DS - Apuntes - Tema 1

Autor: Jose Luis Martínez Ortiz

Índice

1. Introducción de UML **2**
1.1. Diagrama de Clases 2
1.1.1. Objetos 2
1.1.2. Relaciones 3

2. Patrones **6**

1. Introducción de UML

1.1. Diagrama de Clases

Un diagrama de clases UML contiene una serie de elementos que detallaremos a continuación, pero podemos distinguirlos claramente en dos tipo, Objetos y relaciones. Los objetos son una representación de un ente que queremos representar en el módelo UML, como por ejemplo un *empleado* en un sistema software de una empresa o un *vehículo*. Las relaciones se pueden dar entre dos objetos, estos dos objetos pueden ser el mismo pero es un caso un poco especial.

1.1.1. Objetos

Los objetos los identificaremos como clases y siguen el esquema de la figura 1, donde podemos observar una una caja con 4 filas.

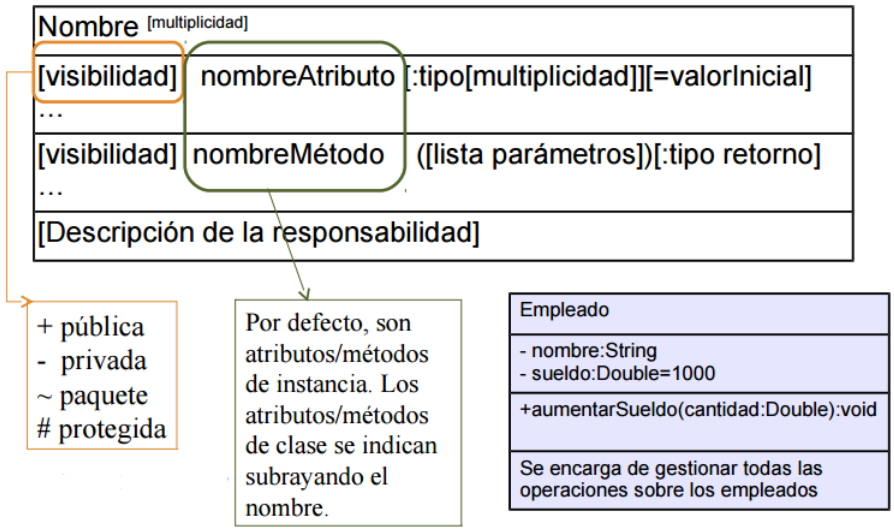

Figura 1: Esquema de una clase uml

En la primera fila se indica el nombre del objeto que se va a modelar, así como su multiplicidad (singleton, interface, abstracta, etc).

En la segunda y tercera fila se indican los atributos y métodos respectivamente, que definen al objeto, para ello indicamos:

1. **visibilidad :** indica cual es la "privacidad" del atributo.
   - $+$: significa que el atributo o método es público y puede ser visto y utilizado por todo el mundo que tenga acceso a al objeto.
   - **-** : significa que el atributo o método es privado y que solo el propio objeto puede verlo y utilizado.
   - $\sim$: significa que el atributo o método es de paquete y solo los objetos que pertenecen al mismo paquete pueden acceder al el, es decir, es público dentro del paquete al que pertenece y es privado para todos los demás.
   - $\#$: significa que el atributo o método es protegido y solo el mismo y sus descendientes pueden verlo y utilizarlo.
2. **nombre** : del atributo o del método. Puede ser de clase si van subrayados. Los métodos/atributos de instancia están asociados a los objetos cuya invocación se realiza mediante envío de mensajes entre objetos y los métodos/atributos de clase Son métodos asociados a la clase que se pueden invocar sin necesidad de crear ninguna instancia (ningún objeto). Por ejemplo en java se definen con la palabra "`static`"

3. **Tipo** : clase a la que pertenece ese atributo.
4. **lista de parámetros** : conjunto de nombre_atributo:tipo.
5. **multiplicidad de atributo** : número de elementos en dicho atributo.
6. **valor inicial** : indica el valor por defecto de dicho atributo.
7. **tipo retorno** : indica el tipo de objeto que devuelve.

En la cuarta fila se hace una breve descripción de la responsabilidad de dicho objeto. Principalmente se utilizan los tipos de clases: Normal, Abstracta e Interfaz.

Abstracción
Una clase abstracta es una clase que no se puede instanciar y se usa únicamente para definir subclases. Las clases abstractas siempre tiene que tener al menos un método sin implementar. Se deben de utilizar para implementar algunos métodos pero dejando otros para las subclases. El nombre de la clase aparece en cursiva. Los métodos abstractos (no implementados) aparecen en cursiva. Una clase abstracta debe ser heredada por subclases. La cabecera de un método de una clase abstracta debe coincidir en las subclases que lo heredan. La visibilidad puede modificarse en las subclases siempre que sea a mayor. También el tipo de retorno, que puede ser un subtipo en las subclases.

Interfaz
Básicamente es una clase abstracta pero que todos los métodos son abstractos, es decir que o se implementan todos los métodos o ninguno. Su utilización también es para la de indicar unas "reglas" que debe de cumplir una clase.

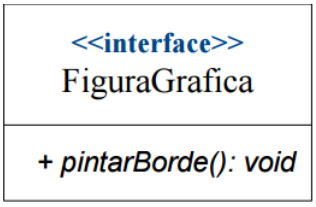

Figura 2: Ejemplo una clase interfaz.

1.1.2. Relaciones

Las relaciones entre objetos pueden ser de muchísimas formas y con múltiples significados para poder modelar todas las opciones que se plantean a la hora de diseñar un software. Se van a describir a continuación las principales relaciones entre dos objetos.

Vamos a definir también los elementos de las asociaciones y un ejemplo:

- **Nombre:** Nombre de la asociación. Se puede indicar hacia dónde se lee empleando el símbolo ▶.
- **Rol:** Se emplea para explicitar el papel que juega cada clase en la asociación.
- **Navegabilidad:** Representa si el conocimiento entre los objetos de los extremos de la asociación es unidireccional (→ o ←) o bidireccional (sin punta de flecha en ninguno de los extremos).
- **Multiplicidad:** Indica el número de objetos de una clase que pueden asociarse con un objeto de la otra clase. Su sintaxis es `valorMínimo..valorMáximo`, pudiendo usarse * para especificar "varios" y así no dar un valor máximo concreto. Si no se indica nada, por defecto es 1.

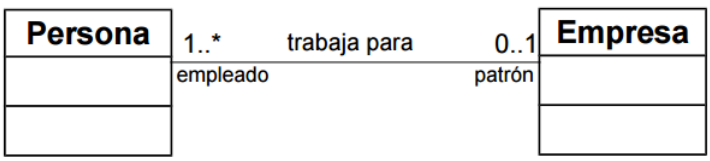

Figura 3: Ejemplo de los elementos de una relación.

Dependencia

Modelan una relación débil y poco duradera en el tiempo, se representa con una fecha con una linea de puntos hacia el objeto que necesita. Significa que en algún momento se requerirá al objeto relacionado para momentos muy concretos, siendo la vida de esta relación normalmente de ámbito de método. En la figura 4

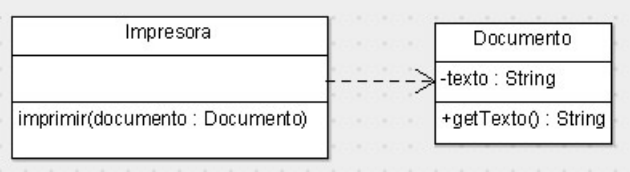

Figura 4: Ejemplo de una relación de dependencia.

Asociación

Modela una relación estructural fuerte y duradera en el tiempo (figura 5).

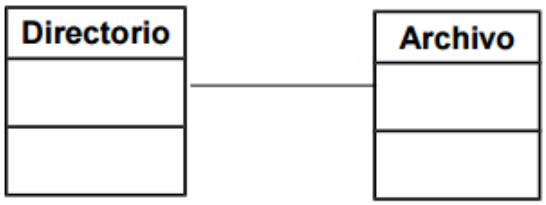

Figura 5: Ejemplo de una relación de asociacion.

Dentro de la asociación encontramos también dos tipos especiales de relaciones que nutren a esta relación de más significado, como es la "Agregación" y la "Composición".

La *Agregación* es una asociación cuyo nombre es "parte de", en la que una de las clases representa el "todo" y la otra la/s "parte/s", es decir, ambos objetos tiene sentido por si mismos pero uno de ellos es parte del otro. Se representa con un rombo blanco en el extremo de la relación.

Figura 6: Ejemplo de una relación de asociacion - agregación.

La *Composición* es como una agregación fuerte, en que la/s parte/s no tiene/n sentido sin el todo. Se representa como un rombo negro en el extremo de la relación. Se puede decir que ambas partes están incompletas sin la otra.

Figura 7: Ejemplo de una relación de asociacion - composición.

Herencia

La herencia en UML se representa como una relación entre dos clases y se denota con un triangulo al final

de la relación. En las subclases, sólo se indican las variables y los métodos nuevos que declaran. También se indican los métodos que, aunque figuren en la superclase, se redefinen en la subclase.

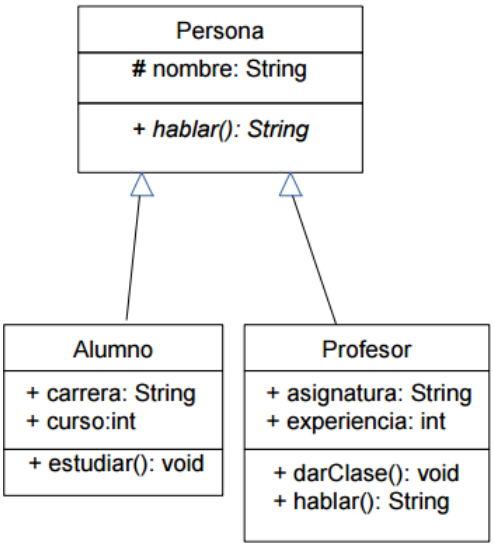

Figura 8: Ejemplo de una relación de Herencia.

La relación de Herencia también se define en dos subtipos, generalización y especialización. En la generalización se identifican rasgos comunes entre varios tipos de entidad y se crea una superclase para todos ellos. Por ejemplo (figura 9) los tipos de entidad COCHE y CAMIÓN de la izquierda se generalizan a la derecha en la superclase VEHÍCULO con los atributos comunes. Se está realizando un refinamiento bottom-up o ascendente.

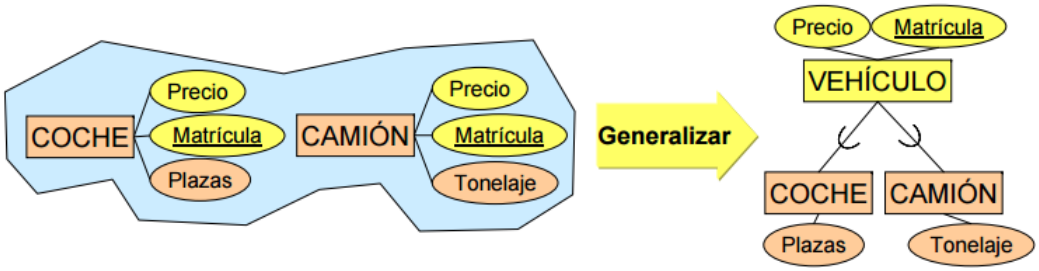

Figura 9: Ejemplo de una relación de Herencia por Generalización.

Es el proceso de dividir un tipo de entidad en subclases. Es lo contrario a generalizar: aquí el refinamiento es top-down o descendente. Un conjunto de subclases se define a partir de alguna característica distintiva. Podemos tener varias especializaciones sobre el mismo tipo de entidad:
– Por tipo de trabajo: Secretario, Técnico, Gerente
– Por método de pago: Asalariado, Por horas

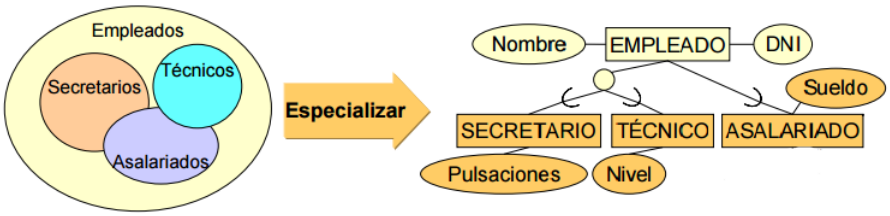

Figura 10: Ejemplo de una relación de Herencia por Especialización.

Realización
La realización es una relación que indica que la clase va a realizar una implementación de la clase interfaz

a la que apunta. Se simboliza como una linea discontinua acabada en un triangulo. La clase que tiene una realización sobre la interfaz esta obligada a implementar los métodos de esta.

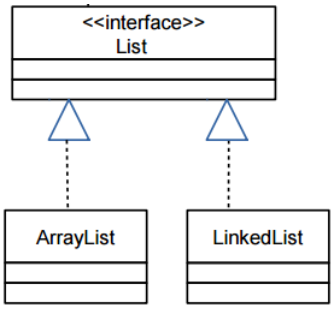

Figura 11: Ejemplo de una relación de realización.

2. Patrones

Un componente software tiene que ser una parte funcional del software, siempre tiene que utilizarse de la misma forma. Los marcos (*Framework*) contiene uno o múltiples componentes software adaptados para resolver un problema muy concreto. Un framework es una infraestructura mínima que provee un esqueleto genérico para un conjunto de abstracciones software con un mismo contexto. Las diferencias de un framework y un patrón son que el patrón esta a un nivel de abstracción mayor, además el framework es mucho más grande y especializado que un patrón. Por ello un framework suele contener varios patrones de diseño, pero al revés no tiene ningún sentido.

Un patrón permite capturar el conocimiento más importante acerca del diseño de la solución a un problema, que permite su posterior reutilización.

En general los patrones se clasifican en tres apartados:

- *Creacionales*: instanciación de objetos reforzando la restricciones en el tipo y número.
- *Estructurales*: organización en la integración de clases de objetos.
- *Comportamentales*: asignación de responsabilidades y comunicación entre los objetos.

Según Christopher Alexander, "Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema, de tal modo que se pueda aplicar esta solución un millón de veces, sin hacer lo mismo dos veces". Los patrones tienen cuatro elementos principales:

1. Un *nombre* significativo en una o dos palabras.
2. El *problema*, que describe cuándo aplicar dicho patrón explica el contexto y el problema en sí.
3. Generan una *solución* a modo de plantilla que puede aplicarse al problema para resolverlo.
4. Las *consecuencias* de aplicar dicho patrón, ya que generan ventajas e inconvenientes. Valores a tener en cuenta a la hora de elegir el patrón.

Los patrones pueden ser de tres tipos: Patrones de Diseño Orientado a Objetos, Patrones arquitectónicos y patrones de Datos. Nosotros nos centramos en los primeros, los Patrones de Diseño Orientado a Objetos. Para estos patrones hay que determinar una serie de atributos:

- **Granularidad**, una granulalidad alta permite un patrón muy funcional pero tiene un un rendimiento muy bajo. En cambio una granularidad baja tiene menos funcionalidad pero consigue un mayor rendimiento.
- **Interfaz**, los patrones ayudan a definir una interfaz indicando que elementos clave y los tipos de los datos debe contener y cuales no, aplicando una serie de restricciones.
- **Jerarquía**, también nos ayuda en la elección de a jerarquía de clases y relaciones, indicando cuando es mejor una clase por *abstracción* o *interfaz*.

La descripción de un patrón contiene los siguientes apartados, aunque no son todos ellos obligatorios siempre que se recoja los cuatro elementos que describimos anteriormente.

Plantilla	
Contexto	describe el entorno en el que se ubica el problema incluyendo el dominio de aplicación
Problema	una o dos frases que explican lo que se pretende resolver
Fuerzas	lista el *sistema de fuerzas* que afectan a la manera en que ha de resolverse el problema incluye las limitaciones y restricciones que han de respetarse
Solución	proporciona una descripción detallada de la solución propuesta para el problema
Intención	describe el patrón y lo que hace
Anti-patrones	"soluciones" que no funcionan en el contexto o que son peores; suelen ser errores cometidos por principiantes
Patrones relacionados	referencias cruzadas relacionadas con los patrones de diseño
Referencias	reconocimientos a aquellos desarrolladores que desarrollaron o inspiraron el patrón que se propone

Según nos define [Bruegge and Dutoit, 2004] hay unos tareas a la hora de diseñar software:

- Desarrollar una jerarquía de entidades de análisis del problema.
- Determinar existencia de un lenguaje de patrones aplicable.
- Determinar los patrones arquitectónicos candidatos.
- Utilizar la colaboración entre patrones de más bajo nivel.
- Determinación de patrones de diseño candidatos.
- Buscar los patrones de diseño adecuados.
- Independientemente del nivel de abstracción, compararlo con otras soluciones pre-existentes.

2.1. Abstracción-Caso