**PROGRAMACIÓN ORIENTADA A OBJETOS**

Distintiva de la estructurada debido a la existencia de herencia, objetos, etc. Lenguaje UML (ESTÁNDAR).

**CONCEPTOS POO**

**ATRIBUTOS**

Características de cada objeto instanciado (variables)

Una clase es un esqueleto que define cómo va a ser un objeto. Podemos crear indefinidos objetos procedentes de esa clase.

**Abstracción**: Tenemos acceso a los métodos públicos, pero no al código.

**Encapsulamiento**: Se ocultan los detalles de un objeto.

**Modularidad**: Permite modular las distintas partes del programa creando funciones o algoritmos. Se evita la redundancia (no hace falta realizar el mismo algoritmo una y otra vez secuencialmente)

**Herencia**: Permite extender una clase, heredando el comportamiento y atributos de la clase extendida.

**Polimorfismo**: Tener una función con el mismo nombre, pero con distinto comportamiento

**UML**

Lenguaje gráfico (de diagramas). Distintos tipos de diagramas:

* Estática
* Dinámica

Dentro de estos los más usados son:

1. **Diagrama de Clases:** Representa de qué se componen las clases y sus relaciones.
2. **Diagrama de Objetos:** Representa los objetos con sus relaciones a lo largo del sistema (evolución del objeto).
3. **Diagrama de Paquetes:** Visualiza los distintos paquetes con sus clases (y relaciones) respectivas que tenemos en nuestro proyecto.
4. **Diagrama de Despliegue:** Detalla el cómo se va a desplegar el software con el hardware y cómo se ejecutará.
5. **Diagrama de Casos de Uso:** Ver el comportamiento del programa con los distintos actores y acciones.
6. **Diagramas de Secuencia:** Similar al anterior, pero se observa la secuencia que toma el programa.
7. **Diagramas de Estado:** Muestra los cambios de estado de los objetos.
8. **Diagramas de Actividad:** Muestra las actividades que se realizan, no el estado.

**EN PROFUNDIDAD**

**DIAGRAMA DE CLASES**

Diagrama principal a la hora de desarrollar un programa POO. Se tienen que distinguir (estándares) los distintos atributos y relaciones.

Las clases pueden tener distinta visibilidad:

* **Package**: Por defecto. Esta clase tiene alcance de paquete.
* **Public**: Cualquier otra clase puede acceder a ella (si está fuera del paquete se tiene que importar).
* **Final:** NO puede heredar.
* **Abstract:** NO se instancia, pero es un guion (indica la estructura). Se tiene que heredar.
* **Private:** Sus métodos son de alcance de clase
* **Protected:** Como package pero es visible desde cualquier clase heredada (incluso de otro paquete).

CONVENIOS DE DIBUJO

* **Una clase se dibuja con un rectángulo**
  + **Primera parte:** Se pone el nombre
  + **Segunda parte:** Se ponen los atributos y si tienen una visibilidad especial, se pone.
  + **Tercera parte:** Se ponen los métodos

|  |
| --- |
| **TRIÁNGULO** |
| +base: float  +altura: float  +anguloA: float  +anguloB: float |
| +GetArea(): float  +GetPerimetro(): float  +GetBase(): float  +SetAltura(altura: float)**: void**  … |

void se sobreentiende, no hace falta ponerlo.

**MÉTODOS**

**Constructor**: Cada instancia llama a este método. Se pueden tener distintos constructores dependiendo de los valores que le pasemos.

**Firma de un método**: Formada por el nombre, argumentos que recibe y tipo de estos.

VISIBILIDAD DE ATRIBUTOS Y MÉTODOS

* **Public:** Se representa con +.
* **Private:** Se representa con -.
* **Protected:** Se representa con #.
* **Package:** Se representa con ~.
* **Static:** Existe antes de crear alguna instancia de la clase y se puede invocar sin necesidad de crear una instancia de dicha clase. Por ello no puede usar atributos de la misma clase que no sean estáticos.

**RELACIONES**

|  |
| --- |
| FIGURA GEOMÉTRICA |
|  |
| GetArea(){GetAreaTriangulo();GetAreaCuadrado()…} |

Tipos de relaciones:

* **Asociación:** Se asocian una serie de atributos de distintas clases (puede ser unidireccional o bidireccional).
* **Dependencia:** Una clase necesita otra clase para definir sus métodos (Como por ejemplo la tabla de arriba. El método GetArea() requiere de la existencia de los métodos de las otras clases).
* **Clase Asociación:** Tipo de clase especial. La propia asociación conlleva algún tipo de información en la relación.

|  |  |
| --- | --- |
| **CLASE MATEMATICAS** | **CLASE RESOLVER** |
| **A: IN**  **B: IN**  **C: OUT** | **A: IN**  **B: IN**  **C: OUT** |
| **+Suma(a: float, b: float): float**  **+Producto(a: float, b: float): float**  **+Dividir(a: float, b: float): float** | **-Suma(a: float, b: float): float (Depende de los métodos de MATEMATICAS)**  **-Dividir(a: float, b: float): float (Depende de los métodos de MATEMATICAS)**  **+Resolver(a: float, b: float, c: float): float** |

* **Composición:** Una clase está compuesta de otras clases (objetos de otras clases). Una cama está compuesta por una clase de **Patas**, otra de **Colchon** y otra de **Somier.** La app obtendrá el precio de cada uno de los elementos y los juntará creando una instancia nueva de **Cama**.
* **Agregación:** Similar a la anterior pero los objetos pueden tener sentido independiente. Es decir, antes, las patas sin la cama no tienen sentido, pero la cafetería de una escuela sí tiene sentido de manera independiente (siempre que sea pública).
* **Herencia:** IMPORTANTE. Diferenciación de POO respecto a los demás paradigmas. Clases heredan métodos, atributos de otra clase además de los suyos propios.
* **Realización:** Muy parecida a la anterior, pero en este caso la clase origen es abstracta o interfaz.

**ESTEREOTIPOS**

* **Boundary**: Se busca analizar la relación o la interfaz que hay entre el sistema y el entorno. Ej: A un sensor de temperatura le pongo una interfaz de usuario.
* **Control**: Modela la funcionalidad interna del sistema. Cómo se puede hacer el control de ciertos movimientos. System.out.println es un ejemplo.
* **Entity**: IMPORTANTE. Nos da el estado interno de nuestro sistema o desarrollo.

**HERRAMIENTAS DE DESARROLLO UML**

* **Eclipse (UML Designer plugin)**
* **NetBeans (PlantUML plugin)**
* **ArgoUML**
* **Visual Paradigm**

**DIAGRAMAS DE CASOS DE USOS**

Muy utilizado en reuniones normalmente para mostrar el cómo se comportará el programa en distintos ambientes/usuarios. Un ejemplo son los distintos departamentos de una empresa: RRHH, dirección, etc… RRHH no tiene por qué ver la información personal de un empleado, pero sí su nómina, así como el empleado también puede ver sus nóminas, pero tampoco los datos personales del empleado de RRHH.

Identificar los actores primero, es decir, quiénes van a utilizar nuestra aplicación y conocer a qué partes tienen o no acceso permitido. Tener en cuenta que un mismo actor tenga distintos accesos.

Todo esto se tiene que documentar de manera extensa.

**TIPOS DE RELACIONES**

* **Asociación**: Relación de un actor con un caso de uso. Se representa con una línea (-----)
* **Extensión:** Extiende/Hereda algunas de las funcionalidades de otro caso de uso. Se representa con una línea discontinua y encima la palabra **extend** (- - -).
* **Generalización**: Herencia. Un caso de uso hereda todas las propiedades del caso de uso del cual parte y puede añadir nuevas o modificarlas. Se representa con una línea continua terminada en triángulo (---->).
* **Inclusión**: Se incluye uno o varios comportamientos de otro caso de uso. Se representa como la de asociación, pero poniendo **include**.

**DIAGRAMAS DE SECUENCIA**

Parten de los diagramas de casos de uso.

Existen mensajes síncronos, representados con una flecha rellena.

**DIAGRAMAS DE ESTRUCTURA**

*Subrayados los ya vistos*

* **Diagramas de clase**
* **Diagramas de objetos**
* **Diagramas de paquetes**
* **Diagramas de estructuras compuesto**
* **Diagramas de componentes**
* **Diagramas de despliegue**
* **Diagramas de perfil**

**DIAGRAMAS DE COMPORTAMIENTO**

* **Diagramas de interacción**
  + **De secuencia**
  + **De resumen de interacción**
  + **De comunicación**
  + **De tiempo**
* **Diagramas de actividad**
* **Diagramas de casos de uso**
* **Diagramas de [máquinas de] estado**

**DIAGRAMAS DE COMUNICACIÓN**

Información que van entre los distintos elementos de un desarrollo.

**DIAGRAMAS DE ESTADO**

Secuencia de tiempo como en los de secuencia, pero se muestra los distintos estados de nuestra máquina o aplicación.