LPS: Ingeniería del Software Orientada a Objetos

Federico Peinado
www.federicopeinado.es

Depto. de Ingeniería del Software e Inteligencia Artificial
disia.fdi.ucm.es

Facultad de Informática
www.fdi.ucm.es

Universidad Complutense de Madrid
www.ucm.es

Ingeniería del Software

- Métodos para desarrollar software en serie
  - Planificación
  - Análisis
  - Especificación
  - Diseño
  - Codificación
  - Documentación
  - Pruebas
  - Implantación
  - Mantenimiento
  - Reutilización

Orientación a Objetos

- No sólo es un paradigma de programación
  - Programación Orientada a Objetos
- Es una filosofía que afecta a *buena parte del proceso* de desarrollo de software
  - Diseño Orientado a Objetos
- Además de la naturalidad que se obtiene al reflejar objetos "reales" en objetos software, el DOO incorpora técnicas de programación que fomentan la reutilización y facilitan la ampliación/adaptación de los programas

DOO versus DE

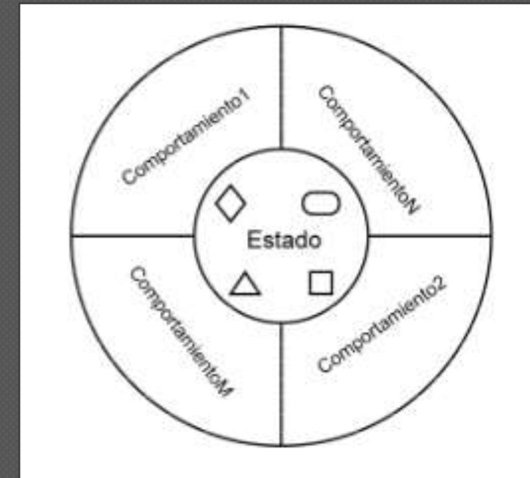
- Diseño Estructurado (DE) tipo Pascal o C
  - Datos y procedimientos están separados
  - Los procedimientos se pasan datos y operan sobre ellos
  - Los programas se dividen en módulos de procedimientos similares, que se usan entre sí y usan a los de otros módulos
- Diseño Orientado a Objetos (DOO) tipo Java o C++
  - Datos (= atributos) y procedimientos (= métodos) están juntos formando lo que llamamos "objetos"
  - Los objetos se comunican entre ellos, transmitiendo órdenes e información (incluso referencias a otros objetos)
  - Los programas se dividen en paquetes de clases (de objetos) similares, que se relacionan de varias formas entre sí y se relacionan con las de otros paquetes

Conceptos fundamentales

- Abstracción
  - *“Ocultar algunos aspectos de bajo nivel para que se muestren con mayor claridad otros de más alto nivel”*
  - Simplificar la realidad que queremos modelar para centrarnos en el comportamiento de los objetos software y no en la implementación de su código
- Encapsulación
  - *“Ocultar algunos aspectos internos para que se muestren con mayor claridad otros aspectos más externos”*
  - Restringir el acceso a los atributos y métodos de los objetos para centrarnos en el tipo de órdenes e información que se transmiten y no en su estructura y funcionamiento interno

Objetos

- Entidades software dinámicas responsables de una parte de la funcionalidad del programa
  - Combinan tanto "datos" como "procedimientos"
  - Se relacionan con otros objetos "heredando" su estado y sus comportamientos, o actuando como "clientes" de la funcionalidad que otros ofrecen
- Tienen identidad propia
  - Son siempre ejemplares de una cierta clase
  - Tienen un identificador único mediante el que otros objetos pueden referenciarlos
- Tienen un estado
  - Atributos: valor de los "datos" que contienen o referencias a otros objetos con los que comunicarse para dar órdenes, enviar o recibir información
- Tienen ciertos comportamientos
  - Métodos: "procedimientos" que es capaz de ejecutar pudiendo, según su estado, operar sobre sus atributos o delegar parte del trabajo aprovechándose de los comportamientos de otros objetos



Clases

- Entidades software que definen la estructura de un tipo concreto de objetos
  - Describe sus atributos y sus métodos
  - Es el punto de partida para crear nuevos objetos
- Relaciones entre clases
  - Generalización y su inversa (especialización)
    - Se realiza mediante la herencia entre clases
  - Composición (*parte de*) y su inversa (*todo*)
    - Se realiza mediante la inclusión de referencias a los objetos que son "partes" en los atributos del objeto que es "todo"
  - Asociación (*usa a*) y su inversa (*es usado por*)
    - También se realiza mediante inclusión de referencias a los objetos "usados" en los atributos del objeto que los "usa"

Comunicación entre objetos

- Proceso formal por el que un objeto proporciona ordenes a otro objeto
  - El objeto *cliente* debe <u>llamar</u> al método apropiado del objeto *receptor*, proporcionarle toda la información necesaria (*argumentos*) y recogiendo los *resultados* cuando los haya
  - El receptor debe realizar la funcionalidad esperada según la *semántica de la orden* dada, o delegar en un tercero dicha responsabilidad (y así sucesivamente…)
    - El método es un *bloque de código a <u>ejecutar</u>* desde el que se puede acceder a los atributos del objeto y modificar su valor, llamar a métodos del propio objeto o de otros objetos, etc.

Diseño Orientado a Objetos

1. Analizar los requisitos del problema
2. Identificar las clases de objetos necesarias
3. Organizar dichas clases
    1. Establecer relaciones de generalización (jerarquía de herencia de clases)
    2. Establecer relaciones de composición
    3. Establecer relaciones de asociación

Cohesión versus Acoplamiento

- Cohesión
  - *"Relación lógica entre los atributos y los métodos de una misma clase"*
  - Cuanto mayor es esta relación mejor se comprende la clase y mejor se mantiene el código del programa
- Acoplamiento
  - *"Relación lógica entre los atributos y los métodos de distintas clases (llamadas clases acopladas)"*
  - Cuanto menor es esta relación mejor se comprenden las clases por separado y mejor se mantiene el código del programa
    - Obviamente siempre debe existir cierto nivel de acoplamiento

Ideal del DOO:
Maximizar cohesión y minimizar acoplamiento

Clases minimalistas

- Filosofía de *"Divide y vencerás"*
  - Mejor intentarlo con muchas clases pequeñas que con pocas y muy grandes
  - En un programa "monolítico" compuesto de una sola clase o muy pocas clases grandes, no tiene mucho sentido hablar de encapsulación
- Cuanto más fácil de comprender es una clase, más fácil de usar, probar y reutilizar, y mejor se mantiene el código del programa

Ideal del DOO:
Una clase = Un concepto

Otros ideales del DOO

- Corrección
- Simplicidad
- Claridad
- Seguridad
- Escalabilidad
- Optimización
- … y en general todas las bondades que persigue la Ingeniería del Software

Las “tres gracias” del DOO

Polimorfismo

Encapsulación

Herencia

Encapsulación

Encapsulación

- Proteger la implementación interna de una clase tras una interfaz simple ayuda a minimizar el acoplamiento entre clases
  - En general *todos los atributos de una clase* deben declararse como privados
- Lo más importante es diseñar esta interfaz
  - Todo lo demás se puede diseñar más tarde
  - Si está mal diseñada, una vez se haya usado por otras clases será mucho más difícil arreglarlo todo

Ejemplos de encapsulación

```
// Opción A
class Punto {
  public int _x;
  public int _y;
}
```

- En la práctica (gracias al compilador) tiene el mismo coste en ejecución
- Todo lo que hace la opción A se puede hacer con la opción B
- De hecho la opción B es más flexible que la opción A (si se empieza usando la opción A, luego para cambiarlo -por ejemplo a la B- se debe cambiar todo el código que accede a **x** e **y**)

```
// Opción B
class Punto {
  private int _x;
  private int _y;

  public int leeX(){
    return _x;
  }
  public int leeY(){
    return _y;
  }
  public void ponX(int x){
    _x = x;
  }
  public void ponY(int y){
    _y = y;
  }
}
```

Ejemplos de encapsulación

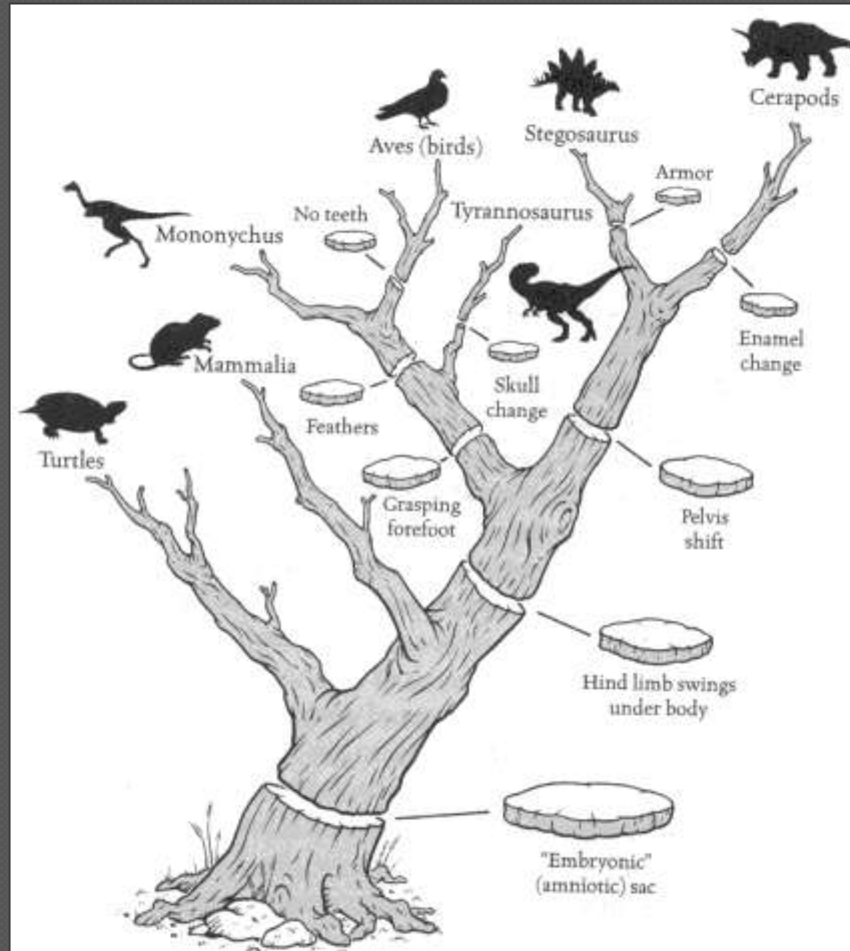
- ¡Ojo! Aquí estamos devolviendo manipuladores de los atributos privados
  - No estamos ocultando realmente la implementación (hay acoplamiento con la clase cliente)
  - No hay control sobre las precondiciones de la clase
  - El cliente se puede quedar con referencias a estos manipuladores y usarlos posteriormente cuando se hayan vuelto inválidos

```
class Circulo {

  private Punto _centro;
  private int _radio;

  public Punto usarCentro() {
    return _centro;
  }
}
```

Ideal de encapsulación

- La encapsulación indica la flexibilidad de una clase
  - ¿Cuanto código externo a esta clase deberíamos cambiar si cambiase la implementación de esta clase?
  - ¿Es posible reducir más el número de métodos públicos que ofrece la clase, para aumentar su protección?
  - …
- Filosofía: *"En vez de pedir a otro objeto la información que necesitas para hacer un trabajo, pídele a dicho objeto que haga el trabajo por ti"*
- Por defecto no ofrecer métodos que accedan o manipulen atributos privados de una clase (`get` y `set`)
  - Y si son necesarios, es preferible que estén en alguna interfaz que implemente la clase, para al menos desacoplar a otras clases de posibles cambios en la estructura interna de la clase

Herencia



Herencia

- Permite especializar un programa sin modificar el código existente, tan sólo añadiendo clases
- Favorece la reutilización al evitar que se repita código (heredado) entre clases
- Favorece la extensibilidad y la flexibilidad
  - Las subclases no sólo pueden ampliar el comportamiento de su superclase (haciendo que esta pueda usarse desde cualquier parte donde pueda usarse la subclase), sino que pueden restringirlo o redefinirlo totalmente

Subclase versus Subtipo

- Principio de Sustitución de Liskov (Liskov, 1988)
  - *"Si B es un subtipo de A entonces en cualquier situación se puede sustituir un ejemplar de A por otro de B obteniéndose el mismo comportamiento observable"*
- La herencia crea subclases, y si además se satisface este principio, crea subtipos
- Si no se satisface este principio, al hacer una referencia a algún objeto de A nos vemos obligados a tener presente la existencia de la clase B (más acoplamiento)

Herencia versus Composición

- La herencia impone una relación muy fuerte
  - El comportamiento heredado se fija de manera estática, de modo que no puede cambiarse en tiempo de ejecución
  - Un cambio en alguna de las superclases obliga a la revisión y modificación de todas las subclases en la jerarquía
- Si una relación entre dos clases se puede expresar de varias formas, debemos usar la relación más débil (menor acoplamiento)
  - La composición es más débil que la herencia
- La composición tiene otras ventajas
  - Mayor flexibilidad sin afectar de ninguna forma a los clientes
  - Clases más robustas y seguras
- Pero la herencia es necesaria para ciertas cosas
  - Acceso a miembros protegidos
  - Sobrecarga de métodos

Ideal de herencia

- Es recomendable hacer jerarquías de clases abstractas
  - Gracias a la herencia habrá varios grados de generalización, facilitando la reutilización
    - Creamos subclases que implementan el estado y el comportamiento de aquella superclase abstracta que queramos aprovechar
  - Las partes más inestables (la *implementación*) dependen de las más estables (los *métodos abstractos*)
  - Cambios localizados que no afectan a todo el programa
  - Diseños más flexibles y modulares
- En Java también es recomendable hacer jerarquías de herencia múltiple de interfaces

Polimorfismo

Polimorfismo

- Capacidad de distintas clases para responder a la misma llamada a método, de modo que cada una lo implementa *de distinta forma*
- Permite ocultar distintas implementaciones bajo una interfaz común y hacer que varíe el comportamiento del método según el objeto que ha recibido la llamada
  - El objeto cliente sólo sabe que el objeto receptor sabe responder a la orden dada
  - El objeto receptor es el responsable de ejecutar la orden adecuadamente
- Al igual que la herencia favorece la flexibilidad, al permitir modificar un comportamiento añadiendo nuevas clases, sin modificar las ya existentes

Tipos de polimorfismo

- *Ad hoc*: Sobrecarga de métodos (*overloading*)
  - Un mismo método tiene múltiples implementaciones, según los argumentos que reciba o el resultado que devuelva
  - Ocurre en Java y muchos otros lenguajes, en tiempo de compilación (*vinculación estática*)
- Por inclusión: Sobrescritura de métodos (*overriding*)
  - Un método que tanto en la superclase como en la subclase tiene los mismos argumentos
  - En Java el método de la superclase queda "redefinido" por el de la subclase en tiempo de ejecución (*vinculación dinámica*)
- Puro: Variables polimórficas
  - Variables(-referencia) declaradas según una clase o interfaz, cuyos valores(-objeto) luego se podrían crear como ejemplares de cualquier subclase o clase que implementa dicho interfaz
  - En Java esto ocurre a menudo, siempre por vinculación dinámica (salvo en *partes especiales* como en llamadas a métodos estáticos o privados, que no se pueden sobrescribir… o usos de atributos)

Ideal de polimorfismo

- Aprovechar las variables polimórficas de Java con su vinculación dinámica
  - ¡Ojo! Consultar el tipo de un objeto (mediante **`instanceof`**) para luego invocar el método específico de ese tipo no es hacer polimorfismo
- En general debemos evitar el uso innecesario de **`instanceof`** y cualquier otra distinción de casos (**`switch`**) por tipo de objeto, o *downcasting* (conversión de tipo a subtipo)

Críticas, dudas, sugerencias…

Federico Peinado
www.federicopeinado.es