Tecnologías orientadas a objetos

Tarea No. 3

Tomás Balderas Contreras
tbaldera@mail.cs.buap.mx
Coordinación de Ciencias de la Computación
Instituto Nacional de Astrofísica, Óptica y Electrónica

2 de octubre, 2002

Contenido

1 Características del Diseño Dirigido por Responsabilidades **1**

2 Técnica de Modelado de Objetos (OMT) **1**

3 Resumen de artículos **3**

3.1 Introducción 3

3.2 Objetivos y requerimientos 3

3.3 Método de generación de código 5

3.3.1 Entidades 5

3.3.2 Asociaciones 6

Índice de figuras

1 El proceso de desarrollo en OMT. 2

2 Generación de jerarquías de clases. 6

3 Generación de código para asociaciones. 8

1 Características del Diseño Dirigido por Responsabilidades

Esta metodología hace un gran énfasis en establecer el *comportamiento* de un sistema. El comportamiento puede ser definido casi desde el momento de la concepción de una idea, y puede ser expresado fácilmente tanto al cliente como al resto de los desarrolladores. Las características más importantes son las siguientes:

- puede ser empleado cuando se dispone de especificaciones ambigüas e incompletas,
- fácil integración con varios aspectos del desarrollo de software,
- los desarrolladores emplean escenarios para determinar el comportamiento del sistema,
- define diferentes responsabilidades para cada clase (componente),
- identifica y asigna colaboradores con los que cada clase debe interactuar para cumplir con las responsabilidades asignadas,
- es adecuado durante el proceso de desarrollo de sistemas grandes.

2 Técnica de Modelado de Objetos (OMT)

Existen varias técnicas para análisis y diseño de sistemas orientados a objetos. Una de las técnicas más conocidas es OMT (Object Modeling Technique) desarrollada por James Rumbaugh. OMT consiste de tres fases: análisis, diseño y diseño de objetos. La figura 1 ilustra estas fases.

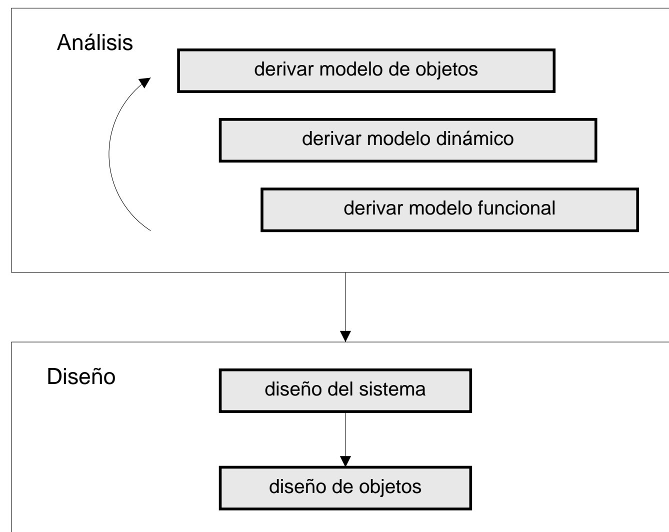

Figura 1: El proceso de desarrollo en OMT.

La etapa de análisis involucra comprender y modelar la aplicación, así como el dominio en el cual opera. La salida consiste de tres modelos:

- El *modelo de objetos* describe los aspectos referentes a la estructura estática del sistema. Este modelo se obtiene después de identificar los objetos presentes, las relaciones entre ellos y sus atributos. El resultado se documenta mediante un diagrama de clases UML y un diccionario de datos[1].

- El *modelo dinámico* describe los aspectos de control y de comportamiento del sistema. Este modelo se deriva de los escenarios típicos de interacción con el sistema. Los eventos entre objetos identificados se muestran mediante diagramas de secuencia y de colaboración[2].

- El *modelo funcional* describe los aspectos funcionales del sistema. El modelo funcional se expresa en un diagrama de flujo de datos.

Los modelos de la fase de análisis no son desarrollados de una sola vez. El análisis es un proceso iterativo y se considera completo cuando todos los requerimientos han sido capturados en los modelos.

Las fases de diseño en OMT no generan nuevos modelos. En la primera etapa están involucrados los siguientes criterios para el diseño de la arquitectura del sistema: la elección del estilo arquitectural, la descomposición del sistema en subsistemas, detalles del sistema operativo y de la plataforma y otros relacionados. El diseño de objetos involucra aspectos algorítmicos o de implantación de los componentes individuales.

Como otros métodos de desarrollo orientado a objetos, OMT ha evolucionado desde la publicación, en 1991, de *Object-Oriented Modeling and Design*. La experiencia adquirida con el

[1] En la notación propia de OMT se documenta mediante un *diagrama de objetos*.
[2] Diagramas de *seguimiento de objetos* y de *flujo de eventos* en la notación de OMT.

empleo del método ha fomentado un gran número de cambios y extensiones, entre ellos el empleo de la notación UML y los cambios drásticos en el modelo funcional.

Las características más notables de OMT son:

- Un rico conjunto de notaciones. Los diagramas de objetos en OMT emplean un vocabulario muy grande para expresar información referente a los objetos y sus relaciones.
- La etapa de análisis ofrece un varias heurísticas para obtener colecciones de objetos y las relaciones entre ellos.

Las dos fases de diseño en OMT están orientadas a la implantación y, como resultado, el paso entre el análisis y el diseño puede ser muy grande.

3 Resumen de artículos

Esta sección expone el contenido del artículo *Mapping UML Designs to Java* escrito por William Harrison, Charles Barton y Mukund Raghavachari [1]. El propósito de este resumen es proporcionar un panorama general del artículo y, por lo tanto, se han omitido comentarios sobre algunas secciones sin abandonar la esencia del documento. Los detalles pueden ser consultados en la referencia en extenso.

3.1 Introducción

Debido a la gran complejidad de los sistemas de software actuales, los desarrolladores han enfocado su atención en herramientas y lenguajes de modelado como auxiliares en el proceso de construcción del software. Uno de estos lenguajes de modelado es UML (*Unified Modeling Language*), que ha tenido buena aceptación porque proporciona desde la descripción de la implantación del sistema hasta el modelo de la arquitectura y de los componentes del mismo, independiente del lenguaje de implantación. Cuando un desarrollador diseña un modelo pensando en una implantación específica, esta persona está obligada a emplear el subconjunto o la interpretación del lenguaje de modelado que se adaptan a las restricciones de tal implantación; por lo tanto el diseño estará delineado por la elección del lenguaje o el estilo de programación.

El artículo presenta un método para generar código fuente a partir de una descripción de modelos. Este enfoque es importante porque, entre otras razones, permite al desarrollador mantener la consistencia entre el modelo del sistema y su implantación. El método propuesto genera el esqueleto del código del sistema desde la especificación de modelos abstractos que hacen muy pocas suposiciones sobre la implantación subyacente. Los autores concentran su atención en la generación de código en lenguaje Java y toman como base la notación UML, además argumentan que su método es aplicable a un conjunto amplio de lenguajes de programación y a cualquier notación de modelado.

3.2 Objetivos y requerimientos

El objetivo del método de generación de código es facilitar el proceso de diseño y desarrollo en los siguientes aspectos:

1. Proporcionar a los diseñadores flexibilidad para modelar sistemas considerando pocos detalles sobre la implantación del mismo,
2. Dotar a los diseñadores con una estructura consistente y de alto nivel basada en el diseño que soporte el uso y extensión del sistema sin exponer detalles de su programación o la de sus componentes.
3. Llevar a cabo la implantación de un componente del diseño de forma independiente a la de los otros elementos.
4. Minimizar el esfuerzo de diseño e implantación.

Los requerimientos del método de mapeo, o proceso de traducción, establecidos y que ayudarán a conseguir el objetivo anterior son los siguientes:

Separar el diseño de la implantación: La implantación de un elemento puede depender del diseño de otro componente pero no de su implantación. El código generado a partir del diseño debe emplear un estilo de programación que permita aislar los aspectos del diseño de las desiciones referentes a la implementación.

Separar el comportamiento de la representación: El código que implementa el comportamiento de los objetos no necesita ser modificado cuando la forma de representar sus datos cambie debido a desiciones de diseño motivadas por aspectos de rendimiento u otros. Es necesario separar el comportamiento de las entidades de su representación o implantación definiendo interfaces de alto nivel para manipular los atributos de las entidades y las asociaciones.

Maximizar la seguridad de los tipos de datos: El código generado debe conservar información referente al tipo de los datos para asistir al compilador en la detección de errores. El objetivo es preservar la fuerte comprobación de tipos y evitar código propenso a errores y el esfuerzo posterior invertido en la modificación de los tipos de datos.

Evitar la pérdida de código: Diseñar mecanismos para identificar y separar código añadido por el programador, de manera que no se pierda cuando el modelo de partida sufra modificaciones y nuevo código deba ser generado[3].

Apoyar el diseño basado en asuntos: Aceptar un conjunto de diagramas UML relacionados, pero que modelan aspectos y características separadas, y generar el código que habría resultado de una composición de varios diagramas.

Permitir herencia múltiple: La complejidad de ciertos sistemas hace necesario que ciertos componentes tengan más de una generalización, que es equivalente a herencia múltiple en la implantación. El proceso de generación de código debe liberar a los diseñadores de las restricciones impuestas por los lenguajes de programación que no soportan relaciones de herencia múltiple.

[3]El término original empleado en el artículo para este escenario es “round-trip”.

Implantar asociaciones: Los diagramas de clase de UML permiten expresar relaciones de asociación entre clases. El método de mapeo debe considerar estas relaciones y generar código para representar roles, multiplicidad, navegabilidad, direccionalidad, clases de asociación, estereotipos, etc.

Unificar estilos: El estilo de codificación del código generado debe ser consistente durante el proceso de mapeo de todas las entidades a partir del modelo. Un buen conjunto de estilos facilita el desarrollo de código, fomenta el reuso y la extensión de código y permite deducir inmediatamente como navegar y accesar los componentes del modelo.

Fomentar reusabilidad de código: Es conveniente y necesario expresar la estructura del código generado en términos de construcciones definidas en paquetes o librerias disponibles para el lenguaje de implantación.

3.3 Método de generación de código

El enfoque descrito en el artículo genera código orientado a objetos que satisface los requerimientos descritos anteriormente. Los modelos de partida son diagramas de clases cuyos elementos están marcados por el estereotipo ≪entity≫ (denotados como *entidades* de ahora en adelante) junto con sus atributos, operaciones, relaciones de generalización y relaciones de asociación, las cuales a su vez incluyen varios anexos como nombres de roles (*rolenames*), multiplicidades, agregados (*aggregations*) y clases de asociación. El método considera a una colección de diagramas UML como una representación particionada de un diagrama compuesto. Además supone que las clases que comparten el mismo nombre en diferentes diagramas representan a la misma entidad y sus componentes son mezclados en una sola.

3.3.1 Entidades

El método genera código para entidades de la siguiente manera. Por cada entidad **X** en el diagrama de clases UML original el proceso genera una interfaz *X* y dos clases, una clase abstracta que implanta la interfaz *X* llamada *XAbst* y una subclase concreta de esta llamada *XInst*. La interfaz declara los siguientes elementos:

- las operaciones definidas por la entidad en el diagrama de clases UML,
- operaciones auxiliares que accesan a los atributos y a las asociaciones existentes, además de operaciones que llevan a cabo la creación de objetos.

La clase *XAbst* implementa todas las operaciones auxiliares mencionadas anteriormente y define la forma exacta en que los atributos son implantados. La clase *XInst* extiende a *XAbst* y proporciona el esqueleto del código de la entidad, el cual es extendido para implantar las operaciones definidas en la entidad **X** de los diagramas y declaradas en la interfaz *X*. No está permitido modificar la interfaz *X* o la clase *XAbst* generada, además de que no debe ser necesario. El comportamiento está implantado y encapsulado en la clase *XInst*, no es necesario generar otra vez esta clase después de realizar cambios en los modelos originales que no tengan relación con el comportamiento de *XInst*.

Cada attributo **attr** definido en la entidad **X** tiene asociadas dos operaciones en la interfaz correspondiente (*X*) llamadas *getAttr()* y *setAttr()*. Estas operaciones son implantadas en la clase *XAbst* y están ocultas para otras clases en el sistema. Mediante este principio el proceso de mapeo debe cumplir con el requisito de separación del comportamiento de la representación de datos.

La jerarquía de clases definida en un diagrama UML se conserva después del mapeo. La interfaz de la entidad **Student** extiende a la interfaz de su generalización, la entidad **Universitymember**. La clase concreta *StudentInst* extiende a la clase *UniversityMemberInst* de forma indirecta a través de la clase *StudentAbst*.

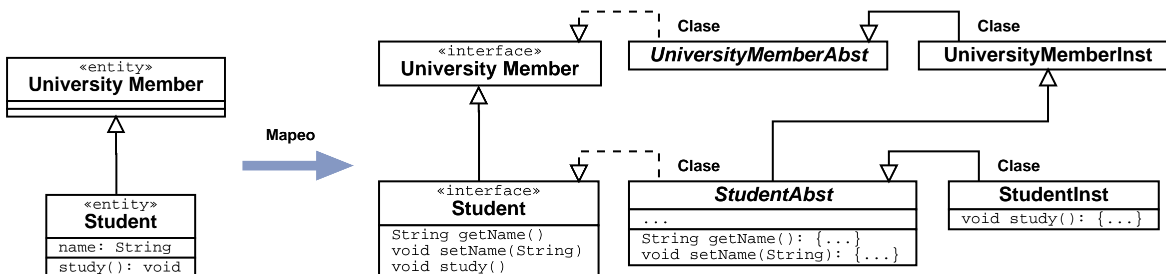

Figura 2: Generación de jerarquías de clases.

La figura 2 ilustra un ejemplo de la aplicación del proceso. La entidad **UniversityMember** es una generalización de **Student**. El proceso de mapeo crea la interfaz *Student*, la clase abstracta *StudentAbst*, la clase *StudentInst*, etc. Además de la operación **study** definida en **Student**, su interfaz contiene operaciones auxiliares get/set asociadas al atributo **name**, las cuales están implementadas en la clase *StudentAbst*. Un esqueleto de **study** se genera en la clase *StudentInst*. La figura omite los detalles referentes a la generación de código para relaciones de asociación.

3.3.2 Asociaciones

UML permite incluir relaciones de asociación a los diagramas de clase, las cuales permiten a las entidades conocer detalles sobre otras [2]. El proceso de generación de código separa la semántica de las asociaciones de su representación e implantación. Los autores introducen en su método una abstracción que encapsula la complejidad de recorrer y actualizar la asociación: el *cursor*.

Las ideas se comprenden mejor mediante el ejemplo ilustrado en la figura 3, que muestra las interfaces generadas a partir de las entidades **Student**, **Department** y **EnrollmentInfo** además del cursor para el rol **belongsTo**. A cada uno de los roles accesibles desde **Student** corresponde una operación auxiliar en la interfaz *Student* que recupera el cursor correspondiente, la implantación de estas operaciones es responsabilidad de la clase *StudentAbst* de la figura 2. Además, para cada rol debe ser generada su interfaz y la clase que la implanta, la figura 3 muestra la interfaz *StudentBelongs ToCursor* y la clase *StudentBelongs ToCursorImpl*. Para este caso la operación *belongsTo()* aplicada sobre una instancia de *StudentInst* regresa un cursor que

permite acceder a un conjunto de vínculos **belongTo** desde dicha instancia.

Cuando el cursor de una asociación se encuentra en un estado válido hace referencia a una instancia de la clase especificada en el otro extremo de la asociación en el diagrama de clases original (en este caso a una instancia de **Department**). La clase *StudentBelongs ToCursorImpl* implanta métodos para insertar y eliminar elementos en una lista de vínculos y para recorrer dicha lista. El diagrama de la figura 3 indica que la interfaz *StudentBelongsTo* extiende a la interfaz *Department*, por lo tanto es posible utilizar el cursor como una referencia a una instancia de **Department**. Este ejemplo define también la clase de asociación **EnrollmentInfo**, en este caso el cursor permite manipular también instancias de esa clase, pues la interfaz del cursor extiende a la interfaz *EnrollmentInfo*.

Referencias

[1] Harrison, W., C. Barton y M. Raghavachari. 2000. Mapping UML Designs to Java. En *Proceedings of the conference on Object-Oriented Programming, Systems, Languages and Applications*. OOPSLA'00. 178–187.

[2] Object Management Group, Inc. 2001. *OMG-Unified Modeling Language Specification.* Versión 1.4.
www.omg.org

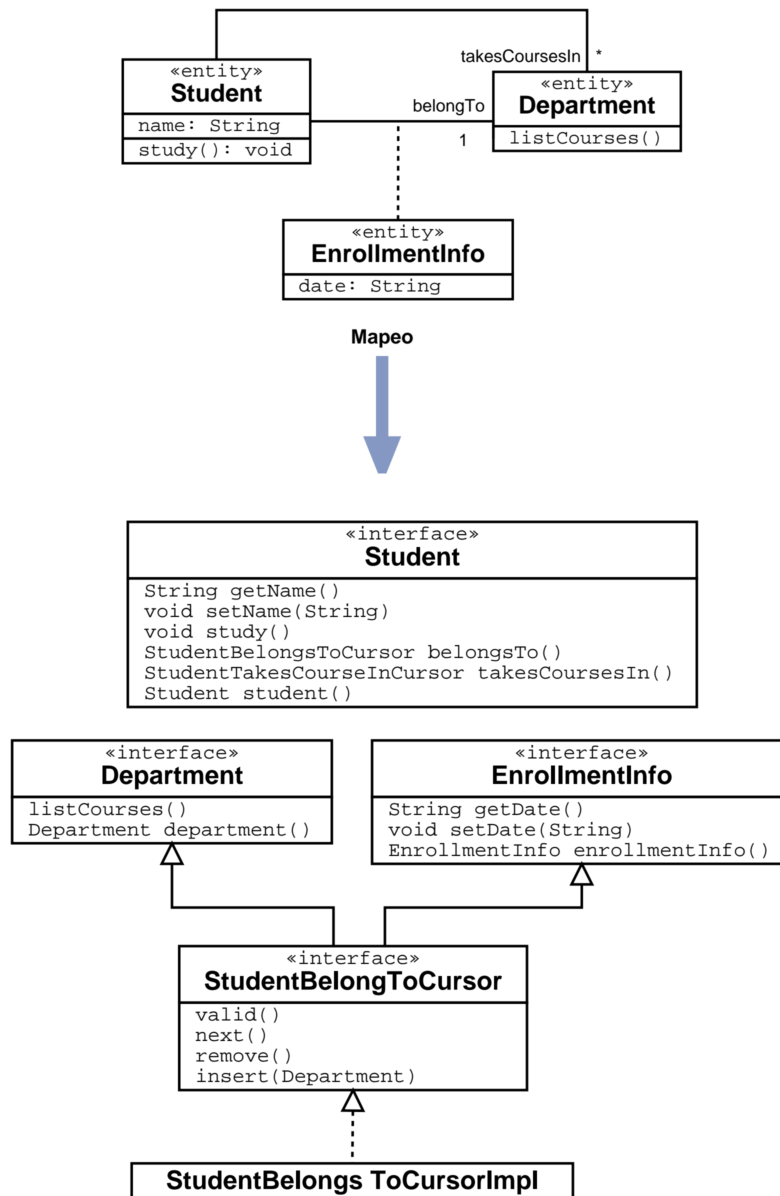

Figura 3: Generación de código para asociaciones.