad de Alicante.

Gómez, J., Pastor, O., Insfrán, E., & Pelechano, V. 1999. From Object-Oriented Conceptual Modeling to Component-Based

Development. *Pages 332–341 of:* Bench-Capon, Trevor, Soda, Giovanni, & Tjoa, A Min (eds), *Proceedings of DEXA'99 International Conference.* LNCS, vol. 1677. Springer-Verlag.

Gómez, J., Insfrán, E., Pelechano, V., & Pastor, O. 1998. The Execution Model: A Component-Based Architecture to Generate Software Components from Conceptual Models. *Pages 87–93 of:* Grundy, J. (ed), *Proceedings of CAiSE'98 Workshop on Component-Based Information Systems Engineering.*

Grau, A., & Kowsari, M. 1997. A Validation System for Object Oriented Specifications of Information Systems. *Pages 277–290 of:* Manthey, R., & Wolfengagen, V. (eds), *Proceedings of ADBIS'97 Symposium.*

Grau, A., Filipe, J. K., Kowsari, M., Eckstein, S., Pinger, R., & Ehrich, H. D. 1998. The TROLL Approach to Conceptual Modelling: Syntax, Semantics and Tools. *Pages 277–290 of:* Ling, T. W., Ram, S., & Lee, M. L. (eds), *Proceedings of 17th International Conference on Conceptual Modeling (ER'98).* LNCS, vol. 1507. Springer-Verlag.

Harel, D. 1984. Dynamic Logic. *Chap. II.10, pages 497–604 of:* Gabbay, D., & Guenthner, F. (eds), *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic.* Synthese Library, vol. 165. Dordrecht: D. Reidel Publishing Co.

Harel, D., & Naamad, A. 1996. The STATEMATE Semantics of Statecharts. *ACM Transaction on Software Engineering and Methodology*, **5**(4), 293–333.

Insfrán, E., Pelechano, V., Gómez, J., & Pastor, O. 1997. Un Estudio Comparativo de la Expresividad de Relaciones entre Clases en OO-Method y UML. *Pages 13–24 of: Actas de las III Jornadas MENHIR.* Universidad de Murcia.

Jackson, M. 1983. *System Development.* Prentice-Hall.

Jackson, RB., Embley, DW., & Woodfield, SN. 1994. Automated Support for the Development of Formal Object-Oriented Requirements Specification. *Pages 135–148 of:* Wijers, G.,

Brinkkemper, S., & Wasserman, T. (eds), *Proceedings of CAiSE'94 International Conference.* LNCS, vol. 811. Springer-Verlag.

Jacobson, I. 1993. Is Object Technology Software's Industrial Platform? *IEEE Software*, **10**(1), 24–30.

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

Jacobson, Ivar, Booch, Grady, & Rumbaugh, James. 1999. Excerpt from The Unified Software Development Process: The Unified Process. *IEEE Software*, **16**(3), 82–90.

Javasoft. 1996. *JavaBeans, Version 1.00.* Tech. rept. Sun Corporation.

Jungclaus, R., Saake, G., & Sernadas, C. 1987. Formal Specification of Object Systems. *Pages 60–82 of:* Abramsky, S., & Milbaum, T. (eds), *Proceedings of the TapSoft's 91.* LNCS, vol. 494. Springer-Verlag.

Jungclaus, R., Saake, G., Hartmann, T., & Sernadas, C. 1995. TROLL: A language for object-oriented specification of information systems. *ACM Transactions on Information Systems*, **14**(2), 175–211.

Kozaczynski, W., & Kuntzmann-Combelles, A. 1993. What it Takes To Make OO Work. *IEEE Software*, **10**(1), 20–23.

Kripke, S. 1980. *Naming and Necessity.* Second edn. Basil Blackwell.

Kush, J., Hartel, P., Hartmann, T., & Saake, G. 1992. Gaining a Uniform View of Different Integration Aspects in A Prototyping Environment. *Pages 35–42 of: Proceedings of DEXA'95 International Conference.* LNCS, vol. 978. Springer-Verlag.

Letelier, P. 1999 (Junio). *Animación Automática de Especificaciones OASIS utilizando Programación Lógica Concurrente.* Ph.D. thesis, Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia.

Letelier, P., Ramos, I., Sánchez, P., & Pastor, O. 1998. *OASIS Versión 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto.* Servicio de Publicaciones UPV.

Liddle, S., Embley, DW., & Woodfield, SN. 1995. Unifying Modeling and Programming Through an Active, Object-Oriented, Model-Equivalent Programming Language. *Pages 55–64 of:* Papazoglou, M. P. (ed), *Proceedings of OOER'95 International Conference.* LNCS, vol. 1021. Springer-Verlag.

Liddle, SW. 1995. *Object-Oriented Systems Implementation: A Model-Equivalent Approach.* Ph.D. thesis, Brigham Young University.

Lipeck, U. W., & Feng, D. 1989. Construction of Deterministic Transition Graphs from Dynamic Integrity Constraints. *Pages 166–179 of:* van Leeuwen, J. (ed), *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science.* LNCS, vol. 344. Berlin: Springer.

Lipeck, U. W., & Saake, G. 1987. Monitoring Dynamic Integrity Constraints Based on Temporal Logic. *Information Systems*, **12**(3), 255–269.

Liu, S., Offutt, A. J., Ho-Stuart, C., Sun, Y., & Ohba, M. 1998. SOFL: A Formal Engineering Methodology for Industrial Applications. *IEEE Transactions on Software Engineering*, **24**(1), 24–45.

Maring, Blayne. 1996. Object-oriented Development of Large Applications. *IEEE Software*, **13**(3), 33–40.

Martin, J., & Odell, J. J. 1994. *Análisis y Diseño Orientado a Objetos.* Prentice-Hall.

Mellor, SJ. 1999. *Automatic Code Generation from UML Models.* Tech. rept. Project Technology.

Meneses, C., Pastor, O., Insfrán, E., & Heuser, C. A. 1999. Entornos Automáticos de Producción de Software a partir de Modelos Conceptuales Orientados a Objetos y Temporales. *Pages 565–577 of:* Universidad Autónoma de Asunción,

Universidad Católica Nuestra Sra.de la Asunción. Paraguay (ed), *Actas de la XXV Conferencia Latinoamericana de Informática CLEI-99.*

Meyer, Bertrand. 1997. *Object-Oriented Software Construction, 2nd Ed.* Second edn. Englewood Cliffs, NJ 07632, USA: Prentice-Hall.

Microsoft. 1996. *The COM Specification.* Tech. rept. Microsoft Corporation.

Milner, R. 1989. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall. SU Fisher Research 511/24.

Mosses, P. D. 1979. *SIS- Semantics Implementation System: Reference Manual and User Guide.* Tech. rept. DAIMI MD-30. Computer Science Department. University of Aarhus.

Mylopoulos, J. 1998. Information Modeling in the Time of the Revolution. *Information Systems*, **23**(3/4), 127–155.

Mylopoulos, John, Chung, Lawrence, & Yu, Eric. 1999. From Object-Oriented to Goal-Oriented Requirements Analysis. *Communications of the ACM*, **42**(1), 31–37.

Oblog. 1999. *The OBLOG Software Development Approach.* Tech. rept. OBLOG Software.

OMG. 1997. *The Common Object Request Broker: Architecture and Specification.* Tech. rept. Object Management Group.

Parsch, H. A. 1990. *Specification and Transformation of Programs: A Formal Approach to Software Development.* Springer-Verlag.

Pastor, O. 1992 (Mayo). *Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el Modelo Orientado a Objetos.* Ph.D. thesis, Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia.

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

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

Pastor, O., Heuser, C., Gómez, J., & Insfrán, E. 1997a. Ingeniería de Ambientes Software: Combinando Expresividades Temporales y Objetuales en la Fase de Modelización Conceptual. *Pages 53–67 of: Actas de las II Jornadas Nacionales en Ingeniería del Software (JIS'97).*

Pastor, O., Insfrán, E., Quiles, G., & Barberá, J. 1997b. Object-Oriented Conceptual Modeling Techniques to Design and Implement a sound and robust ORACLE Environment. *In:* ORACLE (ed), *Proceedings of European Oracle Users Group Conference.*

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

Pastor, O., Insfrán, V. Pelechano E., & Gómez, J. 1998. From Object Oriented Conceptual Modeling to Automated Programming in Java. *Pages 183–196 of:* Ling, T. W., Ram, S., & Lee, M. L. (eds), *Proceedings of 17th International Conference on Conceptual Modeling (ER'98).* LNCS, vol. 1507. Springer-Verlag.

Pastor, O., Insfrán, E., Pelechano, V., & Gómez, J. 1999a. Generación Automática de Prototipos en Entornos Internet/Intranet a partir de Modelos Conceptuales OO. *Revista Computación y Sistemas,* **3**(1), 38–49.

Pastor, O., Gómez, J., Insfrán, E., & Pelechano, V. 1999b. The OO-Method Approach for Information Systems Modeling: From Object-Oriented Conceptual Modeling to Automated Programming. *Submitted to Information Systems.*

Paton, N. W., & Díaz, O. 1999. Active Database Systems. *ACM Computing Surveys*, **31**(1), 63–103.

Pelechano, V., Pastor, O., García, N., & Ramírez, S. 1997. Implementación y Comprobación de Restricciones de Integridad Dinámicas en Entornos de Programación Orientados a Objetos. *Pages 101–116 of:* Díaz, O., & Lopistéguy, P. (eds), *Actas de las II Jornadas en Ingeniería del Software.*

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

Pressman, R. S. 1998. *Software Engineering: A Practical Approach.* Fourth edn. McGraw Hill.

Quatrani, Terry. 1998. *Visual Modeling with Rational Rose and UML.* Addison-Wesley.

Quillian, R. 1968. Semantic Memory. *Pages 227–270 of:* Minsky, M. (ed), *Semantic Information Processing.* MIT Press.

Ramos, I., Pastor, O., Cuevas, J., & Devesa, J. 1992. Objects as Observable Processes. *In: Proceedings of 3rd. Workshop on the Deductive Approach to Information System Design.*

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

Rolland, C. 1998. A Comprensive View of Process Engineering. *Pages 1–24 of:* Pernici, B., & Thanos, C. (eds), *Proceedings of CAiSE'98 International Conference.* LNCS, vol. 1413. Springer-Verlag.

Ross, D., & Schoman, A. 1977. Structured Analysis for Requirements Definition. *IEEE Transactions on Software Engineering*, **3**(1), 6–15.

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

Rumbaugh, James, Jacobson, Ivar, & Booch, Grady. 1998. *The Unified Modeling Language Reference Guide.* Addison-Wesley.

Schulte, R. 1995. *Three-Tier Computing Architectures and Beyond.* Tech. rept. Gartner Group.

Selic, Bran, Gullekson, Garth, McGee, Jim, & Engelberg, Ian. 1992 (6–10 July). ROOM: An Object-Oriented Methodology for Developing Real-Time Systems. *Pages 230–240 of:* Forte, Gene, Madhavji, Nazim H., & Müller, Hausi A. (eds), *5th Int. Work. Computer-Aided Software Engineering.*

Sernadas, A., Sernadas, C., & Ehrich, H. D. 1987. OO Specification of Databases: An Algebraic Approach. *Pages 107–116 of:* Stocker, P. M., & Kent, W. (eds), *Proceedings of VLDB'87 International Conference.*

Shlaer, S., & Lang, N. 1996. *Shlaer-Mellor Method: The OOA96 Report.* Tech. rept. Project Technology Inc., Berkeley, California. Version 1.0.

Shlaer, S., & Mellor, S. J. 1988. *Object-Oriented System Analysis: Modeling the World in Data.* Yourdon-Press.

Shlaer, Sally, & Mellor, Stephen J. 1997. Recursive Design of an Application-Independent Architecture. *IEEE Software,* **14**(1), 61–72.

Snyder, A. 1993. The Essence of Objects: Concepts and Terms. *IEEE Software,* **10**(1), 31–42.

Szyperski, C. 1998. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley.

Trujillo, J. C., Palomar, M., & Gómez, J. 1999. Detecting Patterns and OLAP operations in the GOLD Model. *In: Proceedings of* 2^{nd} *International Workshop on Data Warehousing and OLAP. To be published.* ACM.

Trujillo, J. C., Palomar, M., & Gómez, J. 2000. The GOLD Definition Language (GDL): An Object Oriented Formal Specification Language for Multidimensional Databases. *In: Proceedings of* 15^{nd} *ACM Symposium on Applied Computing. To be published.* ACM.

Vallecillo, A. 1999 (Mayo). *Un Modelo de Componentes para el Desarrollo de Aplicaciones Distribuidas.* Ph.D. thesis, Departamento de Lenguajes y Ciencias de la Computación. Universidad de Málaga.

Vallecillo, A., Hernández, J., & Troya, J. M. (eds). 1999. *Proceedings ECOOP'99 Workshop on Object Interoperability.*

Ward, Paul T., & Mellor, Stephen J. 1985. *Structured Development for Real-Time Systems.* Vol. volume 2: Essential Modeling Techniques. Englewood Cliffs, New Jersey: Yourdon, Inc.

Warmer, Jos, & Kleppe, Anneke. 1998. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley.

Warnier, J. D. 1981. *Logical Design of Systems.* Van Norstrand Reinhold.

Wiederhold, G. 1992. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, **25**(3), 38–49.

Wieringa, R., & Dubois, E. 1998. Integrating Semi-Formal and Formal Software Specification Techniques. *Information Systems*, **23**(3/4), 159–178.

Wieringa, R. J. 1990. *Algebraic Foundations for Dynamic Conceptual Models.* Ph.D. thesis, Vrije Universiteit, Amsterdam.

Wieringa, R. J. 1998 (Amsterdam). *Postmodern Software Design with NYAM: Not Yet Another Method.* Technical report. Faculty of Mathematics and Computer Science, Vrije Universiteit.

Wieringa, R. J., Jungclaus, R., Hartel, P., Saake, G., & Hartmann, T. 1993. OMTROLL: Object Modeling in Troll. *Pages 267–283 of:* Lipeck, U. W., & Koschorreck, G. (eds), *Proceedings*

of IS-CORE 93 Workshop.

Wirfs-Brock, R., Wilkerson, B., & Wiener, L. 1990. *Designing Object Oriented Software.* Prentice-Hall.

Yourdon, E. 1989. *Modern Structured Analysis.* Prentice-Hall.

Yourdon, E., & Constantine, L. L. 1979. *Structured Design: Fundamentals of a discipline of computer program and systems design.* Englewood Cliffs, New Jersey: Prentice-Hall.

Zave, P. 1979. A Comprehensive Approach to Requirements Problems. *Pages 117–127 of: Proceedings of COMPSAC'79 Conference.*

A. Caso de Estudio

En este anexo se presenta la aplicación del método OO-Method (modelo conceptual, especificación OASIS, modelo de ejecución), sobre un caso real de sistemas de información, en concreto el servicio de mantenimiento del hospital general de Alicante.

A.1 Descripción

El caso de estudio es un servicio de un Hospital cuya actividad principal es la gestión de las averías que se producen en él. Este servicio (del que disponen todos los hospitales), se denomina **servicio de mantenimiento**.

Cuando se produce una avería en cualquier servicio del hospital, el personal del servicio afectado comunica la incidencia al servicio de mantenimiento que es el encargado de atenderla.

La gestión de las averías conlleva otra serie de actividades derivadas como son el mantenimiento de las máquinas del hospital, la gestión del personal del servicio, la gestión de los pedidos de las piezas necesarias para la resolución de las averías, etc... que también se desea automatizar. Específicamente tratará:

Averias o Incidencias. Situaciones o avisos que pueden producirse en los servicios del hospital y que normalmente tendrán que ver con el funcionamiento de máquinas o reparaciones en las instalaciones. Las incidencias, una vez notificadas al servicio de mantenimiento, se clasifican en función de:

- prioridad: dependiendo de la gravedad de la avería se le asigna una prioridad alta, media o baja.
- area: dependiendo de la propia naturaleza de la avería, se le asigna un área de resolución.

El estado de una incidencia representa la situación en que ésta se encuentra, es decir:

- en curso: la incidencia ha sido registrada.
- pendiente: la incidencia está pendiente de resolución porque son necesarias ciertas piezas que hay que pedir y no se puede continuar hasta que no se reciban.
- resuelta: la incidencia se ha resuelto, bien porque se ha realizado la reparación o bien porque se ha rechazado su resolución.

Otros aspectos que hay que tener en cuenta son:

- no se puede eliminar una incidencia hasta que no haya sido resuelta.
- la clasificación de una avería sólo la puede realizar el responsable del servicio.

Area. El servicio de mantenimiento se encuentra estructurado en áreas. Cada una de ellas es especialista en la resolución de un tipo de averías. Las áreas son: mecánica, electricidad, electromedicina, carpintería, pintura, A.A./calefacción, fontanería, albañilería.

Personal. Atendiendo a la categoría de los empleados del servicio de mantenimiento, los empleados pueden ser:

- reponsable del servicio: es el jefe de todo el servicio (sólo hay 1).
- maestro: son responsables de las áreas en las que se estructura el servicio, tiene a su disposición un conjunto de operarios para la resolución de las averías. (hay 7 maestros, 1 por área).

- operario: personal cualificado, tiene un turno de trabajo de 8 horas que puede ser mañana, tarde o noche.

Escenario de una avería. Cuando se produce una avería en cualquier servicio del hospital, el usuario del servicio afectado rellena un parte de incidencias (documento 1, ver figura A.1), y lo introduce en alguno de los buzones que se encuentran ubicados por todo el hospital. Diariamente se recogen estos partes y se entregan al responsable del servicio de mantenimiento para que los clasifique según el área de reparación que la va a tratar y la prioridad que precise.

Parte de Incidencia
Orden de Trabajo

Destino	Formulada	Materiales
Mecánica ☐ Electrvcidad ☐ Electromedicina ☐ Carpentería ☐ Pintura ☐ A.A./Calefacción ☐ Fontanería ☐ Albañilería ☐	Servicio _____ Empleado _____	
Descripción	Cód equipo	
Reparación efectuada y causa posible de la avería		
Prioridad Alta ☐ Media ☐ Baja ☐	F. peticion _____ F. realización ____	Operario V°B°

Figura A.1. *Parte de avería y orden de trabajo.*

Cada maestro recibe los partes correspondientes a su área y comprueba con qué operarios cuenta en ese momento, asignando la resolución de la incidencia a cada uno de ellos por medio de una orden de trabajo (documento 2, ver figura A.2). Con esa orden, el operario se dirige al lugar donde se ha producido la avería para evaluar los daños y comprobar el material necesario para resolverla.

Si el operario no dispone del material necesario debe comunicárselo al maestro para que realice una petición de material (documento 3, ver figura) al servicio de suministros del hospital. En este caso la avería queda pendiente de resolución hasta que se reciba el material. Si se dispone del material necesario, el operario procede a la resolución de la avería. Una vez finalizada la reparación y firmado el V°B°, el operario entrega la orden de trabajo al maestro para que la registre.

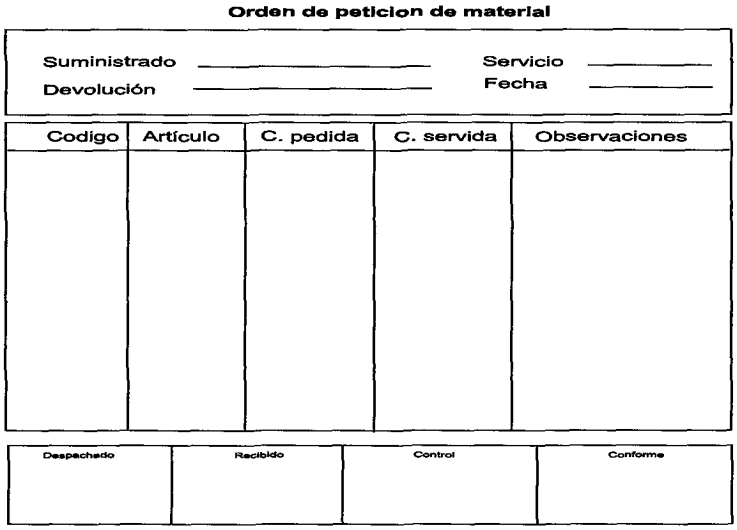


Figura A.2. *Petición de material.*

Escenario de los contratos de mantenimiento de las máquinas. A veces ocurre que se producen averías sobre maquinaria de naturaleza muy específica (ascensores, rayos X, T.A.C., R.M., ...), que el servicio de mantenimiento no puede resolver. En estos casos, si la máquina afectada tiene un contrato de mantenimiento con una empresa externa, la reparación la realiza dicha empresa.

Las máquinas que tienen contratado un servicio de mantenimiento externo, deben de ser revisadas periódicamente para comprobar el buen funcionamiento de las mismas. El maestro debe de controlar que estas revisiones periódicas se realizan y en caso de demora,

ponerse en contacto con la empresa externa para que cumplan con su contrato de mantenimiento.

Escenario de absentismo laboral. El propio servicio de mantenimiento debe de gestionar información referente a sus empleados para llevar un control del las bajas por ILT (incapacidad laboral transitoria). En el caso de que se produzca una ILT de algún empleado del servicio, el responsable debe de rellenar una solicitud de contratación por I.L.T. (documento 4, ver Figura A.3) para el servicio de personal del hospital quien asignará al servicio un nuevo operario a partir de la bolsa de trabajo.

Modelo de Solicitud de contratacion de sustituciones por I.L.T.

Servicio _____
Personal en I.L.T. _____
Tiempo estimado de baja _____
Diagnóstico médico _____
Contratos por I.L.T. que tiene en la actualidad el servicio _____
Motivos por los que se solicita la contratacion _____
Fecha contratación _____
Turno _____

VºBº El administrador

El responsable del servicio

Figura A.3. *Solicitud de I.L.T.*

A.2 Elaboración del Modelo Conceptual

Una forma clara y adecuada de estructurar la descripción general del Sistema de Información objeto de estudio, es la de presentar una relación precisa de requisitos que guíen el proceso de modelado conceptual.

A.2.1 Requisitos del Sistema

De la descripción anterior del sistema de información asumimos que se extraen los siguientes Requisitos (ver Tabla A.1), que debe de cumplir nuestro sistema.

1. El personal del servicio de mantenimiento se encuentra clasificado por categorías. Existe un responsable del servicio (jefe de mantenimiento), un maestro por cada una de las áreas en la que se encuentra dividido el servicio y operarios asignados a ellas.
2. La gestión del absentismo laboral requiere:
   - 2.1 Llevar el control del absentismo laboral de los empleados del servicio de mantenimiento.
   - 2.2 En cualquier momento se puede actualizar la ficha de absentismo de un empleado y/o consultar durante el año en curso los días que ha faltado a trabajar y el motivo.
3. Las máquinas del hospital pueden estar sujetas a contratos de mantenimiento que vinculan las máquinas con empresas externas encargadas por un lado de ofrecer asistencia ante posibles averías y por otro a realizar ciertas revisiones anuales para controlar el buen funcionamiento de éstas.
4. Se desea conocer, cual es la fecha de la última revisión realizada a una máquina que tiene un contrato de mantenimiento.
5. Cuando a un operario se le asigna la resolución de una incidencia, se dirige al lugar donde se ha producido y toma nota en el parte de incidencia del material que va a necesitar.
6. Cuando no se dispone del material necesario para resolver una avería, se debe de realizar un pedido al servicio de suministros del hospital para que compre las piezas necesarias.
7. Cuando se produce una incidencia en el hospital:
   - 7.1 El responsable del servicio afectado rellena el parte de incidencias.
   - 7.2 Diariamente el responsable, clasifica las incidencias asignándoles la prioridad y el área que la atenderá.
   - 7.3 Los partes clasificados se reparten entre los maestros, quienes asignarán los operarios que atenderán la reparación.
   - 7.4 En caso de que la incidencia afecte a una máquina con contrato de mantenimiento se avisará a la empresa externa contratada para que atienda la avería.

Tabla A.1. Requisitos del Sistema Servicio de Mantenimiento

La fase de modelado conceptual se inicia con la construcción de tres modelos: el modelo de objetos, modelo dinámico y modelo funcional. El propósito de esta fase es construir una descripción más precisa a partir de la especificación de requisitos anterior.

A.2.2 Modelo de Objetos

Con este diagrama se identifican y especifican las clases del sistema y las relaciones entre ellas. Cada clase, desde esta perspectiva, tendrá básicamente un conjunto de atributos, servicios y restricciones de integridad, y será agente de un conjunto de servicios ofertados por otras clases del sistema.

Veamos como soportamos cada uno de los requisitos anteriores en este modelo. Diferenciaremos entre clases y relaciones.

> **Requisito 1**
>
> El **personal** del servicio de mantenimiento del hospital se encuentra clasificado por categorías. Existe un **jefe de mantenimiento**, un **maestro** por cada una de las **áreas** en las que se encuentra dividido el servicio y **operarios** asignados a ellas.

Clases. Las palabras resaltadas en negrita se corresponden con clases involucradas en la satisfacción del requisito. Son las siguientes: personal, jefe de mantenimiento, maestro, operario y área. A continuación se presenta la especificación de cada una de ellas desglosando su conjunto de atributos (el atributo subrayado representa el atributo identificador de la clase), y sus servicios.

Personal: empleados del servicio de mantenimiento, están clasificados en tres categorías; responsable o jefe del servicio, maestro y operario.

Jefe de Mantenimiento: es el responsable del servicio de mantenimiento. Un jefe lo es para un periodo de mandato determinado.

Operario: tipo de personal del servicio que atiende la resolución de las averías. Se les asigna un turno de trabajo.

Area: el servicio de mantenimiento se organiza en áreas especializadas en resolver averías concretas (electricidad, fontanería, jardinería, mecánica, ...).

Nombre	Tipo	Dom	Tam	Descripción
dni	cte	string	10	dni
apellidos	cte	string	20	apellidos
nombre	cte	string	20	nombre
fnac	cte	date		f. nacimiento
categoría	cte	string	20	puesto
población	cte	string	20	población
provincia	cte	string	20	provincia
cpostal	cte	string	10	c. postal
teléfono	cte	string	10	teléfono
estado	vble	string	10	trabaja, notrabaja

Tabla A.2. Atributos clase personal

Nombre	Tipo	Argumentos	Descripción
crearPersona	new		creación
destruirPersona	destroy		destrucción
insertarAbsentismo	compartido		inserción en agregación
finalizarAbsentismo	compartido	fechaFin (date)	asignar fecha de fin de absentismo

Tabla A.3. Servicios clase personal

Nombre	Tipo	Dom	Tam	Descripción
finicio	vble	date		f. inicio
ffinal	vble	date		f. final

Tabla A.4. Atributos clase jefe de mantenimiento

Nombre	Tipo	Argumentos	Descripción
asignarPeriodo	propio	fechaInicio (date), fechaFin (date)	asignar periodo de mandato

Tabla A.5. Servicios clase jefe de mantenimiento

Nombre	Tipo	Dom	Tam	Descripción
turno	vble	string	10	turno trabajo

Tabla A.6. Atributos clase operario

Nombre	Tipo	Argumentos	Descripción
asignarTurno	propio	nuevoTurno (string)	asignar turno

Tabla A.7. Servicios clase operario

Nombre	Tipo	Dom	Tam	Descripción
codArea	cte	string	10	código
descripción	cte	string	20	descripción

Tabla A.8. Atributos clase área

Nombre	Tipo	Argumentos	Descripción
crearArea	new		creación
destruirArea	destroy		destrucción

Tabla A.9. Servicios clase área

Relaciones. Una persona se especializa en alguno de sus tipos (jefe de mantenimiento, maestro u operario).

Con esta información se puede dibujar un modelo de objetos inicial como el de la Figura A.4. En concreto, se utiliza el constructor de modelado correspondiente para representar jerarquías de especialización, una flecha que apunta desde las clases especializadas hacia la clase padre.

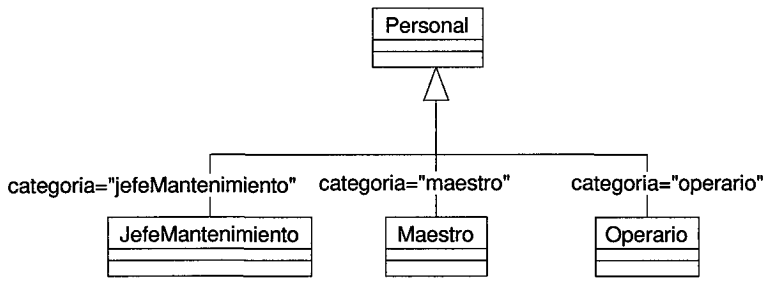


Figura A.4. *Relación de Especialización.*

Cuando se crea una persona (mediante invocación del servicio crearPersona), ésta se especializa según el valor del atributo categoría que toma un valor de entre tres posibles (jefe de manteni-

miento, maestro u operario). Puesto que categoría es un atributo constante la especilización es permanente y tiene vigencia durante toda la vida del objeto creado.

Requisito 2

La gestión del absentismo laboral requiere:

2.1 Llevar el control del **absentismo laboral** de los empleados del servicio de mantenimiento.

2.2 En cualquier momento se puede actualizar la ficha de absentismo de un empleado introduciendo el **tipo de baja** correspondiente, también se puede consultar durante el año en curso los días que ha faltado a trabajar y porqué.

Clases. En este caso, las nuevas clases que se ven involucradas son: absentismoLaboral y tipoBaja. Veamos las especificaciones correspondientes para cada una de ellas.

Absentismo Laboral: calendario anual que sirve para controlar las situaciones de bajas temporales del personal del servicio de mantenimiento.

Nombre	Tipo	Dom	Tam	Descripción
fechaInicio	cte	date		f. inicio
fechaFinal	vble	date		f. final

Tabla A.10. Atributos clase absentismoLaboral

Nombre	Tipo	Argumentos	Descripción
crearAbsentismo	new		creación
destruirAbsentismo	destroy		destrucción
insertarAbsentismo	compartido		inserción en agregación
finalizarAbsentismo	compartido	fechaFin (date)	f. finalización absentismo

Tabla A.11. Servicios clase absentismoLaboral

Tipo baja: representa los distintos tipos de baja codificados que puede sufrir el personal de servicio de mantenimiento (enfermedad común, operación quirúrgica, baja maternal, permiso sin sueldo, libre disposición, ...).

Nombre	Tipo	Dom	Tam	Descripción
codBaja	cte	nat		código
descripcion	cte	string	30	tipo de baja

Tabla A.12. Atributos clase tipo_baja

Nombre	Tipo	Argumentos	Descripción
crearTipoBaja	new		creación
destruirTipoBaja	destroy		destrucción

Tabla A.13. Servicios clase tipo_baja

Relaciones. Una persona puede tener relacionado un conjunto de situaciones de absentismo laboral. Cada situación de absentismo tiene relacionado un tipo de baja determinado.

Los empleados del servicio de mantenimiento pueden faltar al trabajo por varios motivos como por ejemplo, por causa de una enfermedad común, por vacaciones, por maternidad, por día de libre disposición, por intervención quirúrgica u otras muchas causas que están reguladas por ley. La 'no asistencia' al puesto de trabajo por una causa justificada es lo que se conoce como absentismo laboral.

Una situación de absentismo laboral se determina fijando la fecha de inicio y la fecha de fin del periodo afectado, por tanto, cada empleado del servicio de mantenimiento tendrá una relación de agregación dinámica inclusiva con dependencia de identificación a través de su DNI con la clase AbsentismoLaboral (ver Figura A.5). La agregación se define inclusiva porque una instancia de absentismo laboral sólo tiene sentido en el contexto de un empleado. La definimos dinámica porque a un empleado se le pueden asignar varias incidencias de absentismo a lo largo de su vida. Por último,

caracterizamos esta agregación con dependencia de identificación puesto que una instancia de absentismo laboral no puede identificarse por sí sola ya que siempre hará referencia al absentismo laboral de un empleado determinado. Continuando con la descripción de la Figura A.5, la cardinalidad de la agregación indica que un empleado puede tener o ninguna o varias incidencias de absentismo laboral pero que cada incidencia de absentismo pertenece a un empleado.

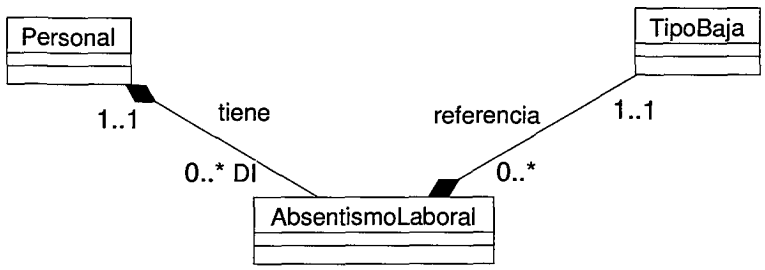


Figura A.5. *Relación de absentismo laboral.*

Examinemos ahora la relación de agregación entre AbsentismoLaboral y TipoBaja. En este caso, la agregación es de tipo referencial. Decir que es referencial implica que pueden existir instancias de la clase componente (tipoBaja) en el contexto externo a la agregación. Esta agregación proporciona un mecanismo para codificar las bajas en el absentismo laboral de un empleado. Además es estática, en el sentido de que en el momento de creación de un absentismo laboral se debe de especificar el tipo de baja. Una vez creado el absentismo no se puede modificar.

Requisito 3

Las **máquinas** del hospital pueden estar sujetas a **contratos de mantenimiento** que vinculan las máquinas con **empresas externas** encargadas por un lado de ofrecer asistencia ante posibles averías y por otro a realizar ciertas **revisiones** anuales para controlar el buen funcionamiento de éstas.

Clases. Este requisito involucra varias clases interesantes del sistema de información que se está modelando, éstas son: máquina, contrato de mantenimiento, empresa externa y revisión. Se presenta la especificación de cada una:

Máquina: aparatos distribuidos por los distintos servicios del hospital que se utilizan para realizar pruebas a los pacientes.

Nombre	Tipo	Dom	Tam	Descripción
codReferencia	cte	nat		referencia de la máquina
ubicación	cte	string	20	lugar donde reside la máquina
marca	cte	string	20	marca de la máquina
modelo	cte	string	20	modelo de la máquina
descripcion	cte	string	30	características
conContrato	vble	Bool		si tiene o no mantenimiento
disponible	vble	Bool		si la máquina está disponible o estropeada
ultRevision	vble	Date		f. última revisión

Tabla A.14. Atributos clase máquina

Nombre	Tipo	Argumentos	Descripción
crearMaquina	new		creación
destruirMaquina	destroy		destrucción
reparar	propio		máquina disponible
averiar	propio		máquina no disponible
registrarRevision	compartido	codRevision (nat)	registrar revisión
insertarContrato	compartido	codRevision (nat)	inserción en agregación

Tabla A.15. Servicios clase máquina

Contrato de Mantenimiento: documento que regula un acuerdo comercial por el cual una empresa externa se compromete a realizar el mantenimiento de máquinas del hospital. El propio contrato obliga a realizar revisiones periódicas sobre las máquinas.

Nombre	Tipo	Dom	Tam	Descripción
codContrato	cte	nat		código
fContrato	cte	date		fecha
descripcion	cte	string	20	descripción

Tabla A.16. Atributos clase contrato_mantenimiento

Nombre	Tipo	Argumentos	Descripción
crearContrato	new		creación
destruirContrato	destroy		destrucción
insertarContrato	compartido		inserción en agregación
insertarRevision	compartido		inserción en agregación

Tabla A.17. Servicios clase contrato_mantenimiento

Empresa Externa: empresas que venden máquinas u ofrecen mantenimiento sobre alguna de las que ya tenemos.

Nombre	Tipo	Dom	Tam	Descripción
cifEmpresa	cte	string	10	CIF
nombre	cte	string	20	nombre
persContacto	cte	string	20	contacto
direccion	cte	string	20	dirección
cPostal	cte	string	10	c. postal
poblacion	cte	string	20	población
provincia	cte	string	20	provincia
pais	cte	string	20	país
telefono	cte	string	10	teléfono
fax	cte	string	10	fax

Tabla A.18. Atributos clase empresa_externa

Revisión: información de las revisiones que pasan las máquinas sujetas a contratos de mantenimiento.

Nombre	Tipo	Argumentos	Descripción
crearEmpresa	new		creación
destruirEmpresa	destroy		destrucción

Tabla A.19. Servicios clase empresa_externa

Nombre	Tipo	Dom	Tam	Descripción
codRevision	cte	nat		código
fRevision	cte	date		fecha revisión
descripcion	cte	string	20	descripción
estado	vble	string	20	pendiente o realizada

Tabla A.20. Atributos clase revision

Nombre	Tipo	Argumentos	Descripción
crearRevision	new		creación
destruirRevision	destroy		destrucción
registrarRevision	compartido		registra una revisión realizada
insertarRevision	compartido		inserción en agregación

Tabla A.21. Servicios clase revision

Relaciones. Una máquina puede tener relacionado un contrato de mantenimiento. Un contrato de mantenimiento se contrata con una empresa externa y obliga a realizar 1 o más revisiones durante el periodo de vigencia del contrato.

La Figura A.6 representa la parte del modelo de objetos que satisface el Requisito 3. La clase máquina se define como una agregación inclusiva y dinámica que tiene como componente la clase ContratoMantenimiento. Se define inclusiva porque el contrato de mantenimiento solo puede existir si está asociado a una máquina. También es dinámica, porque cuando se crea la máquina no es necesario asociarle un contrato de mantenimiento. A su vez contratoMantenimiento tiene una relación de agregación estática y referencial con EmpresaExterna. En este caso se define como referencial porque las empresas pueden existir fuera del contexto de los contratos de mantenimiento. Además es estática porque

en el momento de creación de un contrato debemos de asociar la empresa mantenedora.

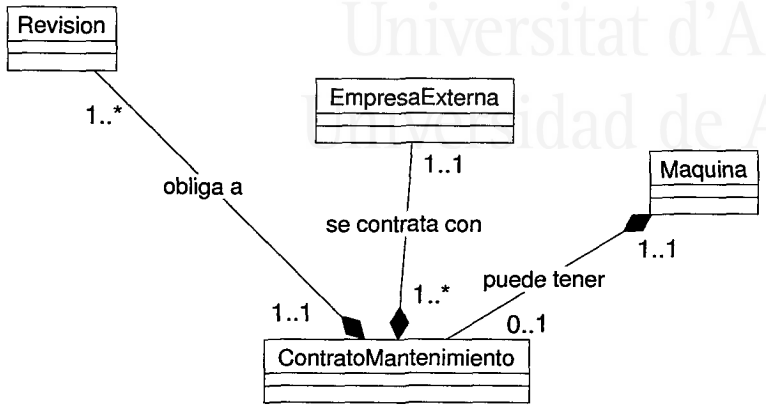


Figura A.6. *Contratos de mantenimiento de las máquinas.*

Por último nos queda la relación de agregación de ContratoMantenimiento con Revisión, que se define como inclusiva y dinámica. Por un lado, es inclusiva porque las revisiones siempre van asociadas a los contratos de mantenimiento y no tiene sentido fuera de ese contexto. Por otro, es dinámica, porque se pueden ir asignando nuevas revisiones a un contrato de mantenimiento a lo largo de su vida. Las cardinalidades de las agregaciones reflejan las siguientes situaciones. Un contratoMantenimiento ofrece mantenimiento sobre una y sólo una máquina, una máquina puede o no tener asociado un contrato de mantenimiento. Un contratoMantenimiento se contrata con una y solo una empresa externa. Una empresa externa puede soportar uno o varios contratos de mantenimiento. Un contrato de mantenimiento obliga a realizar como mínimo una revisión o como máximo m, una revisión está asociada a un solo contrato de mantenimiento.

Requisito 4

Se desea conocer, cual es la fecha de la última revisión realizada a una **máquina** que tiene un **contrato de mantenimiento**.

Clases. En el requisito 4 no se detecta ninguna clase nueva. Pasaremos a estudiar las relaciones involucradas en este requisito.

Relaciones. Como se ha visto antes una máquina puede tener una relación con un contrato de mantenimiento que obliga a pasar ciertas revisiones (1 o más).

Figura A.7. *Revisiones de máquinas.*

La Figura A.7, ilustra la relación revisiones de máquinas en el modelo de objetos. Para ello definimos un servicio compartido entre las clases máquina y revisión. A ese servicio lo denominamos registrarRevision. Intuitivamente, registrarRevision sería un servicio propio de la clase revision. Sin embargo, es necesario definir este servicio como compartido para poder registrar revisiones desde la clase máquina y poder especificar el efecto de la ejecución de ese servicio sobre el atributo que deberá de contener la información de la última revisión realizada. Por consiguiente, el efecto de la ejecución de este servicio sobre el atributo ultRevision de la clase máquina, provocará el cambio en su valor que se fijará con la fecha de la última revisión realizada[1]. Por lo tanto, este requisito estaba resuelto con el modelo resultante del requisito anterior.

Requisito 5

Cuando un operario resuelve una **incidencia**, toma nota en el parte de incidencia de las **piezas** y la cantidad de las mismas que ha utilizado.

Clases. Incidencia, Línea de Incidencia y Piezas. La especificación correspondiente de cada una de ellas se presenta a continuación.

[1] se puede examinar con más detalle este efecto, en la sección del modelo funcional, más adelante

Incidencia: situación o aviso que puede producirse en los servicios del hospital y que normalmente tiene que ver con el funcionamiento de máquinas o instalaciones.

Nombre	Tipo	Dom	Tam	Descripción
codIncidencia	cte	nat		código
fPeticion	cte	date		fecha registro
descripcion	cte	string	10	descripción
prioridad	vble	nat		prioridad
fSolucion	vble	date		f. resolución
fRechazo	vble	date		f. rechazo

Tabla A.22. Atributos clase incidencia

Nombre	Tipo	Argumentos	Descripción
crearIncidencia	new		creación
destruirIncidencia	destroy		destrucción
insertarLineaIncidencia	compartido		inserción en agregación
asignarPrioridad	propio	numPrioridad (nat)	asignar prioridad
insertarOperario	compartido		inserción en agregación
borrarOperario	compartido		borrado en agregación
solucionar	propio		solucionada
rechazar	propio		rechazada
insertarArea	compartido		inserción en agregación
borrarArea	compartido		borrado en agregación
insertarPedido	compartido		inserción en agregación
borrarPedido	compartido		borrado en agregación

Tabla A.23. Servicios clase incidencia

lineaIncidencia: contiene información referente a la incidencia. En concreto las piezas que se han utilizado para su resolución.

Pieza: material utilizado para la resolución de incidencias Por cada pieza interesa conocer su precio unitario, descripción, unidad de medida, cantidad (stock actual), stock mínimo y stock máximo.

Nombre	Arg	Fórmula	Descripción
CLASIFICAR	num_prioridad (nat)	InsAreaIncide(). asig_prioridad (num_prioridad)	Clasificar una incidencia consiste en asignarle un área y establecer su prioridad de atención

Tabla A.24. Transacciones clase incidencia

Nombre	Tipo	Dom	Tam	Descripción
codLinea	cte	nat		número linea
cantidad	cte	nat		cantidad material

Tabla A.25. Atributos clase lineaIncidencia

Nombre	Tipo	Argumentos	Descripción
crearLineaIncidencia	new		creación
destruirLineaIncidencia	destroy		destrucción

Tabla A.26. Servicios clase lineaIncidencia

Nombre	Tipo	Dom	Tam	Descripción
codPieza	cte	nat		código
descripcion	cte	string	30	descripción
unidadMedida	cte	string	10	medida
stockMinimo	vble	nat		stock mínimo
stockMaximo	vble	nat		stock máximo
pUnit	vble	nat		precio unitario
cantidad	vble	nat		stock actual

Tabla A.27. Atributos clase piezas

Relaciones. Una incidencia se relaciona con 1 o más líneas de incidencia, y cada línea de incidencia se relaciona con una pieza.

La Figura A.8 representa la parte de nuestra especificación que satisface este Requisito. La relación de agregación dinámica, inclusiva y con dependencia de identificación entre incidencia y lineaIncidencia representa una relación parte_de que permite asociar a una incidencia el material que se ha utilizado para atenderla. Además

Nombre	Tipo	Argumentos	Descripción
crearPieza	new		creación
destruirPieza	destroy		destrucción
modificarPrecioUni	propio	nuevoPrecio (nat)	modificar precio unitario
modificarStockMin	propio	nuevoStockMin (nat)	modificar stock_mínimo
modificarStockMax	propio	nuevoStockMax (nat)	modificar stock_máximo
incrementarStock	propio	cantIncr (nat)	comprar piezas
decrementarStock	propio	cantDecr (nat)	usar piezas

Tabla A.28. Servicios clase piezas

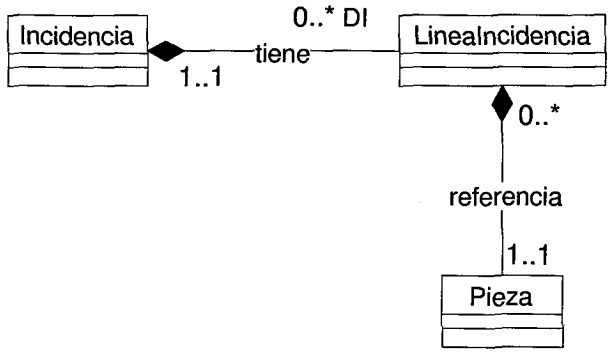


Figura A.8. *Resolución de incidencia.*

se establece una relación de agregación estática y referencial entre lineaIncidencia y pieza para indicar información descriptiva de la pieza utilizada.

Requisito 6

Cuando no se dispone de las **piezas** necesarias para resolver una avería o **incidencia**, se debe de realizar un **pedido** al servicio de suministros del hospital para que compre las piezas necesarias.

Clases. Las nuevas clases que aparecen en este requisito son pedido y línea de pedido. Las especificaciones correspondientes se presentan a continuación:

Pedido: documento que se utiliza para la petición de material.

Nombre	Tipo	Dom	Tam	Descripción
codPedido	cte	nat		código
fPedido	cte	date		fecha del pedido
fRecepcion	vble	date		fecha de recepción
estado	vble	string	10	pendiente, recibido

Tabla A.29. Atributos clase pedido

Nombre	Tipo	Argumentos	Descripción
crearPedido	new		creación
destruirPedido	destroy		destrucción
insertarLineaPedido	compartido		inserción en agregación
recibirPedido	propio	fecha (date)	pedido recibido
insertarPedido	compartido		inserción en agregación
borrarPedido	compartido		borrado en agregación

Tabla A.30. Servicios clase pedido

lineaPedido: información detallada de las piezas del pedido.

Nombre	Tipo	Dom	Tam	Descripción
numLinea	cte	nat		linea del pedido
cantidad	cte	nat		cantidad

Tabla A.31. Atributos clase lineaPedido

Relaciones. Una incidencia se puede relacionar con un pedido. El pedido se relaciona con una o más líneas de pedido. Una línea de pedido se relaciona con una pieza.

La Figura A.9 ofrece soporte para la satisfacción de este Requisito. Cada incidencia tiene una relación de agregación dinámica

Nombre	Tipo	Argumentos	Descripción
insertarLineaPedido	new		creación
borrarLineaPedido	destroy		destrucción

Tabla A.32. Servicios clase lineaPedido

referencial con pedido. Cada pedido tiene una relación dinámica inclusiva con dependencia de identificación con lineaPedido. Por último la agregación estática referencial de lineaPedido con pieza permite terminar de rellenar un pedido con las descripciones de las piezas que se han de comprar.

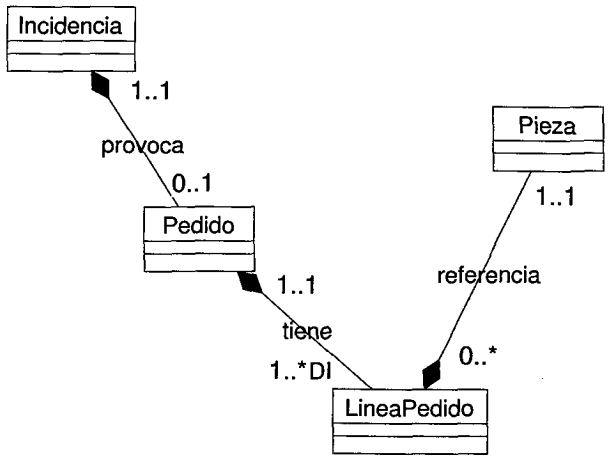


Figura A.9. *Hacer pedidos.*

Requisito 7

Cuando se produce una incidencia en el hospital:

7.1 El responsable del servicio afectado rellena el parte de **incidencias**.

7.2 Diariamente el **jefe de mantenimiento**, clasifica las incidencias asignándoles la prioridad y el **área** que la atenderá.

7.3 Los partes clasificados se reparten entre los **maestros**, quienes asignarán los **operarios** que atenderán la reparación.

7.4 En caso de que la incidencia afecte a una **máquina** con **contrato de mantenimiento** se avisará a la **empresa externa** contratada para que atienda la avería.

Clases. Este requisito involucra una nueva clase que es la clase servicio. Su especificación es la siguiente:

Servicio: los usuarios del sistema son los empleados del hospital, estos están adscritos funcionalmente al servicio donde trabajan (personal, suministros, contabilidad, ...).

Nombre	Tipo	Dom	Tam	Descripción
codServicio	cte	string	10	código
descripción	cte	string	20	descripción

Tabla A.33. Atributos clase servicio

Nombre	Tipo	Argumentos	Descripción
crearServicio	new		creación
destruirServicio	destroy		destrucción

Tabla A.34. Servicios clase servicio

Relaciones. Una incidencia se relaciona con un usuario que es quien informa de ella. El usuario se relaciona con un servicio al que

pertenece. La incidencia se relaciona con el área que la clasifica y también se relaciona con el operario que se le asigna para su resolución. Finalmente una incidencia también se puede relacionar con una máquina que se ve afectada.

La Figura A.10 representa la satisfacción de estos últimos requisitos. La clase incidencia es una clase compuesta con las siguientes características. Tiene una relación de agregación estática referencial con usuario para satisfacer el req. 7.1, a su vez usuario tiene una agregación estática referencial con servicio para indicar el servicio al que pertenece un usuario. Tiene una relación de agregación dinámica referencial con área para satisfacer el req. 7.2. Tiene una relación de agregación dinámica referencial con operario para satisfacer el req. 7.3, y finalmente tiene una relación de agregación dinámica referencial con máquina para satisfacer el req. 7.4.

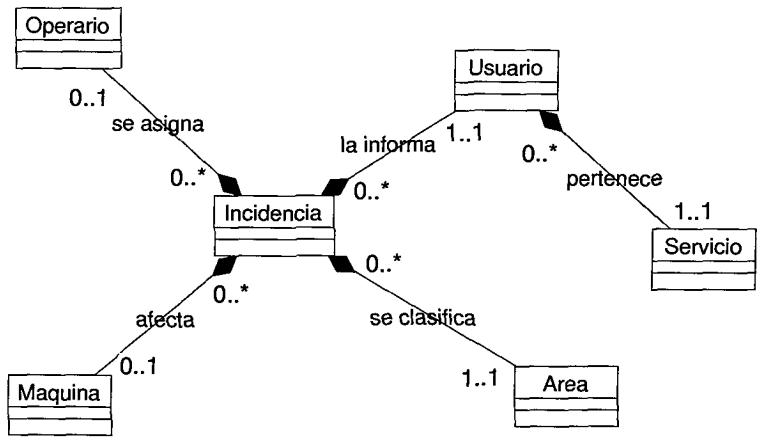


Figura A.10. *Gestión de incidencias.*

Relaciones de Agente. En el modelo de objetos también se representan las relaciones de agente que indican qué clases pueden activar qué servicios de otras clases. La tabla A.35 muestra todas las relaciones de agente que aparecen en la especificación del sistema servicio de mantenimiento.

```
-----------------------------------------------------------------
Clase Agente: Jefe de Mantenimiento
-----------------------------------------------------------------
Abs_Laboral (crearAbsentismo, destruirAbsentismo,
                insertarAbsentismo, finalizarAbsentismo)
Area(crearArea, destruirArea)
ContratoMantenimiento(crearContrato, destruirContrato,
                        insertarContrato, insertarRevisión)
Empresa_Externa(crearEmpresa, destruirEmpresa)
Incidencia(insertarLineaIncidencia, insertarArea, borrarArea,
            insertarPedido, borrarPedido, CLASIFICAR)
Maquina(crearMaquina, destruirMaquina, insertarContrato)
Operario(asignarTurno)
Personal(crearPersona, destruirPersona, insertarAbsentismo,
            finalizarAbsentismo)
jefeMantenimiento(asignarPeriodo) Revision(insertarRevision,
destruirRevision) Servicio(crearServicio, destruirServicio)
TipoBaja(crearTipoBaja, destruirTipoBaja)
-----------------------------------------------------------------
Clase Agente: Maestro
-----------------------------------------------------------------
Incidencia(insertarOperario, borrarOperario, insertarPedido,
            borrarPedido)
Maquina(reparar, averiar, registrarRevision)
Pedido(crearPedido, destruirPedido, insertarLineaPedido,
        recibirPedido, insertarPedido, borrarPedido)
Pieza(crearPieza, destruirPieza, modificarPrecioUni,
        modificarStockMin, modificarStockMax, incrementarStock,
        decrementarStock)
-----------------------------------------------------------------
Clase Agente: Operario
-----------------------------------------------------------------
Incidencia(solucionar,rechazar, insertarLineaIncidencia)
-----------------------------------------------------------------
Clase Agente: Usuario
-----------------------------------------------------------------
Incidencia(crearIncidencia)
-----------------------------------------------------------------
```

Tabla A.35. Relaciones de agente del sistema

Como ejemplo podemos ver cómo la clase jefeMantenimiento se define como agente (entre otros), de los servicios crearTipoBaja y destruirTipoBaja de la clase TipoBaja. En el modelo de objetos también se especifican los servicios compartidos. En nuestro sistema la mayoría de los servicios compartidos que aparecen se definen implícitamente, y se corresponden con los servicios de inserción y

borrado de las agregaciones dinámicas especificadas anteriormente.

A.2.3 Modelo Dinámico

Abordamos el siguiente paso de modelado conceptual especificando el modelo dinámico. Este modelo nos permite especificar los posibles ciclos de vida válidos de los objetos de nuestro sistema y la comunicación interobjetual. Para ello se usan dos tipos de diagramas: el diagrama de transición de estados (uno por cada clase), y el diagrama de interacción (genérico para todo el sistema).

Diagrama de transición de estados. Empezaremos presentando los más sencillos para ir progresivamente mostrando los más complejos, comentando todas sus características y explicando su correspondencia con la descripción del sistema.

Clase área: La Figura A.11 muestra el diagrama de transición de estados (DTE) para la clase del dominio área. En este caso es un DTE básico caracterizado por los estados de no existencia, creación y destrucción. Cuando se crea un objeto de tipo área a través de la ejecución del servicio crearArea se pasa al estado area0. Desde este estado se puede destruir el objeto ejecutando el servicio de destrucción destruirArea. El * asociado a la transición denota que cualquier agente válido puede activar el servicio.

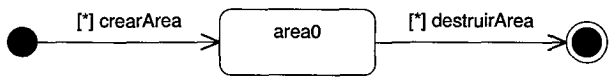


Figura A.11. *DTE de la clase área.*

Clase personal: Estudiemos el DTE de una clase un tanto más compleja. La Figura A.12, muestra el DTE para la clase personal. Del estado de no existencia pasamos al estado persona0 cuando acontece el servicio crearPersona. Desde este estado pueden ocurrir varias cosas; si acontece el servicio insertarAbsentismo, se pasa al estado persona1. Este estado caracteriza a una persona que tiene

abierta un absentismo laboral. Desde este último estado, volveremos al estado person0 si acontece el servicio finalizarAbsentismo. Desde el estado persona0 se puede destruir un objeto de tipo persona a través del servicio destruirPersona.

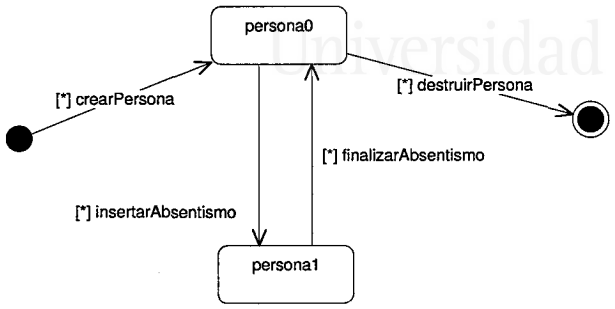


Figura A.12. *DTE de la clase personal.*

A continuación veamos el DTE asociado a la clase Incidencia que es el DTE más complejo de nuestro sistema por que los objetos que pertenecen a esta clase pueden pasar por cinco estados diferentes a lo largo de su ciclo de vida.

Clase incidencia: La Figura A.13 muestra el DTE de la clase incidencia. La creación de una incidencia (ejecución del servicio crearIncidencia), lleva al estado de creación incidencia0. En este estado la incidencia está pendiente de ser clasificada. Puede ejecutarse la transacción local CLASIFICAR, pasando al estado incidencia1. Desde incidencia1, puede acontecer insertarOperario, que consiste en asignar un operario a la incidencia, pasando al estado incidencia2. En este estado pueden sucederse varios servicios; se puede insertar un pedido (insertarPedido) o se pueden insertar líneas de incidencia con el material utilizado en la reparación (insertarLinIncidencia). Desde el estado incidencia2 se pasa al estado incidencia3 cuando se activa el servicio rechazar, en el caso de que la incidencia no tenga solución, o se active el servicio solucionar en el caso de haber podido ser reparada. Desde el estado incidencia3 se puede activar el servicio destruirIncidencia para destruir una incidencia.

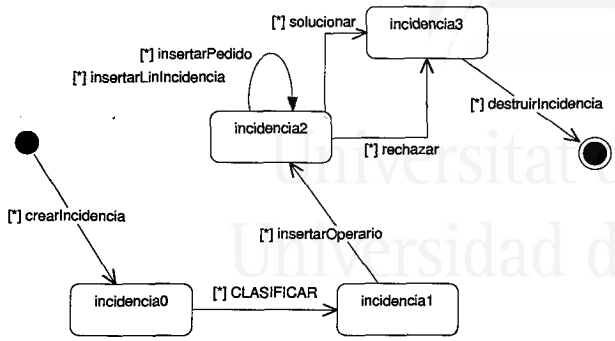


Figura A.13. *DTE de la clase incidencia.*

Clase pedido: La clase pedido (ver Figura A.14), únicamente tiene dos estados de vida diferentes, un estado inicial al crearse (pedido0), en el que se van insertando las líneas de pedido con las piezas que vamos a solicitar (insertarLinPedido). Cambiaremos de estado cuando recibamos el pedido y lo registremos. Esto se realiza con el servicio recibirPedido que nos llevará al estado pedido1. Una vez en este estado podemos borrar el pedido del sistema con destruirPedido.

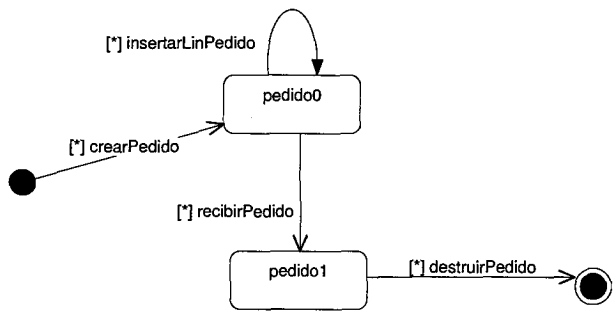


Figura A.14. *DTE de la clase pedido.*

Clase Máquina: La Figura A.15 ilustra que una máquina puede encontrarse en dos estados posibles, en funcionamiento (estado maquina0) o averiada (estado maquina1). Desde el estado de creación (maquina0), se puede: asignar un contrato a la maquina (insertarContrato), registrar la fecha de la última revisión realizada (registrarRevision), ejecutar el servicio averiar, que produce una

transición al estado maquina1 (estado de avería). Desde maquina1, el servicio reparar nos devuelve al estado maquin0. Sólo desde maquin0 se puede activar el servicio de destrucción destruirMaquina.

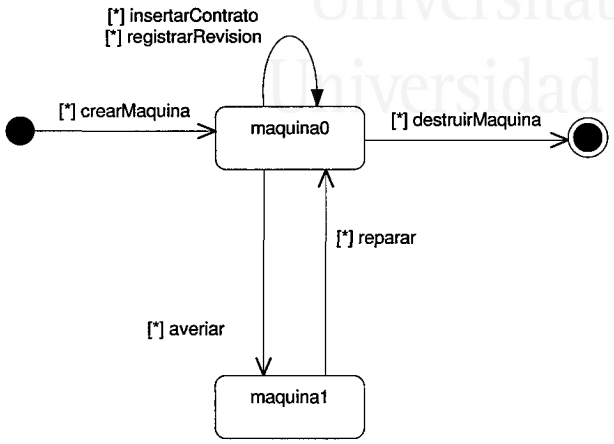


Figura A.15. *DTE de la clase máquina.*

Clase revisión: Un objeto revisión pasa por tres etapas a lo largo de su vida, inicialmente se introduce como parte del contrato de una maquina (estado revision0). En este estado la revisión está pendiente de realizarse. Cuando se realiza (activación del servicio registrarRevision), se pasa al estado revision1. Este estado representa una revisión ya realizada. Finalmente se puede borrar una revisión a través del servicio destruirRevision. La Figura A.16 muestra esta situación.

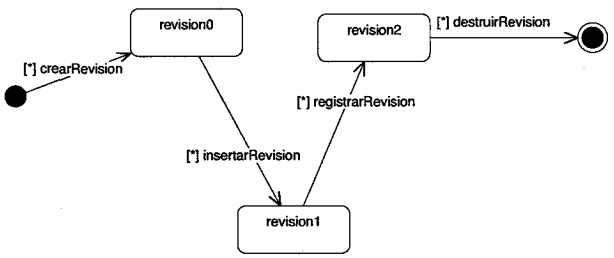


Figura A.16. *DTE de la clase revisión.*

Las clases restantes, cuyos DTE's no se introducen (Empresa Externa, Servicio, Tipo Baja,...) siguen todas un esquema parecido al de la clase area presentada en la Figura A.11.

Diagrama de Interacción de Objetos. Este último diagrama perteneciente al modelo dinámico, nos permite especificar la comunicación interobjetual mediante la especificación de condiciones de disparo e interacciones globales.

En nuestro sistema encontramos algunos ejemplos de comunicación interobjetual. En concreto, la especificación de un disparo y la especificación de dos interacciones globales. Examinémoslas con más detalle.

La figura A.17 representa un disparo de nuestro sistema.

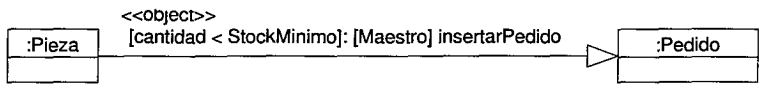


Figura A.17. *Disparo de stock mínimo.*

Este disparo parte de la clase pieza, donde si se cumple la condición de que la cantidad de la pieza sea menor que el stockMinimo, entonces se activa el servicio insertarPedido de la clase pedido. De esta manera se repone la cantidad de piezas necesarias para el mantenimiento. Como vemos es un disparo tipo object, ya que afecta a un objeto concreto de la clase pedido.

En lo que se refiere a interacciones globales existen dos en nuestro sistema.

La Figura A.18 muestra la interacción global etiquetada como recibirPiezas. El sentido que tiene esta interacción es el de reponer las piezas cuando se ha recibido un pedido. De manera que el primer evento es el de recibirPedido de la clase pedido, una vez hecho esto se ejecuta a continuación incrementarStock de la clase pieza.

De forma análoga, pero al contrario, la interacción global de la figura A.19 usarPiezas, refleja que cuando se ha solucionado alguna

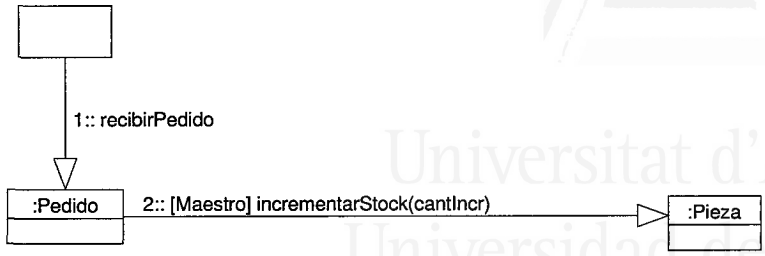


Figura A.18. *Interacción global recibirPiezas.*

incidencia utilizando ciertas piezas del almacén, la cantidad de stock resultante para esa pieza se debe de decrementar en función de las piezas usadas.

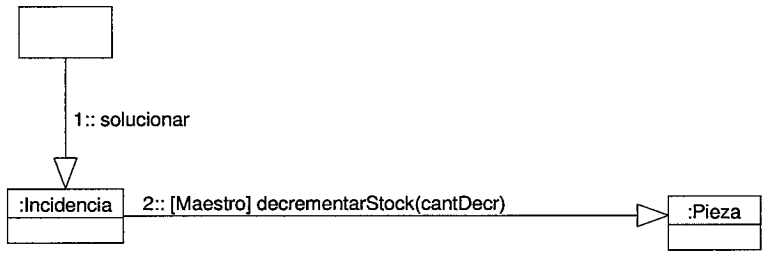


Figura A.19. *Interacción global usarPiezas.*

A.2.4 Modelo Funcional

Pasamos a construir el modelo funcional en el cual especificaremos el efecto que tienen sobre los atributos variables de las clases del dominio la ejecución de los servicios.

La tabla A.36 muestra una relación de los atributos variables que aparecen en nuestro sistema y cual es su categorización. Pasemos a detallar cada uno de estos atributos.

Clase incidencia: para esta clase sus atributos variables son prioridad , fSolucion y fRechazo. Para el atributo prioridad, el único evento que lo modifica es asignarPrioridad y la categoría correspondiente es de estado ya que el nuevo valor es el valor recogido

```
-----------------------------------------------------
              ATRIBUTOS VARIABLES
-----------------------------------------------------
Incidencia (prioridad, fSolucion, fRechazo)
Personal (estado)
JefeMantenimiento (finicio, ffin)
Operario (turno)
Piezas (stockMinimo, stockMaximo, pUnit)
Pedido (estado,fRecepcion)
Maquina (conContrato, disponible, ultRevision)
Abs_Laboral (fechaFinal)
Revision (estado)
-----------------------------------------------------
```

Tabla A.36. Atributos variables

en el parámetro numPrioridad(independientemente del valor que tuviera anteriormente). El proceso es análogo para los atributos fSolucion y FRechazo.

Nombre	Acción portadora	Efecto de la accion	Condición de evaluación
prioridad	asignarPrioridad	numPrioridad	
fSolucion	solucionar	fechaSol	
fRechazo	rechazar	fechaRech	

Tabla A.37. Atributos de estado clase incidencia

Clase personal: el atributo estado se define como un atributo de situación puesto que el valor de este atributo se recoge en un dominio limitado. En concreto, este atributo puede tomar los valores 'trabajando' o 'no trabajando' dependiendo del servicio que se active. El nuevo valor del atributo viene especificado en la columna "nuevo valor".

Clase jefeMantenimiento: En este caso tenemos dos atributos que se categorizan de estado. La ejecución del servicio asignarPeriodo implica que los atributos finicio y ffin se actualicen con el valor introducido correspondiente.

Nombre	Valor actual	Acción portadora	Nuevo valor	Acción liberadora
estado	"trabaja"	insertarAbsentismo	"no trabaja"	finalizarAbsentismo
	"no trabaja"	finalizarAbsentismo	"trabaja"	insertarAbsentismo

Tabla A.38. Atributos de situación clase personal

Nombre	Acción portadora	Efecto de la accion	Condición de evaluación
finicio	asignarPeriodo	fechaInicio	
ffin	asignarPeriodo	fechaFin	

Tabla A.39. Atributos de estado clase resp_servicio

Clase Operario: Veamos la clase operario. El atributo turno de esta clase se actualiza con el valor correspondiente, el servicio responsable de esta actualización es asignarTurno.

Nombre	Acción portadora	Efecto de la accion	Condición de evaluación
turno	asignarTurno	nuevoTurno	

Tabla A.40. Atributos de estado clase Operario

Clase Pieza: los objetos de la clase pieza, tienen 4 atributos variables. Los atributos stockMinimo, stockMaximo y pUnit son atributos de estado y su semántica es idéntica a los atributos de la clase incidencia. En este caso cada servicio tiene asociadas cierta condición de evaluación que se debe de cumplir para que el efecto del servicio se lleve a cabo.

El atributo cantidad se define como atributo cardinal, y es modificado por dos servicios incrementarStock y decrementarStock. Estos servicios incrementan o decrementan la cantidad de una pieza en función de la cantidad que se pasa como argumento al servicio.

Clase pedido: la clase pedido tiene un atributo de situación denominado estado que representa el estado en el que puede encontrarse un pedido. La acción portadora (recibirPedido), establece el

Nombre	Acción portadora	Efecto de la accion	Condición de evaluación
stockMinimo	modificarStockMin	nuevoStockMin	nuevoStockMin $\leq$ stockMaximo
stockMaximo	modificarStockMax	nuevoStockMax	nuevoStockMax $\geq$ stockMin
pUnit	modificarPrecioUni	nuevoPrecio	

Tabla A.41. Atributos de estado clase pieza

Nombre	Acción incr	Acción decr	Condición de evaluación	Efecto de la acción
cantidad	incrementarStock	decrementarStock		cantidad + cantIncr cantidad - cantDecr

Tabla A.42. Atributos cardinales clase pieza

cambio de valor del atributo. También esta clase tiene un atributo de estado denominado fRecepcion.

Nombre	Valor actual	Acción portadora	Nuevo valor	Acción liberadora
estado	"pendiente"	recibirPedido	"recibido"	

Tabla A.43. Atributos de situación clase pedido

Nombre	Acción portadora	Efecto de la accion	Condición de evaluación
fRecepcion	recibirPedido	fecha	

Tabla A.44. Atributos estado clase pedido

Clase máquina: el atributo variable conContrato se especifica como un atributo de situación. Este atributo contiene información acerca de si una máquina tiene o no asociado un contrato de mantenimiento. La disponibilidad o no de la máquina se controla mediante el atributo disponible también categorizado como de

situación puesto que es modificado indistintamente por la activación de los servicios averiar o reparar de la clase máquina de entre un conjunto de valores posibles, en este caso true o false.

El atributo ultRevision se categoriza de estado y almacenará la fecha de la última revisión realizada a una máquina. El servicio regRevision es el encargado de reflejar este cambio.

Nombre	Valor actual	Acción portadora	Nuevo valor	Acción liberadora
conContrato	"false"	insertarContrato	"true"	
disponible	"false"	averiar	"true"	reparar
disponible	"true"	reparar	"false"	averiar

Tabla A.45. Atributos de situación clase máquina

Nombre	Acción portadora	Efecto de la accion	Condición de evaluación
ultRevision	regRevision	contratoMnto.-revision.fRevision	

Tabla A.46. Atributos estado clase máquina

Clase Abs_Laboral: para esta clase definimos un atributo de estado. Cuando se desea finalizar una situación de absentismo, mediante la activación del servicio finalizarAbsentismo se modifica el valor del atributo fechaFinal. De esta forma se cierra un periodo de absentismo.

Nombre	Acción portadora	Efecto de la accion	Condición de evaluación
fechaFinal	finalizarAbsentismo	fechaFin	

Tabla A.47. Atributos de estado clase Abs_Laboral

Clase revisión: la clase revisión tiene un atributo variable. A través de este atributo podemos conocer si la revisión en curso ha sido

realizada o todavía está pendiente de realizarse. El atributo estado es el encargado, en este caso de almacenar esta información.

Nombre	Valor actual	Acción portadora	Nuevo valor	Acción liberadora
estado	"pendiente"	registarRevision	"realizada"	

Tabla A.48. Atributos de situación clase revisión

Tras finalizar la especificación del modelo funcional, damos por finalizada la fase de modelado conceptual. El siguiente paso consiste en obtener la especificación formal del sistema a partir de la información capturada en esta fase. Este proceso se realiza de forma automática.

A.3 Especificación OASIS

A continuación se presenta la especificación formal OASIS que se obtiene y que sirve como repositorio de alto nivel del sistema.

```
CONCEPTUAL SCHEMA mant_hospital domains nat,bool,int,date, string

complex class Personal aggregation of
    Abs_Laboral(inclusive,dynamic,multivalued,disjoint,strict,
    null)

identification
    by_identification0 :  (DNI)  ;

constant_attributes
    dni : String ;
    apellidos : String ;
    nombre : String ;
    fnac : Date ;
    categoria : String ;
    poblacion : String ;
    provincia : String ;
    cpostal : String ;
    telefono : String ;

variable_attributes
    estado : String ;

private_events
  var
   tipo_baja: Nat;
   v_dni: String;
   fechaIni, fechaFin: Date;
  end_var
    crearPersona () new;
    destruirPersona (v_dni) destroy;

shared_events
    insertarAbsentismo (fechaIni,tipo_baja);
    finalizarAbsentismo (fechaFin);

valuation
    estado="trabaja" [insertarAbsentismo
           (fechaIni,tipo_baja)] estado="no trabaja" ;
    estado="no trabaja" [finalizarAbsentismo
           (fechaFin)] estado="trabaja" ;

preconditions
    jefeMant:insertarAbsentismo
    (fechaIni,tipo_baja) if estado = "trabaja" ;
    jefeMant:finalizarAbsentismo
```

```
        (fechaFin) if estado = "no trabaja" ;

    process
        Personal = jefeMant:crearPersona () Person0;
        Person0  = jefeMant:destruirPersona (v_dni) +
                   jefeMant:insertarAbsentismo
                   (fechaIni,tipo_baja)
                   Person1;

        Person1  = jefeMant:finalizarAbsentismo (fechaFin);
    end_class


    complex class absentismoLaboral aggregation of
        Tip_Baja(relational,static,multivalued,disjoint,strict,null)

    identification
        by_identification0 : dni  ;

    constant_attributes
        fechaInicio : Date ;

    variable_attributes
        fechaFinal:  Date ;

    private_events
      var
       tipo_baja: Nat;
       v_dni: String;
       fechaIni, fechaFin: Date;
      end_var
        crearAbsentismo () new;
        destruirAbsentismo (v_dni) destroy;

    shared_events
        insertarAbsentismo (fechaIni,tipo_baja);
        finalizarAbsentismo (fechaFin);

    process
        Abs_Laboral = crearAbsentismo () Abs_La0;
        Abs_La0     = destruirAbsentismo (v_dni);

    end_class


    complex class jefeMant specialization of Personal
     where categoria= "jefeMant"

    variable_attributes
        finicio : Date ;
        ffinal : Date ;

    private_events
```

```
  var
   fecha_ini,fecha_fin: Date;
  end_var
    asignarPeriodo (fecha_ini,fecha_fin);

constraints static
    ffinicio <= ffinal;

valuation

process
    jefeMant = jefeMant:Personal.crearPersona () Person0;
    Person0    = jefeMant:Personal.destruirPersona () +
                        asignarPeriodo (fecha_ini,fecha_fin) Person0 +
                        jefeMant:insertarAbsentismo (fechaIni,tipo_baja)
                        Person1;
    Person1           = jefeMant:finalizarAbsentismo (fechaFin);

end_class


complex class Maestro specialization of Personal where categoria =
"maestro"

variable_attributes
    area : String ;

private_events
  var
   codigo_area: Int;
  end_var
    asignarArea (codigo_area);

valuation

process
    Maestro = jefeMant:Personal.crearPersona () Person0;
    Person0 = jefeMant:Personal.destruirPersona (v_dni) +
              asignarArea (codigo_area) +
              jefeMant:insertarAbsentismo (fechaIni,tipo_baja)
              Person1;
    Person1 = jefeMant:finalizarAbsentismo (fechaFin);

end_class


complex class Operario specialization of Personal where categoria
= "operario"

variable_attributes
    turno : String ;

private_events
```

```
  var
   nuevo_turno: String;
  end_var
   asignarTurno (nuevo_turno);

valuation

process
    Operario = Personal.crearPersona () Person0;
    Person0  = Personal.destruirPersona (v_dni) + asignarTurno (nuevo_turno)
               jefeMant:insertarAbsentismo (fechaIni,tipo_baja)
               Person1;
    Person1  = jefeMant:finalizarAbsentismo (fechaFin);

end_class


class Tipo_Baja

identification
    by_identification0 :  (cod_baja)  ;

constant_attributes
    cod_baja : Nat ;
    descripcion : String ;

private_events
    crearTipoBaja () new;
    destruirTipoBaja () destroy;


process
    Tipo_Baja = crearTipoBaja() Tipo_B0;
    Tipo_B0   = destruirTipoBaja ();

end_class

complex class Incidencia aggregation of
    Usuario(relational,static,univalued,nodisjoint,flexible,not null),
    Area(relational,dynamic,univalued,nodisjoint,flexible,not null),
    Pedido(relational,dynamic,univalued,disjoint,flexible,not null),
    Maquina(relational,static,univalued,nodisjoint,flexible,null),
    Linea_Inciden(inclusive,dynamic,multivalued,disjoint,strict,null),
    Operario(relational,dynamic,univalued,nodisjoint,flexible,not null)

identification
    by_identification0 :  (codIncidencia)  ;

constant_attributes
    codIncidencia : Nat ;
    fPeticion : Date ;
    descripcion : String ;
```

```
variable_attributes
    prioridad : Nat ;
    fSolucion : Date ;
    fRechazo  : Date ;

private_events
  var
   fechaSol,fechaRech: Date;
   cd_incidencia,numPrioridad: Nat;
  end_var
  crearIncidencia () new;
  destruirIncidencia (cd_incidencia) destroy;
  asigPrioridad (numPrioridad);
  solucionar (fechaSol);
  rechazar (fechaRech);

shared_events
  insertarOperario ();
  borrarOperario ();
  insertarArea ();
  borrarArea ();
  insertarPedido ();
  borrarPedido ();


constraints static
    fPeticion <= fSolucion;

valuation
    [asigPrioridad (numPrioridad)] prioridad=numPrioridad;
    [solucionar (fechaSol)] fSolucion=fechaSol;
    [rechazar (fechaRech)] fRechazo=fechaRech;

transactions
    CLASIFICAR = insertarArea() . asigPrioridad(numPrioridad);

process
    Incidencia = Usuario:crearIncidencia () Incide0;
    Incide0    = CLASIFICAR incide1;
    incide1    = insertarOperario () incide2;
    incide2    = (Maestro:insertarPedido () +
                  Operario:insertarLinIncidencia()) incide2 +
                 (Operario:rechazar (fechaRech) +
                  Operario:solucionar (fechaSol)) incide3;
    incide3    = jefeMant:destruirIncidencia (cd_incidencia);

end_class


complex class Maquina aggregation of
    contrato_Mnto(inclusive,dynamic,univalued,disjoint,strict,null),
    revision(relational,dynamic,univalued,disjoint,strict,null)
```

```
identification
    by_identification0 :  (cod_referencia)  ;

constant_attributes
    cod_referencia : Nat ;
    ubicacion : String ;
    marca : String ;
    modelo : String ;
    descripcion : String ;


variable_attributes
    conContrato : Bool ;
    disponible : Bool ;

private_events
  var
   fecha,fecha_reparada,fecha_averiada: Date;
   cd_referencia,c_contrato, c_revision: Nat;
  end_var
    crearMaquina () new;
    destruirMaquina (cd_referencia) destroy;
    reparar ();
    averiar ();

shared_events
    insertarContrato ();
    registrarRevision (c_revision);


valuation
    conContrato=FALSE [insertarContrato()] conContrato=TRUE ;

preconditions
    insertarContrato () if conContrato = FASE ;

process
    Maquina = jefeMant:crearMaquina () Maquin0;
    Maquin0 = Maestro:averiar () Maquin1
              + jefeMant:destruirMaquina(cd_referencia) +
              (registrarRevision (c_revision) +
              insertarcontrato ()) Maquin0;
    Maquin1 = Maestro:reparar () Maquin0;

end_class


complex class contrato_Mnto aggregation of
    Empresa_Externa(relational,static,univalued,nodisjoint,
    strict,not null),
    revision(inclusive,dynamic,multivalued,disjoint,
    strict,not null)
```

```
identification
    by_identification0 :  (cod_contrato)  ;

constant_attributes
    codContrato : Nat ;
    fContrato : Date ;
    descripcion : String ;

private_events
    crearContrato () new;
    destruirContrato () destroy;

shared_events
    insertarContrato ();
    insertarRevision ();

process
    contrato_Mnto = crearContrato () contra0;
    contra0 = destruirContrato () + insertarRevision () contra0;

end_class


class Empresa_Externa

identification
    by_identification0 :  (cifEmpresa)  ;

constant_attributes
    cifEmpresa : String ;
    nombre : String ;
    persContacto : String ;
    direccion : String ;
    cpostal : String ;
    poblacion : String ;
    provincia : String ;
    pais : String ;
    telefono : String ;
    fax : String ;


private_events
  var
   nif_empresa: Nat;
  end_var
    crearEmpresa () new;
    destruirEmpresa (nif_empresa) destroy;

process
    Empresa_Externa = jefeMant:crearEmpresa () Empres0;
    Empres0 = jefeMant:destruirEmpresa (nif_empresa);
```

```
end_class


class revision

identification
    by_identification0 :  (codRevision)  ;

constant_attributes
    codRevision : Nat ;
    fRevision : Date ;
    descripcion : String ;

variable_attributes
    estado : String ;

private_events
  var
   fecha_actual,fecha: Date;
   c_revision: Int;
   cd_revision: Nat;
  end_var
    crearRevision () new;
    destruirRevision (c_revision) destroy;

shared_events
    registrarRevision (fecha_actual);
    insertarRevision (cd_revision,fecha);


valuation
    estado="no realizada"
    [registrarRevision (cd_revision,fecha)]
    estado="realizada" ;

process
    revision = jefeMant:crearRevision ()
               revisi0;
    revisi0  = Maestro:insertarRevision
              (cd_revision,fecha) revisi1;
    revisi1  = Maestro:registrarRevision
              (fecha_actual) revisi2;
    revisi2  = jefeMant:destruirRevision (c_revision);

end_class

class Servicio

identification
    by_identification0 :  (codServicio)  ;

constant_attributes
    codServicio : Nat ;
```

```
    descripcion : String ;

private_events
  var
   cd_servicio: Nat;
  end_var
    crearServicio () new;
    destruirServicio (cd_servicio) destroy;

process
    Servicio = jefeMant:crearServicio () Servic0;
    Servic0 = jefeMant:destruirServicio (cd_servicio);

end_class


class Area

identification
    by_identification0 :  (codArea)  ;

constant_attributes
    codArea : Nat ;
    descripcion : String ;

private_events
  var
   cd_area: Nat;
  end_var
    crearArea () new;
    destruirArea (cd_area) destroy;


process
    Area = jefeMant:crearArea () Area0;
    Area0 = jefeMant:destruirArea (cd_area);

end_class


complex class Pedido aggregation of
    lineaPedido(inclusive,dynamic,multivalued,
    disjoint,strict,not null)

identification
    by_identification0 :  (codPedido)  ;

constant_attributes
    codPedido : Nat ;
    fPedido : Date ;

variable_attributes
    estado : String ;
```

```
        fRecepcion : Date;

    private_events
      var
       cd_pedido: Nat;
      end_var
        crearPedido () new;
        destruirPedido () destroy;
        recibirPedido (cd_pedido);

    shared_events
        insertarLineaPedido () ;
        insertarPedido ();
        borrarPedido ();

    valuation
        estado="pendiente" [recibirPedido (cd_pedido)]
        estado="recibido" ;

    process
        Pedido  = crearPedido () Pedido0;
        Pedido0 = (recibirPedido (cd_pedido) Pedido1)
                  + insertarLineaPedido () Pedido0;
        Pedido1 = Maestro:destruirPedido ();

    end_class


    complex class lineaPedido aggregation of
        Pieza(relational,static,univalued,nodisjoint,
        flexible,not null)

    identification
        by_identification0 :  (numLinea)  ;

    constant_attributes
        numLinea : Nat ;

    private_events
        insertarLineaPedido () new;
        borrarLineaPedido () destroy;

    process
        lineaPedido = insertarLineaPedido () Linea_0;
        Linea_0 = borrar ();

    end_class


    class Pieza

    identification
        by_identification0 :  (codPieza)  ;
```

```
constant_attributes
    codPieza : Nat ;
    descripcion : String ;
    unidadMedida : String ;

variable_attributes
    stockMinimo : Nat ;
    stockMaximo : Nat ;
    pUnit : Nat ;
    cantidad : Nat ;

private_events
  var
   cd_pieza,nuevo_precio,nuevo_stock_min,nuevo_stock_max
   ,cantIncr,cantDecr: Nat;
  end_var
    crearPieza () new;
    destruirPieza (cd_pieza) destroy;
    modificarPrecioUni (nuevo_precio);
    modificarStockMin (nuevo_stock_min);
    modificarStockMax (nuevo_stock_max);
    incrementarStock (cantIncr);
    decrementarStock (cantDecr);

constraints static
    cantidad >= 0;
    stock_minimo <= stock_maximo;

valuation
    nuevo_stock_min <= stockMaximo
    [modificarStockMin (nuevo_stock_min)]
    stockMinimo=nuevo_stock_min ;
    nuevo_stock_min > stockMaximo
    [modificarStockMin (nuevo_stock_min)]
    stockMinimo=stockMinimo ;
    nuevo_stock_max < stock_minimo
    [modificarStockMax (nuevo_stock_max)]
    stockMaximo=stockMaximo ;
    nuevo_stock_max >= stockMinimo
    [modificarStockMax (nuevo_stock_max)]
    stockMaximo=nuevo_stock_max ;
    (cantidad + cantIncr) < stockMaximo
    [incrementarStock (cantIncr)]
    cantidad= cantidad + cantIncr ;
    (cantidad - cantDecr) < stockMaximo
    [decrementarStock (cantDecr)]
    cantidad= cantidad - cantDecr ;
    [modificarPrecioUni (nuevo_precio)]
     pUnit=nuevo_precio;

preconditions
    decrementarStock (cantDecr)
```

```
        if (cantidad - cantDecr) > stockMinimo ;
        Pedido:incrementarStock (cantIncr)
        if (cantidad + cantIncr) > stockMaximo ;

    triggers
        Object :: insertarPedido () if cantidad < stockMinimo;
    process
        Pieza  = crearPieza () Pieza0;
        Pieza0 = destruirPieza (cd_pieza) +
                (modificarPrecioUni (nuevo_precio)
                + modificarStockMin (nuevo_stock_min)
                + modioficarStockMax (nuevo_stock_max)
                + decrementarStock (cantDecr) +
                Pedido:incrementarStock (cantIncr)) Pieza0;

    end_class


    complex class Linea_Inciden aggregation of
        Pieza(relational,static,univalued,nodisjoint,
        flexible,not null)

    identification
        by_identification0 :  (codLinea)  ;

    constant_attributes
        codLinea : Nat ;

    private_events
        crearLineaIncidencia () new;
        destruirLineaIncidencia () destroy;

    process
        Linea_Inciden = crearLineaIncidencia () Linea_0;
        Linea_0 = destruirLineaIncidencia ();

    end_class



    global_interactions

    transactions
        recibirPiezas = Pedido.recibirPedido(cd_pedido).
        [Maestro]:Pieza.incrementarStock(cantIncr) ;
        usarPiezas = Incidencia.solucionar(fecha_sol).
        [Maestro]:Pieza.decrementarStock(cantDecr) ;

    end_global_interactions

    END CONCEPTUAL SCHEMA
```

A.4 Aplicación generada

A continuación se presentan algunas pantallas que propocionan una visión general del aspecto del prototipo de aplicación generado y accesible a través del sistema JEM. Las pantallas ilustran como se satisfacen los requisitos iniciales en la aplicación.

Requisito 1

El **personal** del servicio de mantenimiento del hospital se encuentra clasificado por categorías. Existe un **jefe de mantenimiento**, un **maestro** por cada una de las **áreas** en las que se encuentra dividido el servicio y **operarios** asignados a ellas.

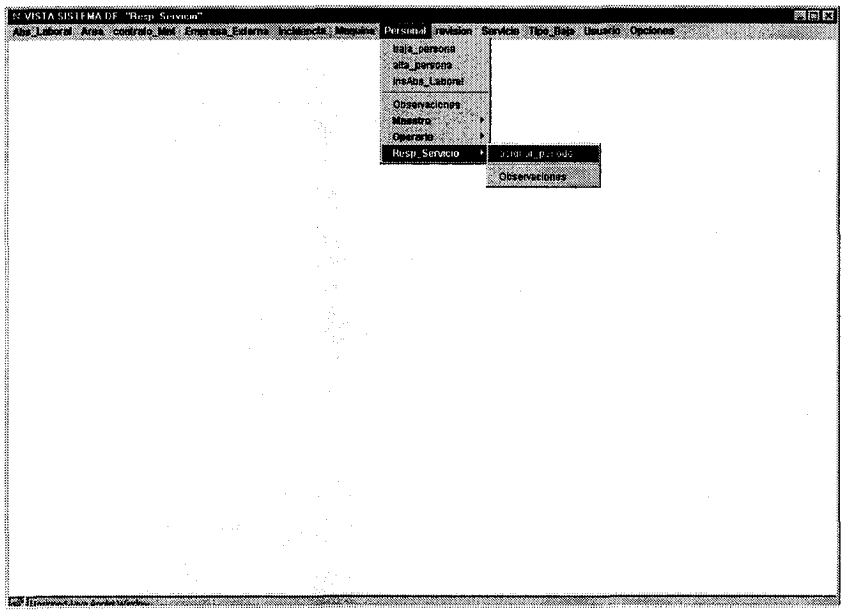


Figura A.20. *Menú de especialización.*

La Figura A.20 ilustra la vista del sistema para el agente jefeMantenimiento (o también llamado Resp_servicio). Se puede apreciar como en el prototipo generado, las especializaciones aparecen de

forma natural el menú de opciones asociado a la clase 'padre' de la especialización. En este caso, la clase personal puede especializarse según el requisito 1 en tres tipos de personal diferente (maestro, operario, Resp_servicio), como se comentó en la elaboración del modelo conceptual. La Figura ilustra la situación en que el usuario del prototipo activa el servicio asignarPeriodo (asignar_periodo en la imagen) de la clase Resp_servicio.

Requisito 2

La gestión del absentismo laboral requiere:

2.1 Llevar el control del **absentismo laboral** de los empleados del servicio de mantenimiento.

2.2 En cualquier momento se puede actualizar la ficha de absentismo de un empleado introduciendo el **tipo de baja** correspondiente, también se puede consultar durante el año en curso los días que ha faltado a trabajar y porqué.

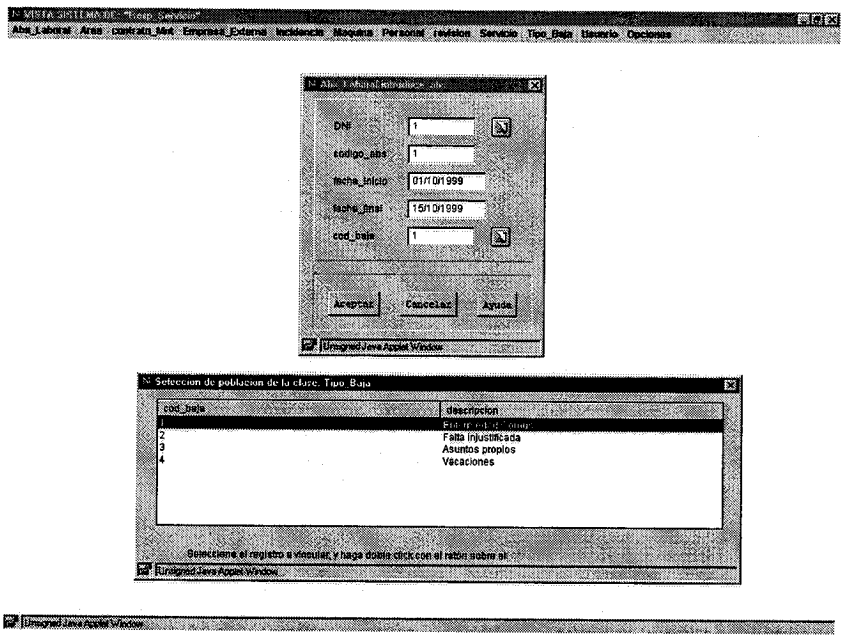


Figura A.21. *Registrar un absentismo laboral.*

Para ilustrar el comportamiento del prototipo generado en la satisfacción de este requisito se presentan las Figuras A.21 y A.22. La Figura A.21 muestra una situación en la cual el usuario ha activado el servicio insertarAbsentismo. La ejecución de este servicio permite la introducción de la información necesaria para registrar una situación de absentismo de un empleado. La Figura A.21 ilustra la introducción de información relevante para la activación de este servicio (en particular, dni de la persona afectada, fechas de inicio y fin del periodo y código de la baja). Obsérvese que el código de la baja (cod_baja) se selecciona de entre la población de la clase Tipo_Baja. En este caso se selecciona una situación de enfermedad común.

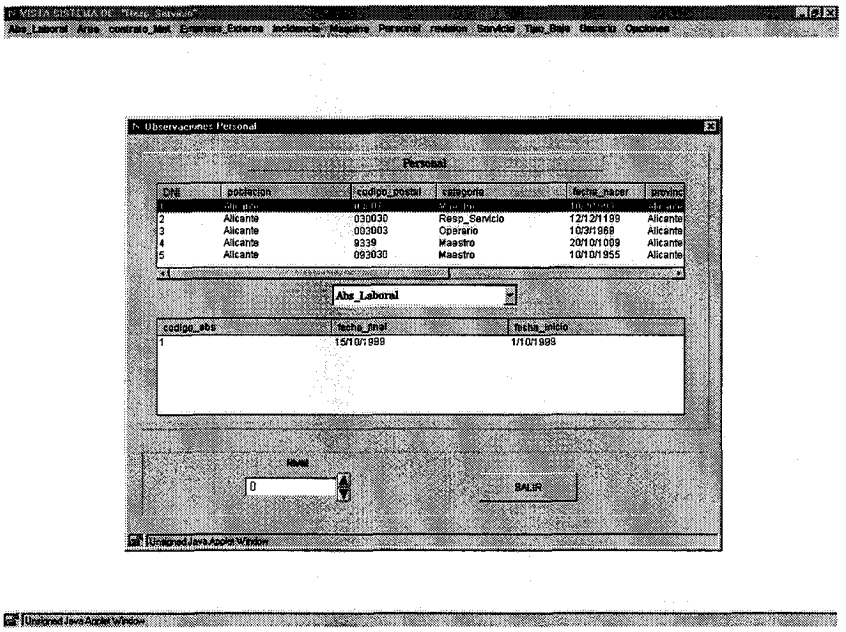


Figura A.22. *Consulta de absentismo laboral.*

La Figura A.22 muestra la ejecución de una observación sobre la clase persona. En particular muestra para el empleado seleccionado las situaciones de absentismo laboral que ha sufrido.

Requisito 3

Las **máquinas** del hospital pueden estar sujetas a **contratos de mantenimiento** que vinculan las máquinas con **empresas externas** encargadas por un lado de ofrecer asistencia ante posibles averías y por otro a realizar ciertas **revisiones** anuales para controlar el buen funcionamiento de éstas.

Para relacionar una máquina con un contrato de mantenimiento, primero hay que crearlo. La Figura A.23 muestra la activación del servicio crearContrato (en la figura contrato_mnt.introducir), donde se introduce la información relevante para el contrato. Según la descripción del requisito, se introduce (entre otra), información de la máquina sobre la que recae el contrato (cod_referencia), la empresa externa responsable del contrato (cif_empresa) y, finalmente, información sobre las revisiones asociadas al mismo.

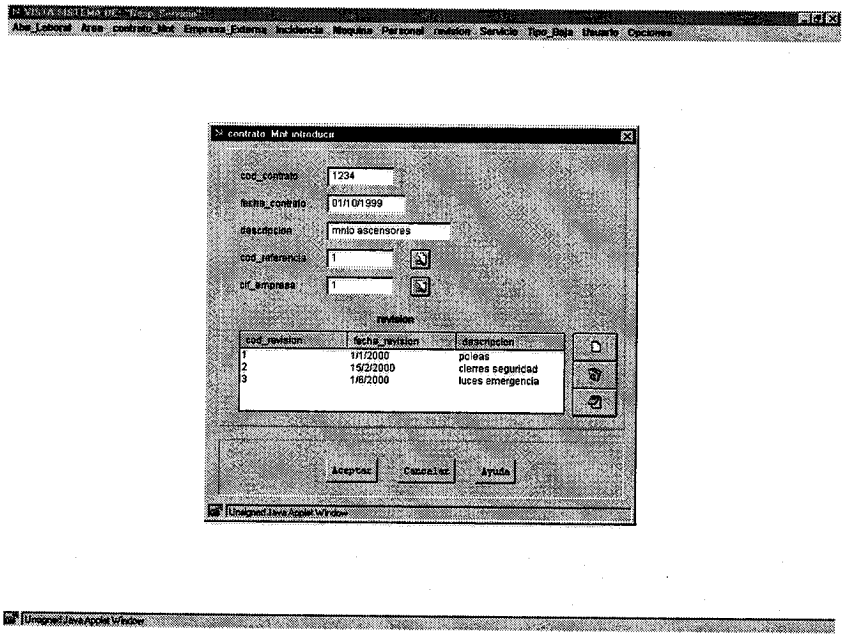


Figura A.23. *Contratos de mantenimiento de máquinas.*

Requisito 4

Se desea conocer, cual es la fecha de la última revisión realizada a una **máquina** que tiene un **contrato de mantenimiento**.

De forma análoga a como se hizo para el requisito 2, se puede consultar el estado de las revisiones de una máquina realizando la consulta (observación) correspondiente sobre la máquina involucrada tal y como ilustra la Figura A.24.

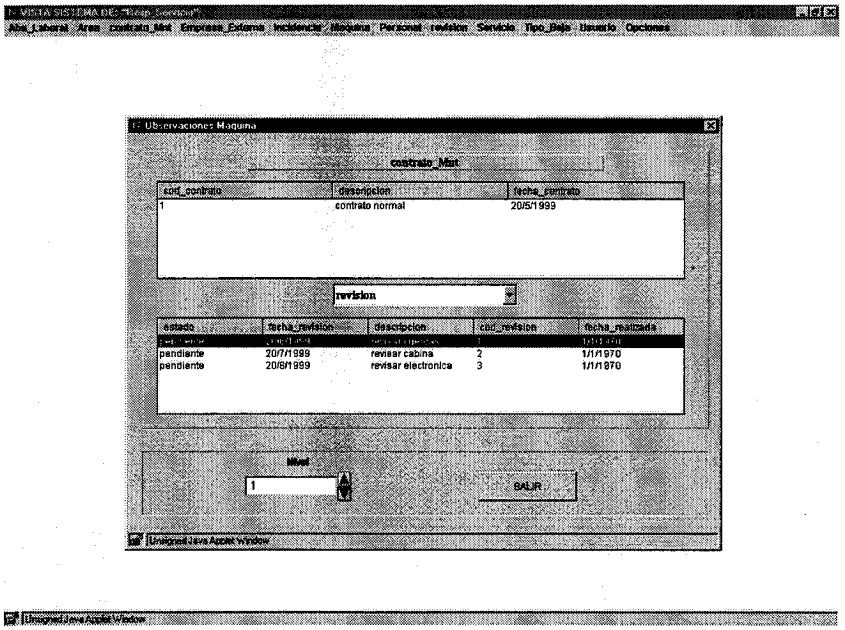


Figura A.24. *Estado de las revisiones.*

Requisito 5

Cuando un operario resuelve una **incidencia**, toma nota en el parte de incidencia de las **piezas** y la cantidad de las mismas que ha utilizado.

La activación del servicio insertarLineaIncidencia (en la Figura A.25 Linea_Incidencia.insertar_linea) permite la introducción de la infor-

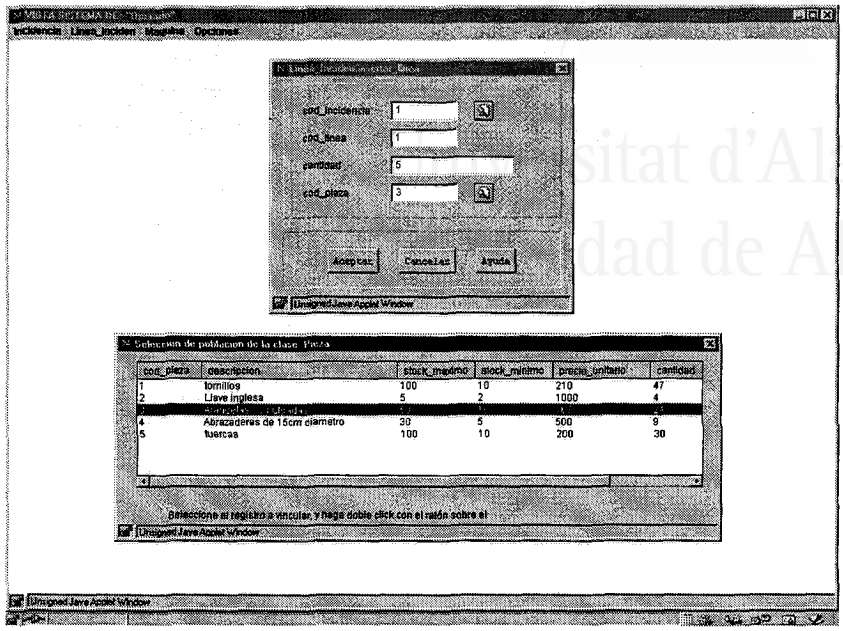


Figura A.25. *Insertar líneas de incidencia.*

mación necesaria para satisfacer este requisito. En concreto, información sobre las piezas y la cantidad de ellas usadas.

Requisito 6

Cuando no se dispone de las **piezas** necesarias para resolver una avería o **incidencia**, se debe de realizar un **pedido** al servicio de suministros del hospital para que compre las piezas necesarias.

Como se puede ver en la Figura A.26, este requisito se satisface mediante la activación del servicio crearPedido (en la figura Pedido.introducir) de la clase pedido, a partir del cual se puede introducir toda la información relevante del pedido que se necesita.

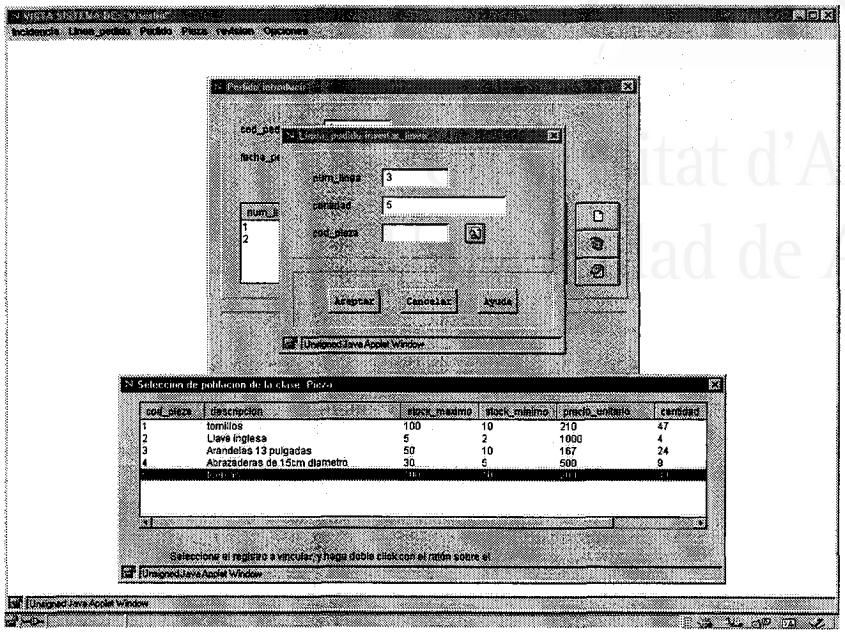


Figura A.26. *Introducir un pedido.*

Requisito 7

Cuando se produce una incidencia en el hospital:

7.1 El responsable del servicio afectado rellena el parte de **incidencias**.

7.2 Diariamente el **jefe de mantenimiento**, clasifica las incidencias asignándoles la prioridad y el **área** que la atenderá.

7.3 Los partes clasificados se reparten entre los **maestros**, quienes asignarán los **operarios** que atenderán la reparación.

7.4 En caso de que la incidencia afecte a una **máquina** con **contrato de mantenimiento** se avisará a la **empresa externa** contratada para que atienda la avería.

Finalmente, las Figuras A.27, A.28, y A.29 ilustran las activaciones correspondientes de servicios que se han de seleccionar para

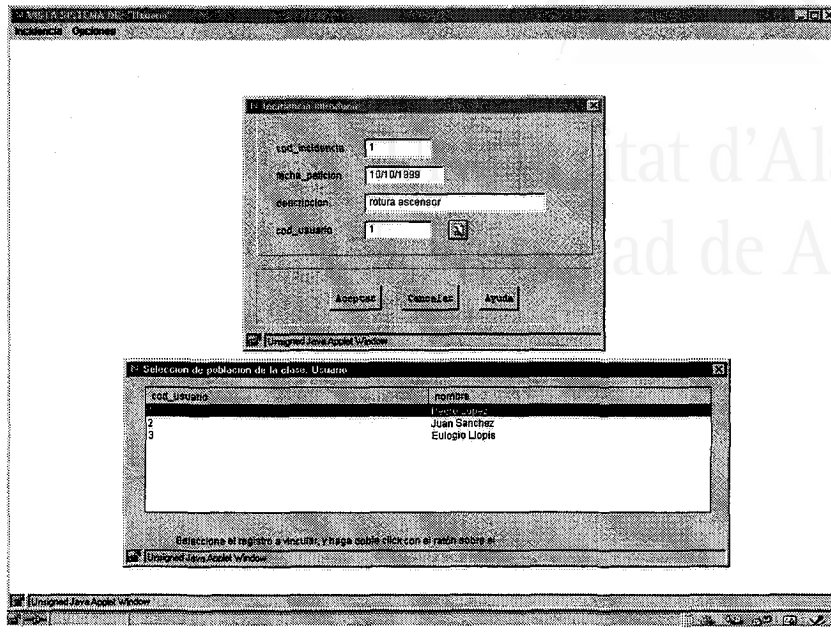


Figura A.27. *Registrar la ocurrencia de una incidencia.*

satisfacer la funcionalidad especificada en el requisito 7. En concreto, la Figura A.27 ilustra la situación que se produce cuando se activa el servicio correspondiente para registrar la ocurrencia de una incidencia (crearIncidencia, en la Figura representado como Incidencia.introducir), clasificar la misma introduciendo información relevante sobre la prioridad y el área que atenderá la incidencia (ver Figura A.28) y, por último, la resolución de una incidencia (ver Figura A.29) mediante la activación del servicio correspondiente de la clase incidencia (solucionar).

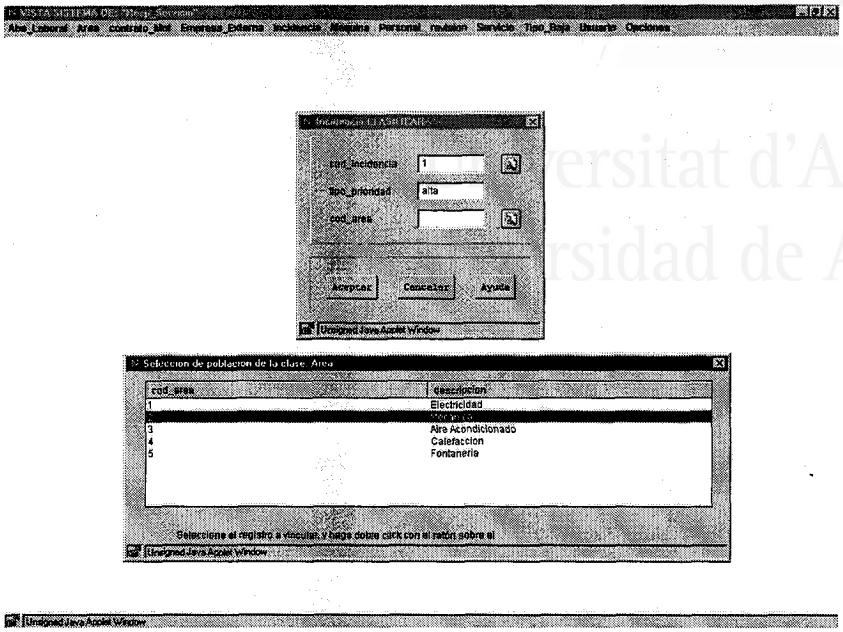


Figura A.28. *Clasificar una incidencia.*

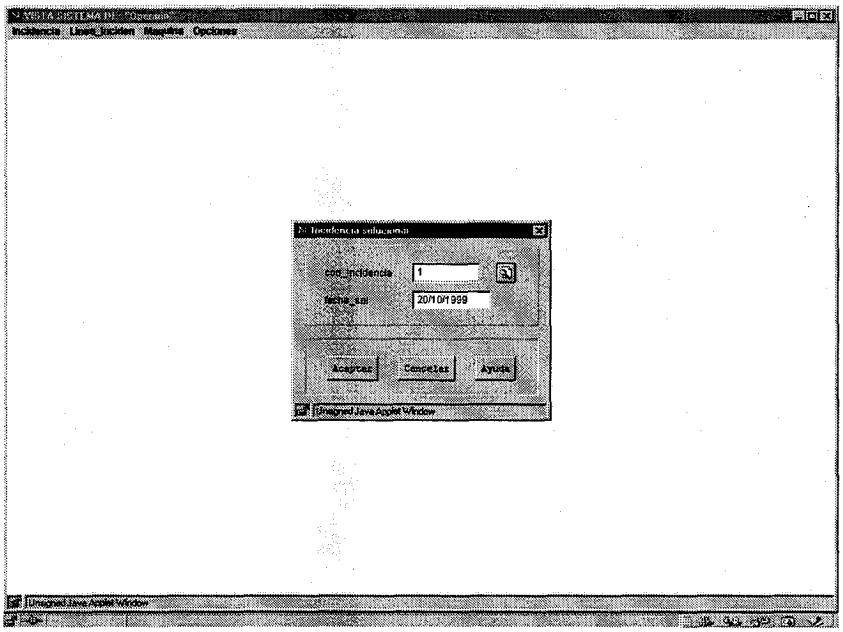


Figura A.29. *Solucionar una incidencia.*

B. Especificación Sintáctica de la Gramática OASIS

A continuación se presentan las distintas construcciones sintácticas de la versión 2.2 del lenguaje OASIS usada.

B.1 Introducción

Se sigue un desarrollo de presentación de abajo a arriba (bottom-up), empezando con las construcciones más simples y finalizando con la construcción que representa el esquema conceptual.

Notación. Para la construcción de la gramática se han seguido las siguientes reglas:

- los símbolos terminales están en **negrita** y los no terminales entre '<' y '>'.
- la derivación de un símbolo no terminal se expresa con '::='.
- todos los símbolos situados entre '[' y ']' son opcionales.
- las alternativas se separan con '|'.
- el símbolo $\in$ representa la cadena vacía.

Especificación Léxica. Todos los símbolos no terminales acabados en -id son identificadores y pueden formarse con una cadena de letras, dígitos y subrayados, que empiece por letra ($letra\{letra|digito\}*$). Los identificadores usados son:

```
<var_id>                : variable definida en los enunciados
<agent_id>              : agente que activa un evento
```

```
<domain_param_id>       : parámetros de un dominio
<domain_id>             : nombre del dominio
<funct_id>              : función de un dominio
<conceptual_schema_id>  : nombre del esquema conceptual
<class_id>              : nombre de la clase
<ident_id>              : atributo de identificación
<const_atr_id>          : atributo constante
<var_atr_id>            : atributo variable
<derived_atr_id>        : atributo derivado
<class_atr_id>          : atributo de clase
<atr_id>                : conjunto de atributos de la clase
<event_id>              : nombre del evento
<transaction_id>        : nombre de la transacción
<process_id>            : nombre del proceso
```

La formación de las constantes ha de seguir las siguientes reglas:

```
<bool_const>    ::= true | false
<nat_const>     ::= <digito> <digito>*
<int_const>     ::= <signo>  <nat_const> (el signo puede ser + o -)
<real_const>    ::= <signo> [ <nat_const> ] . <nat_const> e <int_const> |
                    <signo> <nat_const> . [ <nat_const> ] e <int_const>
<string_const>  ::= " <simbolo>* "
<time_const>    ::= <d>[<d>]  :  <d>[<d>]  [ : <d>[<d>]
                    [ : <d>[<d>][<d>]  ] ]
                        ( <d> representa un dígito)
<date_const>    ::= <d> [ <d> ] - <d> [ <d> ] - <d><d> [ <d><d> ]
<const>         ::= <nat_const> | <int_const> |  <real_const> |
                    <bool_const> | <string_const> |
                    <time_const> | <date_const>
```

B.2 Fórmulas

Operadores. Los operadores aritméticos binarios se utilizan para los términos.

```
<binary_op>     ::= + | - | * | /
```

Los operadores relacionales se utilizan para las fórmulas.

```
<rel_op>        ::= = | < > | < | <= | > | >=
```

Los operadores booleanos se utilizan para las fórmulas.

```
<bool_op>       ::= and | or
```

Los operadores de proceso se emplean para las fórmulas de las transacciones y procesos.

```
<process_op>        ::= + | .
```

Las funciones integradas se emplean para formar los términos en una asociación.

```
<asoc_functions>    ::= sum | avg | count | max | min
```

Tipos de variables y atributos. Cuando se definen atributos en una clase o variables en un enunciado hay que especificar el tipo. Tenemos tres alternativas posibles:

1. identificador de clase: se crea una instancia de la clase;
2. identificador de dominio: se crea una instancia del dominio;
3. tipo predefinido: se obtiene un tipo básico de Oasis previamente especificado en la sección domains (ver dominios más adelante).

```
<predef_type>   ::= nat | int | real | bool | string | time | date
<type>      ::= <predef_type> | <class_id> | <domain_id>
```

Términos. Los términos se utilizan para formar las fórmulas, se distinguen dos tipos:

1. Por una parte los términos empleados para las fórmulas de las evaluaciones, derivaciones, precondiciones, restricciones (estáticas y dinámicas) y disparos. Estos pueden ser constantes, variables o atributos de cualquier clase de la especificación. Además se pueden combinar con operadores para formar otros términos y utilizar paréntesis para agrupar o alterar la precedencia de operadores. También son términos las funciones de los dominios, que pueden tener parámetros que a su vez serán términos, y por último las funciones integradas en las asociaciones.

```
<term>      ::= <const> | <var_id> | [ <class_id_seq> . ] <atr_id>
              | <term> <binary_op> <term>
              | <funct_id> [ ( <term_list> ) ]
              | <asoc_functions> ( <class_id> . <atr_id> )
                  [ where ( rbas_wff ) ]
              | ( <term> )
```

2. En segundo lugar tenemos los términos que representan las acciones o cambios de estado en las clases y que son event y transaction. Ambos se usan para construir las fórmulas de las transacciones y procesos y en las cláusulas de los disparos, pero event se usa además en las fórmulas de las restricciones de integridad dinámicas, en las cláusulas de las evaluaciones y en las de las precondiciones.

```
<agent>       ::= <agent_id> | [ <agent_id_list> ]
<event>       ::= [ <agent>  : ] [ <class_id_seq> . ]
                    <event_id> ( [<var_id_list> ]  )
<transaction> ::= [ <agent>  : ] [ <class_id_seq> . ]
                    <transaction_id>
<action>      ::= <event> | <transaction>
```

Tipos de Fórmulas. Se distinguen seis tipos de fórmulas.

1. El primer tipo es el más básico y se emplea principalmente para construir las cláusulas de las precondiciones, condiciones de especialización, y en las fórmulas de transacciones y procesos. Éstas fórmulas se pueden formar mediante constantes booleanas, por combinación de términos con operadores relacionales y por combinación de fórmulas con conectivas lógicas y/o paréntesis.

```
<bas_wff>   ::= <term> <rel_op> <term>
              | not <bas_wff> | <bas_wff> <bool_op> <bas_wff>
              | ( <bas_wff> )
```

2. El segundo tipo es el que se utiliza para las evaluaciones y derivaciones. Se forman como las anteriores y también con if then else. La palabra clave any se utiliza en clases asociadas para indicar que la fórmula será cierta si la cumple al menos un componente de la asociación.

```
<atr_wff>   ::= <term> <rel_op> <term>
            | not <atr_wff> | <atr_wff> <bool_op> <atr_wff>
            | ( <atr_wff> )
            | if <atr_wff> then <atr_wff> [ else <atr_wff> ]
              end if
            | any <atr_wff>
```

3. El tercer tipo es el empleado para las cláusulas de las restricciones de integridad estáticas. Éstas se forman mediante la combinación de términos con operadores relacionales, conectivas lógicas y paréntesis. Para las asociaciones, se puede emplear una restricción que se cumplirá si todos los objetos del conjunto (for all members) tienen el mismo valor de un atributo determinado.

```
<sic_wff>   ::= <term> <rel_op> <term>
            | not <sic_wff> | <sic_wff> <bool_op> <sic_wff>
            | ( <sic_wff> )
            | for all members same ( <class_id> . <atr_id> )
```

4. El cuarto tipo de fórmulas se emplea en las cláusulas de las restricciones de integridad dinámicas. Pueden formarse con un término del tipo event o mediante comparación de dos términos (con un operador relacional). También se puede usar los operadores lógicos, paréntesis y los operadores temporales always, sometimes con o sin until y since.

```
<dic_wff>   ::= <event>
            | <term> <rel_op> <term>
            | not <dic_wff> | <dic_wff> <bool_op> <dic_wff>
            | ( <dic_wff> )
            | always <dic_wff>
            | always <dic_wff> until <dic_wff>
            | always <dic_wff> since <dic_wff>
            | sometimes <dic_wff>
            | sometimes <dic_wff> until <dic_wff>
            | sometimes <dic_wff> since <dic_wff>
```

5. A continuación se presentan las fórmulas que se utilizan en las cláusulas de los disparos. Éstas son iguales que las del segundo tipo a diferencia de que aquí no se puede usar la construcción if then else.

```
<trg_wff>    ::= <term> <rel_op> <term>
               | not <trg_wff> | <trg_wff> <bool_op> <trg_wff>
               | ( <trg_wff> )
               | any <trg_wff>
```

6. Por último se tienen las fórmulas empleadas para las cláusulas de las transacciones y de los procesos, que son muy parecidas. Pueden formarse con eventos del tipo event o transaction, combinarlos con los operadores de procesos, emplear paréntesis o visualizar una variable con el operador ?.

```
<transaction_wff>   ::= <action>
                  | ? <var_id>
                  | <transaction_wff> <process_op> <transaction_wff>
                  | ( <transaction_wff> )
                  |  <atr_wff>  <transaction_wff>

<process_wff>       ::= <action>
                  | ? <var_id>
                  | [ <class_id_seq> . ] <process_id>
                  | <process_wff> <process_op> <process_wff>
                  | ( <process_wff> )
                  |  <atr_wff>  <process_wff>
```

B.3 Enunciados

En este apartado se presentan los enunciados que se utilizan para componer las clases elementales y complejas, y los dominios.

Clases Elementales.

Identificación. La parte de la identificación está formada por una palabra clave seguida por una lista de declaraciones.

```
<ident_decl>        ::= <ident_id> : ( <const_atr_id_list> ) ;
<identification_st> ::= identification
                <ident_decl_items>
```

Atributos. Para la definición de los atributos, tres posibilidades: constantes, variables y derivados. En todos los casos el enunciado

comienza con una palabra clave y a continuación una lista de declaraciones. Cada declaración consta de una lista de identificadores separados por comas, el tipo de los atributos y opcionalmente la palabra clave class para indicar que el atributo es de clase. En los atributos variables se puede añadir un valor de inicialización después del tipo. A continuación se presentan las reglas junto con algunos ejemplos.

```
<const_decl>          ::= <const_atr_id_list> : <type> [ class ] ;
<constant_atr_st>     ::= constant attributes
                  <const_decl_items>


<var_atr_decl>        ::= <var_atr_id_list> : <type> [ ( <const> ) ]
                          [ class ] ;
<variable_atr_st>     ::= variable attributes
                  <var_atr_decl_items>


<derived_decl>        ::= <derived_atr_id_list> : <type> [ class ] ;
<derived_atr_st>      ::= derived attributes
                  <derived_decl_items>
```

Eventos Privados. Para las clases elementales se pueden definir variables que tienen ámbito local a la clase en que se definen.

```
<variables> ::= var <var_decl_items> end var
<var_decl>  ::= <var_id_list> : <type> ;
```

Los eventos privados se definen con un identificador de evento, una lista de parámetros y opcionalmente, si es new o destroy.

```
<event_type>              ::= new | destroy
<private_event_decl>      ::= <event_id> ( [ <var_id_list>] )
                              [ <event_type> ]  ;
<private_events_st>       ::= private events [ <variables> ]
                  <private_event_decl_items>
```

Eventos compartidos. La especificación de eventos compartidos es similar a la de los privados, a excepción de que no se utilizan las palabras clave new y destroy, y de que es necesario especificar con qué clases se comparten los eventos.

```
<shared_event_decl> ::= <event_id> ( [<var_id_list>] )
```

```
                        with <class_id_list> ;
<shared_events_st>  ::= shared events [ <variables> ]
                    <shared_event_decl_items>
```

Restricciones. La declaración de restricciones se divide en dos secciones: estáticas y dinámicas. En cada una de ellas se utilizan las respectivas fórmulas. Ambas partes son opcionales.

```
<constraints_st>    ::= constraints [ <variables> ]
                        [ static <sic_wff_seq> ; ]
                        [ dynamic <dic_wff_seq> ; ]
```

Evaluaciones. El enunciado de evaluaciones se especifica con una palabra clave, una lista opcional de declaración de variables y la lista de las cláusulas de las evaluaciones. Estas cláusulas están formadas por una fórmula opcional de las del segundo tipo y que indica el estado inicial o precondición que debe cumplirse para que se pueda aplicar la evaluación. A continuación se debe especificar el término del tipo event (entre corchetes) que indica la ocurrencia del evento correspondiente. Después se puede especificar una parte opcional (sólo es válida para las asociaciones) y por último una fórmula que representa el estado final o postcondición.

```
<for_all_members>   ::= for all members [ of <class_id> ]
<valuation_decl>    ::= [ <atr_wff> ] [ <event> ]
                        [ <for_all_members> ] <atr_wff> ;
<valuation_st>      ::= valuation [ <variables> ]
                    <valuation_decl_items>
```

Derivaciones. En las cláusulas de las derivaciones se utilizan fórmulas del tipo 2. Se reducen a la asignación sobre un atributo derivado de una fórmula.

```
<val>               ::= <term> | <atr_wff>
<derivation_decl>   ::= [  <atr_wff>  ]
                        <derived_atr_id> = <val> ;
<derivation_st>     ::= derivation [ <variables> ]
                    <derivation_decl_items>
```

Precondiciones. Un término del tipo event representa el evento sobre el que se añade una precondición. La precondición consta de la palabra clave if y una fórmula del tipo 1. Para las asociaciones

se puede especificar que la fórmula la cumplan todos los miembros del conjunto (for all members).

```
<prec_decl>          ::= <event> if [ <for_all_members> ]
                             <bas_wff> ;
<preconditions_st>   ::= preconditions [ <variables> ]
                         <prec_decl_items>
```

Disparos. En los disparos, la condición del disparo se indica con una fórmula del tipo 5. La acción a realizar se expresa con un término del tipo event o transaction que opcionalmente puede ser prefijado con el destino de la acción para indicar a quien afecta el disparo. Los destinos pueden ser el objeto mismo (self), toda la población de la clase (class) o un objeto de la clase (nombre de la clase). En el caso de asociaciones, la acción a realizar puede ir precedida por indicando que la acción debe ocurrir para todos los componentes de la instancia.

```
<destination>        ::= self | class | <class_id>
<trigger_decl>       ::= [ <for_all_members> ] [ <destination> :: ]
                                 <action> if <trg_wff> ;
<triggers_st>        ::= triggers [ <variables> ]
                       <trigger_decl_items>
```

Transacciones. La declaración de transacciones se establece mediante un identificador de transacción y una fórmula.

```
<transact_decl>      ::= <transaction_id> = <transaction_wff> ;
<transactions_st>    ::= transactions [ <variables> ]
                       <transact_decl_items>
```

Procesos. La declaración de procesos es similar a la de transacciones pero utilizando las fórmulas para procesos.

```
<process_decl>   ::= <process_id> = <process_wff> ;
<processes_st>   ::= processes [ <variables> ]
                     <process_decl_items>
```

Clases Complejas. En este apartado se presenta la sintaxis adicional necesaria para especificar clase complejas. En Oasis se definen cuatro tipos de clases complejas.

```
<complex_st>     ::= <aggregation> | <association> |
                     <specialization> | <generalization>
```

Agregación. Para definir una agregación a partir de un conjunto de clases, para cada clase del conjunto se indica el identificador de la clase y los parámetros de ésta en la agregación. En concreto, seis parámetros. Si no se especifica alguno de ellos, se asume el valor por defecto (subrayado).

```
<arg1>       ::= relational | inclusive
<arg2>       ::= static | dynamic
<arg3>       ::= disjoint | nodisjoint
<arg4>       ::= strict | flexible
<arg5>       ::= univalued | multivalued [ ( <nat_const> ) ]
<arg6>       ::= null | not null
<arg>        ::= <arg1> | <arg2> | <arg3> | <arg4> | <arg5> | <arg6>
<agr_class>  ::= <class_id> [ ( <arg_list> ) ]
<aggregation>    ::= aggregation of <agr_class_list>
```

Asociación. En la declaración de una clase asociada se debe especificar la clase que se desea agrupar y con qué criterio.

```
<group_item>         ::= <identification_id> | <class_id>
<association>        ::= association of <class_id>
                    group by <group_item>
```

Especialización. Para la especialización se establece la lista de clases a especializar, separada por comas, y opcionalmente una fórmula del tipo 1 que deben cumplir las instancias de la clase especializada.

```
<specialization>     ::= specialization of <class_id_list>
                    [ where <bas_wff> ]
```

Generalización. Para definir una generalización se identifica la lista de clases a generalizar. Si además se quiere establecer que una instancia de la clase general sólo pueda pertenecer a una de sus descendientes, se utiliza la palabra clave disjoint.

```
<generalization>     ::= [ disjoint ] generalization of <class_id_list>
```

Dominios. Los dominios permiten definir tipos de datos básicos. En este sección se especifican las funciones del dominio y las ecuaciones que éstas deben cumplir. Posteriormente se indica la sintaxis completa de un dominio.

La declaración de las funciones consiste en un identificador, una lista opcional de parámetros y el tipo de la función.

```
<funct_decl>    ::= <funct_id> [ ( <var_id_list> ) ] : <type> ;
```

Las ecuaciones son igualdades entre dos términos, donde cada término puede ser una constante, una variable o una función del dominio, que a su vez puede tener como argumentos una lista de términos.

```
<eq_term>   ::= <const>
            | <var_id>
            | <funct_id> [ ( <eq_term_list> ) ]
<eq>        ::= <eq_term> = <eq_term> ;
```

B.4 Unidades de la Especificación

Una vez presentados todos los enunciados que forman las clases, y los dominios, se presenta el esquema general de cada una.

En las clases elementales son obligatorias las partes de identificación, eventos privados y procesos. Para las clases complejas los enunciados obligatorios y opcionales varían según el tipo que sea.

```
<class>     ::= class <class_id>
            <identification_st>
            [ <constant_atr_st> ]
            [ <variable_atr_st> ]
            [ <derived_atr_st> ]
            [ <class_atr_st> ]
            <private_events_st>
            [ <shared_events_st> ]
            [ <constraints_st> ]
            [ <valuation_st> ]
            [ <derivation_st> ]
            [ <preconditions_st> ]
            [ <triggers_st> ]
            [ <transactions_st> ]
            <processes_st>
            end class

            | complex class <class_id> <complex_st_items>
            [ <identification_st> ]
            [ <constant_atr_st> ]
```

```
[ <variable_atr_st> ]
[ <derived_atr_st> ]
[ <class_atr_st> ]
[ <private_events_st> ]
[ <shared_events_st> ]
[ <constraints_st> ]
[ <valuation_st> ]
[ <derivation_st> ]
[ <preconditions_st> ]
[ < triggers_st> ]
[ <transactions_st> ]
[ <processes_st> ]
end class

| domain <domain_id> [ ( <domain_param_id_list> ) ]
[ import <domain_id_list> ]
functions [ <variables> ]
    <funct_decl_items>
equations [ <variables> ]
    <eq_items>
 end domain
```

B.5 Esquema Conceptual

Finalmente, en el esquema conceptual se distinguen tres partes:

1. Una sección domains en la que se especifican los dominios predefinidos usados.
2. Una lista de especificaciones de clases y dominios.
3. Una sección opcional de interacciones globales para la definición de transacciones entre clases.

```
<espec> ::= conceptual schema <conceptual_schema_id>
            [ domains <predef_type_list> ]
            <class>
            [ global interactions
                <transactions_st>
                end global interactions
            ]
            end conceptual schema
```

C. El Componente Generador de Código

En este apéndice se muestra el componente que implementa el proceso automático de generación de código del modelo de ejecución presentado en esta tesis.

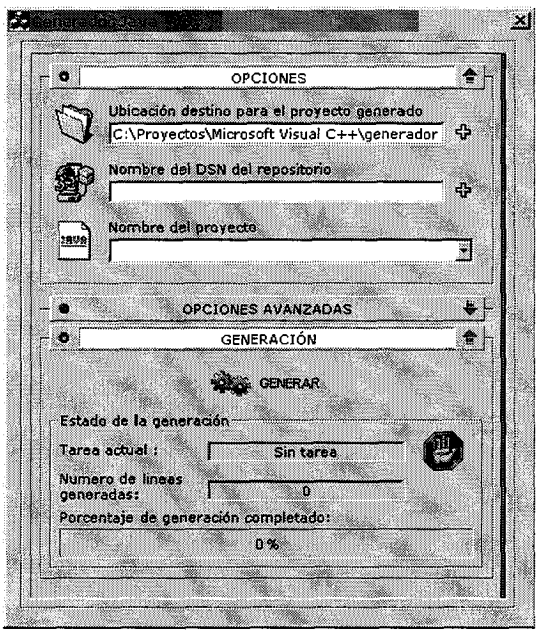


Figura C.1. *Componente Generador de Código.*

La implementación del generador de código se basa en un único componente activeX diseñado en Visual Basic 6.0 y compilado con la misma versión como proyecto de Control ActiveX.

Este componente puede ser reutilizado en el contexto de una herramienta CASE para, una vez capturado un modelo conceptual,

iniciar el proceso de generación de código. La apariencia gráfica del componente se muestra en la Figura C.1.

El componente contiene 3 menús etiquetados como OPCIONES, OPCIONES AVANZADAS y GENERACIÓN que permiten inicializar ciertas propiedades del proceso de generación.

En el primer menú, se establece el directorio destino donde se desea generar, el nombre del repositorio OASIS (DSN) que contiene la especificación del modelo conceptual del cual se desea generar código y el nombre del proyecto de aplicación.

Los símbolos + a la derecha de las cajas de texto de éste primer menú abren un cuadro de diálogo que facilita la introducción de esta información. Por ejemplo, el símbolo + que figura a la derecha de la caja de texto de especificación del repositorio abriría la ventana de la Figura C.2:

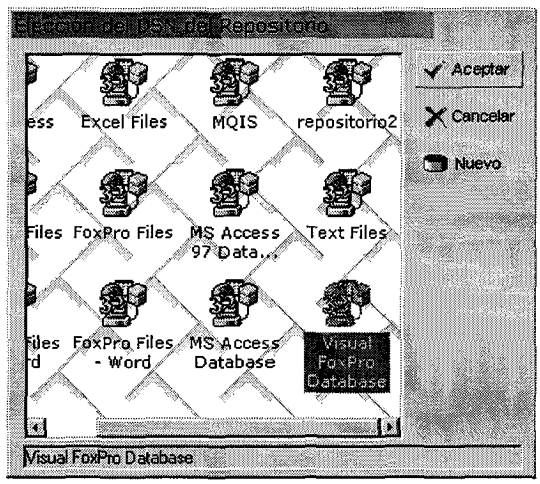


Figura C.2. *Configuración del mediador.*

Dónde se seleccionaría el repositorio OASIS correspondiente.

El menú de OPCIONES AVANZADAS tiene la apariencia gráfica que se presenta en la Figura C.3:

En este diálogo se especifica el nombre del paquete destino que contendrá el código generado y el nombre del mediador del reposi-

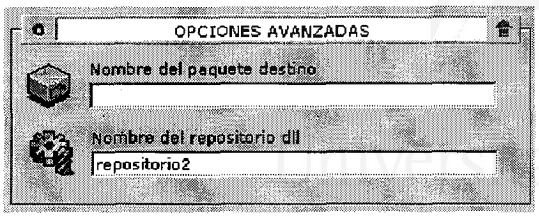


Figura C.3. *Opciones avanzadas.*

torio (en este caso en formato de librería dll) que permite efectuar el proceso de lectura de la especificación conceptual sobre el repositorio (almacenado sobre una base de datos relacional). La introducción de estos parámetros es opcional y por defecto tanto el nombre del mediador del repositorio como el nombre del paquete destino tienen un valor.

Por último, el menú GENERACIÓN permite observar el estado de proceso de generación cuando éste se inicia al pulsar el botón generar. En este sentido, la información que se proporciona es el número de líneas que se han generado hasta el momento, la tarea que se está realizando (lectura del repositorio, generación de estructura, generación de comportamiento y generación de especificaciones de interface de los componentes). Una barra de progreso indica el porcentaje de código generado.

Una vez que el código ha sido generado, el propio componente inicia el proceso de compilación en el lenguaje destino para obtener los componentes software que posteriormente serán publicados en el sistema JEM de acuerdo a la estrategia presentada.