Generación de código Java a partir de clases UML

A lo largo de estos ejemplos, trataremos de mostrar las diferencias que existen a nivel de codificación (es decir, pasar de un diagrama de clases UML a código fuente, para facilidad del lector solo consideramos el lenguaje de programación Java, aunque se pueden utilizar otros como C++, C#, Pyhton, PHP, entre otros).

Una clase

Inicializamos con una simple clase. Una clase se describe por sus tres compartimentos: nombre, atributos y métodos.

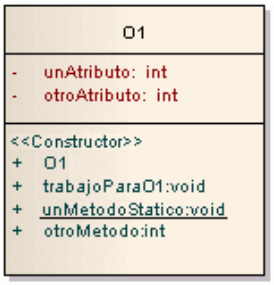


Figura 1: Una clase en UML.

UML1.java

Asociación entre clases

Consideramos ahora, una primera asociación «dirigida» entre la clase O1 y la clase O2. Esta relación de asociación en el sentido de O1 hacia O2 (si no tuviera la flecha, la asociación sería considerada bidireccional), demanda, que en el código de la clase O1, se envíe un mensaje hacia O2, como se muestra a continuación.

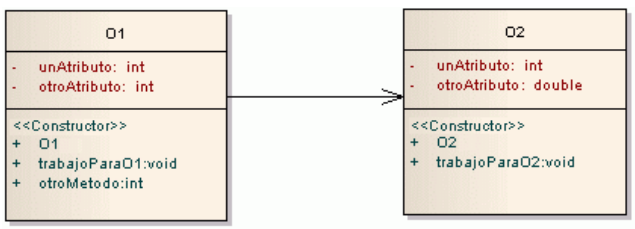


Figura 2: Un asociación dirigida.

UML2.java

Composición y agregación

Este ejemplo nos permite diferenciar los mecanismos de vida y muerte de los objetos que resultan de los diferentes tipos de relaciones, en este ejemplo, la clase O1 estará vinculada a la clase O3 mediante un vínculo de agregación, y a la clase O2 mediante un vínculo de composición (el rombo es vacío para la agregación y relleno para la composición). Es importante tener cuidado la colocación del rombo del lado de la clase contenedora y de lo que no es contenedora.

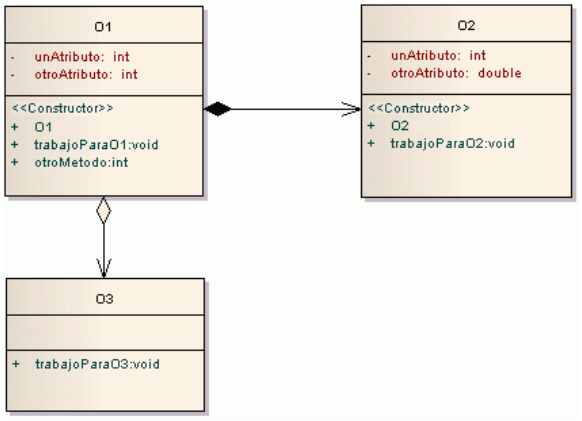


Figura 3: Composición y agregación.

En el caso de Java, proponemos dos versiones del código que realiza este tipo de relación de composición. En el primer código (UML3.java), el objeto O2 es un componente de O1, por la sencilla razón de que sólo existe en el interior de O1. De hecho, se construye en el constructor de O1, con el resultado de que la única referencia de este objeto O2 es un atributo de O1. Con el fin de ilustrar esta dependencia extrema entre el objeto O1 y el objeto O2, utilizamos un método llamado finaliza().

Recordemos que el "destructor" permite, justo antes de la eliminación de un objeto, asegurarse de que los recursos utilizados, únicamente a partir de este objeto, serán igualmente liberados. El destructor no recibe argumentos y es de no retorno. En la práctica, se podría tratar de una conexión a un archivo (el método se encargará por lo tanto de liberar el acceso a este archivo), o de una conexión a la red, por lo tanto, el método podría interrumpir, después de haber eliminado el único objeto conectado a la red. En este caso, lo utilizaremos, solo para indicar en qué momento se elimina el objeto.
El siguiente código crea un conjunto de objetos en varias ocasiones, para que el recolector de basura se invoque de forma automática, y recupere un cierto número de estos objetos que están inutilizables. A diferencia de los códigos mostrados anteriormente, no hay aquí una llamada explícita al recolector de basura. Preferimos utilizarla como es en la mayoría de los casos, es decir, la decisión sólo se hace cuando sea necesario, cuando la memoria comienza a saturarse. Simplemente forzaremos su uso mediante el uso de un ciclo. El resultado de la simulación de los dos códigos Java se

indica a continuación. Sólo mostramos una parte, dado que el ciclo se ejecuta hasta 10,000.

UML3.java

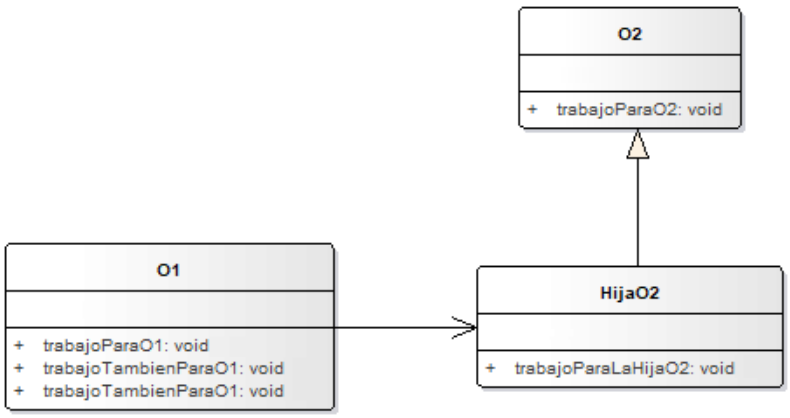
Podemos constatar, en el resultado de la simulación, en la asignación de las referencias de los objetos O1 a nulo, el recolector de basura hace su trabajo, como se esperaba, al deshacerse gradualmente, y sin saberlo, de los objetos O1 llegar a ser incómodo. Tengamos en cuenta que esta asignación de nulo no tiene la obligación de desaparecer el objeto, ya que la misma referencia unObjetoO1 se comparte por todos los objetos O1, y que cada uno de estos objetos se referirá al tiempo de una iteración del ciclo.

Al final de esta iteración, que será entregado al recolector de basura. Pero lo que más nos preocupa aquí es la eliminación de los objetos de O2, mientras que en ninguna parte del código lo hicimos explícitamente. El objeto O1, que desaparece, conduce igualmente a la desaparición de la única referencia al objeto de O2, lo que permite al recolector de basura también actuar. Sin embargo, observamos que los objetos O3, que solo se agregan, sobreviven hasta la conclusión del programa.

UML3bis.java

El contenido de este código presenta una forma aún más radical de hacer que los objetos de O2 sean completamente dependientes de los objetos de O1. En el código UML3bis.java, la clase O2 se declara y crea dentro de la clase O1. La clase O2 se convierte en interna a O1. Este mecanismo, de uso bastante raro en Java y C#, porque bastante sutil, hace posible solidarizar fuertemente las clases O1 y O2. Tras escribir esto, solo la clase O1 podrá tener la clase O2. El enlace de composición entre objetos se refuerza extendiéndose al nivel de clase. En todos sus métodos, la clase O2 podrá usar todo lo que caracteriza a la clase O1 y viceversa.

Herencia y asociación dirigida



En el diagrama UML anterior, y en el código que lo traduce en Java, la clase O1 puede dirigirse a la clase HijaO2 como HijaO2 o como O2. Finalmente, tiene la posibilidad de enviar dos mensajes a la clase HijaO2: trabajoParaO2() o trabajoParaLaHijaO2(). Además, la clase O1, en otro de sus métodos, recibe un argumento de tipo «HijaO2». Esto nos permite ilustrar el principio de sustitución, que dice que podemos llamar a este método, pasando indistintamente un argumento del tipo «HijaO2» o un argumento del tipo «O2».

Herencia1.java

La diferencia esencial en este código de Java radica en el hecho de que el primer método trabajoTambienParaO1(), codificado para recibir un argumento de superclase de tipo O2, se llamará ahora con un argumento de tipo subclase, sin que esto plantee ningún problema (dado que se hace casting implícito). Ten en cuenta también que la sobrecarga de un método por el simple hecho de que uno de los argumentos es de la subclase de un argumento del método de sobrecarga no plantea ningún problema porque, en el tiempo de ejecución, es fácil elegir el método correcto para ejecutar según el tipo estático del argumento que pasamos en el método. Si es un argumento de superclase, solo puede ser el método provisto para este propósito. Por otro lado, si es un argumento de tipo subclase y si existe, uno elegirá ejecutar el método previsto para este propósito. De lo contrario, podemos recurrir al método que se supone debe ejecutarse en un objeto de tipo superclase y cuya subclase se hereda de todos modos.