PRÁCTICA 6: YO NO AGUANTO ESTE SIN DIOS[1]

1. Introducción

Las asociaciones UML 2.0, aunque puedan parecer simples e inocentes, contienen una importante carga semántica que hace que su implementación no sea tan trivial o directa como en principio podía parecer. Este fenómeno es habitual en los lenguajes de modelado software, ya que su objetivo es el de ofrecer un conjunto de potentes abstracciones con las que modelar determinados aspectos de un producto software y no crear una especificación detallada de cada aspecto de la implementación de un producto software.

Por desgracia, la mayoría de los desarrolladores desconocen la semántica de las asociaciones UML 2.0. Consecuentemente, una gran mayoría de los modelos de dominio existentes o están mal creados o están mal implementados. Este desconocimiento se justifica, en parte, por el hecho de que esta semántica ha ido variando a través de las diferentes versiones de UML, lo que ha contribuido a generar una cierta confusión entre los desarrolladores.

Sin embargo, las asociaciones UML, y en especial a partir de la versión del estándar 2.0, resultan de gran utilidad para definir de manera sencilla relaciones entre clases, así como ciertas restricciones sobre dichas relaciones.

Por ejemplo, las asociaciones bidireccionales permiten especificar de forma muy sencilla que dos referencias entre objetos de unas determinadas clases deben estar ligadas. Es decir, si un objeto X de una clase A está relacionado con, es decir tiene una referencia a, un objeto Y de una clase B, entonces el objeto Y debe estar relacionado con el objeto X. Si el valor de las referencias de X o Y cambiase, el valor de la referencia asociada también debería cambiar. Es decir, cambios en un objeto afectan a otros objetos asociados.

No obstante, a la hora de implementar un modelo de dominio debemos asegurar que estas restricciones asociadas se mantengan a nivel de código. Es decir, utilizando la terminología del *diseño por contrato*, debemos preservar todos los *invariantes* definidos implícitamente en el modelo de dominio a nivel de implementación.

El objetivo general de esta práctica es que el alumno interiorice la semántica de las asociaciones UML 2.0. Para ello implementará diversas asociaciones UML 2.0 de un modelo de dominio dado. Además, para la propagación de los cambios requeridos para mantener la integridad de las asociaciones bidireccionales utilizará el patrón *Observer*.

La siguiente sección describe los objetivos de esta práctica de manera más detallada.

[1] En homenaje a la escena final (http://youtu.be/5nDTRvYRPl0) de "*Amanece que no es poco*" (José Luis Cuerda, 1989)

2. Objetivos

Los objetivos concretos de esta práctica son:

(1) Comprender el funcionamiento de las asociaciones en UML 2.0, y en especial de las bidireccionales.
(2) Comprender el funcionamiento del *Patrón Observador*.
(3) Profundizar en el concepto de *invariante*.
(4) Aprender a trabajar con *precondiciones*.
(5) Ser capaz de implementar asociaciones bidireccionales en UML 2.0 utilizando el patrón observador.

Para alcanzar dichos objetivos, el alumno deberá realizar de forma satisfactoria una serie de actividades relacionadas con el problema que se describe a continuación.

3. Gestor de Matrimonios Heterosexuales y Monógamos

El diseño de la Figura 1 representa un modelo conceptual de datos de una pequeña aplicación para la gestión de matrimonios con hijos en Narnia. Son matrimonios monógamos, por lo que cada persona puede estar casada a lo sumo con otra persona. Además, por simplicidad, son matrimonios heterosexuales[2], por lo que siempre debe haber un hombre que ejerza el rol de padre y una mujer que ejerza el rol de madre. Además, por la lógica implícita al concepto de matrimonio, si una persona está casada con otra persona, ésta última debe estar también casada con la primera.

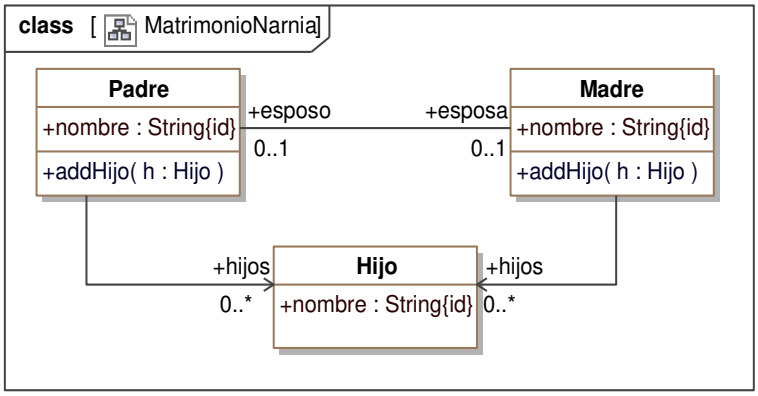

Figura 1. Modelo de Dominio

Según la legislación vigente en Narnia, sólo se pueden tener hijos dentro del matrimonio. Ello significa que el método *addHijo* sólo se puede invocar sobre instancias de la clase *Padre* o *Madre* que se encuentren felizmente casadas.

Los hijos, por ley de Narnia, son de ambos miembros de un matrimonio. Por tanto, los hijos de la esposa de un padre, han de ser hijos de dicho padre. De igual forma, los hijos del esposo de una madre han de ser hijos también de dicha madre.

[2] No obstante, Narnia, al igual que cualquier otra civilización avanzada respeta y reconoce los derechos de la comunidad homosexual.

Dichas restricciones no se encuentran reflejada en el diseño de la Figura 1, por lo que habría que especificarlas de forma externa mediante un lenguaje apropiado para tal propósito. En el caso de UML, dicho lenguaje es OCL (*Object Constraint Language*). Utilizando OCL definiríamos dos invariantes, uno para la clase *Padre* y otro para la clase *Madre*. La Figura 2 muestra los invariantes descritos expresados en OCL.

```
01 context Padre:
02 invariant padreHijos('Mis hijos lo son también de su madre') :
03  not esposa.oclIsUndefined() implies (self.hijos = self.esposa.hijos);
04
05 context Madre:
06 invariant madreHijos('Mis hijos lo son también de su padre'):
07  not esposo.oclIsUndefined() implies (self.hijos = self.esposo.hijos);
```

Figura 2. Invariantes para del diseño de la Figura 1 destinados a satisfacer las leyes de Narnia

Por desgracia, la gente se divorcia. En caso, la custodia de los hijos pasa a manos del estado, por lo que dejan de estar ligados tanto a su padre como a su madre. Los habitantes de Narnia afortunadamente son eternos, por lo que no se consideran casos de viudedad.

Las bodas y los divorcios se realizan a través de la asignación de valores a las correspondientes propiedades. La Figura 3 muestra ejemplos de boda, divorcio normal y divorcio con segundas nupcias.

```
// Ejemplo de boda
Padre romeo   = new Padre("Romeo");
Madre julieta = new Madre("Julieta");
romeo.Esposa  = julieta;

// Ejemplo de divorcio normal, donde ambos cónyuges pasan a estar solteros.
Madre belen    = new Madre("Belén Esteban");
Padre jesulin  = new Padre("Jesulín de Ubrique");
belen.Esposo   = jesulin;
jesulin.Esposa = null;

// Ejemplo de divorcio tormentoso, ya que uno de los cónyuges se divorcia
// para contraer segundas nupcias con otra persona.
Madre patricia = new Madre("Patricia LLosa");
Padre mario = new Padre("Mario Vargas LLosa");
Madre isabel = new Madre("Isabel Preysler");
patricia.Esposo = mario;
mario.Esposa = isabel;
```

Figura 3. Ejemplos de bodas y divorcios

4. Actividades

El alumno, para alcanzar los objetivos perseguidos, deberá completar las siguientes actividades:

1. Crear una estructura de clases, sin comportamiento, que implemente el modelo de dominio de la Figura 1. Los atributos y referencias entre clases se implementarán como propiedades de C#, y no como pares de *getters* y *setters*.
2. Implementar la restricción de integridad definida por la asociación bidireccional existente entre *Padre* y *Madre* mediante el uso de observadores. No está permitido añadir ninguna operación extra a la interfaz pública de las clases *Padre* o *Madre*, salvo las que correspondan al uso de observadores.
3. Preservar los invariantes de la Figura 2 mediante la utilización de observadores que detecten cambios en la colección hijos de la clase *Padre* o *Madre*.

5. Criterios de Evaluación y Aclaraciones

La practicará se entregará a través de la plataforma Moodle siguiendo las instrucciones en ella proporcionadas.

La práctica se calificará atendiendo a los siguientes criterios de evaluación:

(1) Correcta implementación de los aspectos estructurales (definición de atributos) del diseño (1 punto).
(2) Cumplimiento de la Guía de Buenas Prácticas en Programación (1 punto).

Para el observador u observadores creados para la gestión de la integridad de la asociación bidireccional entre las clases *Padre* y *Madre*:

(3) Correcta definición de la interfaz o interfaces de observación (2 puntos).
(4) Correcta definición de los métodos para gestionar la incorporación y eliminación de suscriptores a las clases observadas (1 punto).
(5) Correcta implementación de la lógica de los métodos de observación (1.5 puntos).
(6) Correcta implementación de las notificaciones a observadores (0.5 puntos).

Para el observador u observadores creados para la gestión del invariante de la Figura 2:

(7) Correcta definición de la interfaz o interfaces de observación (2 puntos).
(8) Correcta definición de los métodos para gestionar la incorporación y eliminación de suscriptores a las clases observadas (1 punto).
(9) Correcta implementación de la lógica de los métodos de observación (1.5 puntos).
(10) Correcta implementación de las notificaciones a observadores (0.5 puntos).

Con objeto de facilitar el trabajo del alumno, estará disponible a través de la plataforma un programa de pruebas que permita al alumno verificar la corrección funcional de la implementación creada. No obstante, merece la pena aclarar que el que una práctica supere todos los casos de prueba no significa que ésta sea necesariamente correcta, ya los observadores podrían estar deficientemente implementados, o incluso no implementados.

Pablo Sánchez Barreiro