Programación Orientada a Objetos

Laura M. Castro Souto

Primer Cuatrimestre
Curso 2000/2001

Índice de cuadros

Capítulo 1

Evolución de los lenguajes de programación

La programación ha ido evolucionando desde sus orígenes, donde los lenguajes eran de muy bajo nivel y las posibilidades reducidas, refinándose cada vez más hasta alcanzar el alto nivel y relativa sencillez actuales.

Veremos a grandes rasgos cómo se ha desarrollado este proceso.

1.1. Programación no estructurada

En sus primeros momentos, decimos que la programación llevada a cabo en lenguajes como el BASIC es *no estructurada*. Los programas eran una serie de líneas de código con saltos (instrucciónes tipo GOTO) y en un único bloque (sin posibilidad de subrutinas; BASIC cuenta únicamente con la instrucción GOSUB <DIRECCIÓN>).

1.2. Programación procedimental

Con la aparición de lenguajes como PASCAL, surge la posibilidad del empleo de *subrutinas* (funciones y procedimientos), lo que da nombre a esta "etapa" en que la programación se dice *procedimental*.

1.3. Programación estructurada

La necesidad de dar claridad y legibilidad al código provoca que se reconozcan una serie de **principios** o reglas a la hora de estructurar un programa. Surge así la *programación estructurada*, cuyas directivas son:

- Estructura jerárquica top-down.
- Diseño modular.
- Salidas y entradas claramente definidas.
- Único punto de entrada y único punto de salida (también aplicable a funciones).
- Constructores de programación estructurada: *secuencia*, *selección*, *iteración* y *llamadas a procedimientos*.

Existe una analogía entre este modo programación, esta forma de programar, propugnada por la programación estructurada y el llamado **diseño mediante refinamiento progresivo**. Se afronta el problema desde la "cima" y se va analizando de forma que se delimitan sucesiva y claramente las partes y funciones que realizarán cada una de las cuales, hasta llegar a las más sencillas, directamente implementables de una manera fácil.

1.4. Programación modular

El siguiente paso consistió en la división de los largos programas, compuestos de líneas y líneas de código en secuencia sin interrupción en diferentes módulos, en los que, por ejemplo, podían trabajar simultáneamente los diferentes integrantes de un grupo ó departamento.

Los *módulos*, sin embargo, no permiten la instanciación ni la exportación de tipos. Este iba a ser el siguiente peldaño en la escalera de la evolución de los lenguajes de programación.

1.5. TADs

Los *Tipos Abstractos de Datos* cuentan con una serie de características que exponemos a continuación:

- Exportan una definición de tipo.
- Hacen disponibles operaciones para manipular instancias del tipo (*interfaz*).
- El interfaz es el único mecanismo de acceso a la estructura del TAD.
- Es posible crear múltiples instancias del tipo (algo que no era posible en programación modular).

El segundo y tercer puntos son comunes a los módulos. Son los otros dos los que marcan la diferencia.

La filosofía consiste en que creamos nuevos tipos y los usamos como si fuesen predefinidos. Es el paso previo a la orientación a objetos.

1.6. Programación Orientada a Objetos

[1]

Clases y **objetos** serán para nosotros conceptos similares a *tipos* y *variables*. Es decir, la relación que existe entre un objeto y la clase a la que pertenece (la clase de la que es una *instancia*[2].) es análoga a la que hay entre una variable y el tipo del que es declarada.

Una **clase** es un *TAD* con una característica añadida: permite la *herencia*.

$$\text{CLASE} = \mathsf{TAD} + \mathsf{herencia}$$

[1] SIMULA 67 es el primer lenguaje que se puede considerar que soporta P.O.O.

[2] Realmente la palabra *instancia* no existe en español; debería usarse *ejemplo*, *ejemplificación* o *concreción*

Se define así un nuevo paradigma de programación, en el que podemos tener diferentes jerarquías y/o especializaciones, en forma de **subclases** y **superclases**, habilitando al mismo tiempo el **polimorfismo**.

El refinamiento progresivo usado hasta ahora era un refinamiento *orientado al flujo de datos*, que no favorece la reutilización de código. Con la programación orientada a objetos, en cambio, las clases más generales pueden servirnos siempre para nuevos usos que las requieran.

Los programas en P.O.O. manejan objetos que se comunican unos con otros mediante el intercambio de *mensajes*.

El diseño en P.O.O. no es ya, pues, top-down sino **bottom-up**; en lugar de ir de lo principal a lo específico se hace al contrario. De aquí la gran importancia de una correcta *identificación de los objetos* que vayan a participar en nuestro proyecto.

1.7. Resumen

El concepto de Orientación a Objetos surge en torno a 1970 de mano de Alan Ray. Xerox PARC fue una de los primeros interesados en lo que se quería imponer como un nuevo paradigma de programación.

El primer lenguaje totalmente orientado a objetos fue **Smalltalk**, creado por Dyrabook. El único "fallo" que le fue achacado fue la falta de una interface WIMP[3].

Parecía que esta nueva concepción iba a cercarse alrededor de la IA y sus redes semánticas (los objetos parecían adaptarse a sus directivas ES_PARTE_DE y ES_UN) y marcos (*frames*).

No obstante, pronto los demás lenguajes comenzaron a crear extensiones de sí mismos intentando incorporar las características que se definían fundamentales para la O.O.: B. Stroustrup, de los laboratorios Bell, construyó a partir de C un lenguaje C con clases, que más tarde seróa conocido como **C++**[4].

Pero no sólo C tuvo su extensión, también Lisp, con **CLOS** (COMMON LISP OBJECT SYSTEM) y muchos otros.

También se crearon lenguajes totalmente orientados a objetos como **Java**[5], que incorporó los hoy tan populares *applets*, el *garbage collector* y sobre todo su característica *multiplataforma* (bytecode).

Ahora mismo, está a punto de ver la luz la versión que Microsoft ha hecho del lenguaje Java, que se llamará **C#** (pronunciado *C sharp*). Este lenguaje compila el código fuente a un código interno (*Intermediate Language*, similar al bytecode) y posteriormente a código máquina.

Hoy en día los lenguajes O.O. tienen la supremacía en el campo de la RAD (RAPID APPLICATION DEVELOPMENT), con ejemplos como **Delphi**, aunque algunos de los que se usan no cumplen todos los requisitos (p. ej. **Visual Basic**), que se consideran sólo *basados en objetos*.

---

[3] Windows Icons Mice Pointers.

[4] Existe una anécdota sobre el nombre del lenguaje C++: C surge de la evolucíon de B, un lenguaje construido a partir de BCPL (BASIC CPL), a su vez "hijo" de CPL (COMBINED PROGRAMMING LANGUAJE). Después de tener un B y un C, se hicieron apuestas sobre si el siguiente en la saga sería 'D' (por orden alfabético) o 'P' (por BCPL). Stroustrup evitó el compromiso añadiendo el operador *sucesivo* a la letra C, C++.

[5] Los creadores de Java en principio le llamaron *Oak* (roble), pero como el nombre ya estaba registrado, le pusieron lo que en inglés americano significa coloquialmente "café". Además, sus componentes se denominan *Java Beans*, que no significa otra cosa que "granos de café".

Diferencia entre lenguajes *orientados a* y *basados en* objetos

Los **Lenguajes Basados en Objetos** poseen una sintaxis y una semántica que permiten la creación de objetos, pero no poseen todas las características necesarias para ser considerados plenamente orientados a objetos. La característica más comúnmente nosoportada suele ser la posibilidad de *herencia* (fallo de VB).

Dentro de los **Lenguajes Orientados a Objetos** se hace una distinción entre L.O.O. **puros** (como Smalltalk, Eiffel, Java,...), que aunque por ser más abstractos son más fáciles de manejar pero más lentos, no permiten otra forma de hacer las cosas salvo la correcta, mientras que los llamados L.O.O. **híbridos** (C++, Object Pascal, CLOS, Prolog++,...) mezclan la O.O. con otro paradigma (programación estructurada / funcional / lógica,...) y a pesar de poder ser más fáciles de aprender, si se conocen aquéllos de los que derivan, proclamarse más rápidos y eficientes, y ofrecer otro tipo de soluciones cuando la O.O. no es lo más adecuado, no ofrecen toda la versatilidad y potencia de una completa O.O.

Capítulo 2

Elementos básicos de la Orientación a Objetos

En un programa, tenemos *objetos* que se comunican entre sí, que tienen un determinado *estado interno*, que pertenecen a una *clase*,... Pero ¿qué significan todos estos términos? Eso es lo que definiremos en este tema.

2.1. Clases

Ya comentamos en el tema anterior que la relación CLASE - OBJETO era equiparable a la relación TIPO - VARIABLE (página 6).

A grandes rasgos, podemos decir que una **clase** es una *plantilla* que define cómo han de ser los objetos que pertenezcan a ella. Se definen una serie de métodos, entre ellos constructores, que permiten *instanciar* ó crear objetos de esa clase y manejarlos.

La definición formal de clase abarca dos aspectos:

- **Definición estructural:** se define el estado y comportamiento de los objetos de esa clase (**atributos** y **métodos**).
- **Propósito de creación de nuevos objetos:** la propia clase ha de definir métodos para crear sus objetos[1].

En **Java**:

```
[MODIFICADORES] class Nombre [extends clase]
                -----         -------
                             [implements interface,...]
                              ----------
{
 \\ Atributos

 \\ Métodos
}
```

[1]Estos métodos especiales se denominan *métodos constructores*, como ya hemos comentado.

Donde los modificadores pueden ser:

* **public** → clases que son accesibles desde fuera del paquete (módulo[2]) en que está definida.
* **abstract** → clases que no pueden instanciarse, sólo se definen para ser extendidas.
* **final** → clases que no pueden extenderse con subclases.

Los **interfaces** son clases especiales que pueden contener nombres de funciones pero nunca sus implementaciones. Si una clase *implementa* un interfaz, le da cuerpo a esas funciones.

Ejemplo:

```
public class Caja
{
 // Atributos

 private int value;

 // Métodos

 public Caja()
 { value = 0; }

 public void setValue(int v)
 { value = v; }

 public int getValue()
 { return value; }
}
```

Cuando no se utiliza la cláusula EXTENDS es como si pusiésemos:

```
class Caja extends Object
                   ------
```

Todas las clases heredan de la clase *Object*.

Convenios:
Puesto que Java distingue mayúsculas y minúsculas, se siguen las siguientes pautas:
• Los nombres de clases van con mayúsculas.
• Los nombres de atributos, con minúsculas.
• Los nombres de funciones empiezan con minúscula, y si contienen más de una palabra, las demás empiezan con una mayúscula, sin dejar espacios ni poner otro tipo de caracteres intermedios delimitadores o similares.

[2]Agrupación de clases.

Los **métodos constructores** se definen con el nombre de la clase. No son obligatorios, pues aunque no se pongan existe uno por defecto que inicializa todos los atributos a 0 ó a NULL.

2.2. Objetos

2.2.1. Creación de objetos

Se hace de la siguiente manera:

```
Caja x;

x = new Caja();
    ---
x.setValue(7);
```

2.2.2. Definición

Un **objeto** puede definirse como:

↪ Colección de operaciones que comparten un estado (todas referidas al mismo).

↪ Encapsulación de un conjunto de operaciones y métodos que pueden ser invocados externamente y que recuerdan el estado.

2.2.3. Características principales

Un **objeto** ha de tener:

↬ **Identidad:** debe ser identificable, uno y distinto de otros, aunque objetos de las mismas características se agrupen bajo la misma clase.

↬ **Características declarativas:** usadas para determinar el estado (atributos y su valor).

↬ **Características procedimentales:** que modelan el comportamiento (métodos).

↬ **Características estructurales:** que organizan los objetos unos con respecto a otros (relaciones de herencia, agregación[3],...).

Identidad

La **identidad** es la propiedad de un objeto que lo distingue de todos los demás. Se consigue a través de un **OID** (Object Identifier, *Identificador de Objeto*) independiente del estado (valor de los atributos) del objeto.

Es algo interno que genera y gestiona el sistema y que no se puede ver ni modificar. Suele basarse en punteros y direcciones; de hecho, los propios objetos son siempre *punteros* por razones de asignación dinámica en tiempo de ejecución de espacio de memoria en

[3]Un objeto puede contener a otros.

la *pila* (stack) y el *montículo* (heap); en la primera se almacenan, por *bloques de activación*, los punteros a los objetos y así pueden usarse *offsets* (desplazamientos) para acceder más rápidamente, (en tiempo de compilación no podría hacerse con los objetos en sí sin conocer qué tamanho tienen) mientras que en el segundo se almacenan los propios objetos.

A la hora de comparar dos objetos distinguimos dos situaciones:

- Identidad: dos objetos son *idénticos* si son el mismo objeto, es decir, $\text{OID}_1 = \text{OID}_2$.
- Igualdad: dos objetos son *iguales* si tienen el mismo estado (los mismos valores de sus atributos)[4], pero $\text{OID}_1 \neq \text{OID}_2$.

Idénticos	**Iguales**
(Si cambiamos el valor de x, cambia el valor de y - efecto colateral-)	(Aquí no)

Ahora bien, ¿qué ocurre cuando queremos copiar un objeto? Existen dos "tipos" de copias: la **copia superficial** y la **copia profunda**, que definiremos a continuación. En ambos casos se trata siempre de *objetos iguales*.

> Dos objetos son **superficialmente iguales** si los *atributos* de los objetos son *iguales*. Esta definición *no* es *recursiva*.

La copia superficial está implementada en el método sobreescribible clone de la clase Object a la que pertenecen todos los objetos.

> Dos objetos son **profundamente iguales** si sus *atributos* son *recursivamente iguales*, en todos los niveles.

Características declarativas

Se usan los llamados **modificadores de acceso** para calificar los métodos y atributos que determinarán el estado del objeto: public, private, protected[5] y package[6].

Otros modificadores son también static, usado con atributos que no pertenecen a un objeto particular sino a la clase, y final, usado para constantes.

[4] Suponemos, obviamente, que hablamos de objetos de la misma clase.

[5] Un método o atributo protected es visible desde las subclases y también desde las demás clases pertenecientes al mismo paquete.

[6] Opción por defecto.

Características procedimentales (comportamiento)

Siempre se dice que en POO se busca tener una determinada serie de objetos que se *comunican mediante mensajes*; un *mensaje* se interpreta como una llamada a uno de los procedimientos del objeto pues, como hemos comentado, sus métodos determinan su comportamiento, son su "interfaz" para interactuar con él.

Los métodos constan de:

firma o *cabecera*, compuesta a su vez por:

nombre

tipo y nombre de parámetros, siempre pasados *por valor*, salvo en el caso de los objetos, que al ser, como hemos hecho notar, punteros, son realmente *referencias*.

tipo de resultado, un único tipo de retorno permitido; en caso de devolver más cosas se utilizan o bien array o bien objetos que engloben en sus atributos todo lo que se requiera.

implementación o *código* del procedimiento

Los métodos (y atributos) *de clase* tienen en realidad más utilidad de la que pueda parecer en teoría, pero depende del caso particular que estemos tratando.

En cuanto a los métodos *constructores*, siempre existe uno por defecto, no tenemos por qué definirlo nosotros, pero es habitual *sobrecargarlos*[7].

En Java no hay métodos *destructores*, ya que su función es realizada en la mayoría de los casos por el Garbage Collector[8]. En caso de que no sea así siempre se puede recurrir a la sobreescritura del método finalize, declarado en la clase Object.

El método main ha de ser static porque puede llamarse para ejecutarse sin que esté creado el objeto.

[7] *Sobrecargar* un método consiste en definir varios procedimientos con el mismo nombre pero con parámetros distintos.

[8] La única pega de esta característica de Java es que el Garbage Collector es *asíncrono*, se ejecuta "cuando quiere".

Capítulo 3

Propiedades básicas de la Orientación a Objetos

Tradicionalmente se han tomado como propiedades básicas de la Orientación a Objetos las *características de Booch*[1], que son básicamente cuatro:

- ⟼ Abstracción.
- ⟼ Encapsulamiento o encapsulación.
- ⟼ Modularidad.
- ⟼ Jerarquía.

Además, se añaden:

- ⟼ Polimorfismo[2].
- ⟼ Tipificación y ligadura[3].

3.1. Abstracción

Definición

> Representación de las *características* fundamentales de algo *sin* incluir antecedentes ó *detalles irrelevantes*.

La **abstracción** es fundamental para enfrentarse a la complejidad del software. La Programación Orientada a Objetos fomenta el uso de abstracciones (clases, métodos, especificadores de acceso → privacidad...). Elemento clave: la *clase*, plantilla para crear objetos, especificar su estado y métodos para modificarlo).

[1] Fue uno de los diseñadores de UML.
[2] En esta propiedad es donde reside realmente la potencia de la O.O.
[3] Más bien características de los lenguajes.

3.2. Encapsulamiento

Definición

> Proceso de almacenamiento en un *mismo compartimiento* de los elementos de una abstracción que constituyen su estructura y comportamiento.

Abstracción y **encapsulamiento** son complementarios: la primera se centra en el comportamiento observable del objeto y el segundo en la implementación (métodos, atributos). Contribuye a la ocultación de la información: parte pública (*interfaz*) y parte privada (*implementación*).

Ventajas

- Permite un razonamiento más eficiente al eliminar los detalles de bajo nivel.
- Permite que cambios en el diseño sólo afecten de forma local.

En Java se materializa mediante los especificadores de acceso aplicados en las clases, que limitan la visibilidad de métodos y atributos.

3.3. Modularidad

Definición

> *Propiedad* que tiene un *sistema* que ha sido descompuesto en un conjunto de módulos que han de ser *cohesivos* y *débilmente acoplados*.

Que dos módulos sean *cohesivos* significa que agrupan abstracciones que guardan alguna relación lógica. Que sean *débilmente acoplados* indica que se minimiza la dependencia entre ellos[4].

Ventajas

- La fragmentación disminuye la complejidad.
- Crea fronteras bien definidas y documentadas dentro de un programa, lo que también contribuye a una menor complejidad y también a una mayor comprensión del programa, mejor estructuración...

Existe relación entre la **abstracción**, el **encapsulamiento** y la **modularidad**, son conceptos *sinérgicos*, persiguen el mismo fin.

ENCAPSULACIÓN Un objeto definido alrededor de una abstracción MODULARIDAD

Encapsulamiento y modularidad son barreras alrededor de esa abstracción para protegerla.

[4]Estos conceptos no son exclusivos de la POO.

3.3.1. Criterios de Modularidad de Meyer

[5]

Son una serie de directivas que definen lo que se admite como una correcta modularización en un programa.

1) Descomposición: el software debe ser descompuesto en un número de subproblemas menos complejos que estén interconectados mediante una estructura sencilla y que sean independientes (lo suficiente como para poder trabajar por separado en ellos, *débil acoplamiento*).

2) Composición: propiedad de que los módulos se combinen libremente para producir nuevos sistemas, con la posibilidad de ser completamente ajenos a aquél en que se desarrolló el módulo.

3) Comprensibilidad: módulos comprensibles sin necesidad de examinar aquéllos otros con los que interactúan, comprensibles en sí mismos, o en el peor caso analizando el menos número posible de ellos.

4) Continuidad: pequeños cambios en las especificaciones del problema han de provocar pocos cambios en pocos módulos.

5) Protección: cuando se produce un error o situación anormal o no prevista en un módulo debe quedar confinado en él o bien propagarse sólo a los módulos "vecinos", no a todos.

Para que estos criterios se vean satisfechos debe atenderse a los siguientes principios de diseño:

⋈ Correspondencia directa: La estructura modular (del sistema) software tiene que ser compatible con cualquier otra estructura modular que hayamos obtenido en el proceso de modelado.

⋈ Pocas interfaces: Cada módulo debe comunicarse con el menor número de módulos posible, con el fin de evitar la propagación de errores.

⋈ Interfaces pequeñas: La cantidad de información que se pasa ha de ser lo más pequeña posible.

⋈ Interfaces explíctias y legibles: Se deben especificar claramente los datos y procedimientos que se muestran o no al exterior.

⋈ Ocultación de información: Sólo se muestra la parte del método que se necesita mostrar.

[5] Meyer fue el creador de Eiffel.

En Java:

clases Son el primer paso de la modularidad, encapsulan métodos y atributos y permiten ocultar información (private, protected, public,...).

paquetes Son el segundo paso de la modularidad; constituyen una *ordenación lógica*, puesto que hacen lo mismo con las clases que éstas con sus componentes.

ficheros Con extensión .JAVA, se usan básicamente como unidades de compilación, constituyen la *ordenación física*. Sólo puede haber una clase pública por fichero, y el nombre de éste ha de coincidir con el de aquélla. Si no contiene ninguna, el nombre puede ser cualquiera. Al compilar, se genera un .CLASS por cada clase que esté presente en el fichero.

3.3.2. Objetivos de los paquetes

▷ Son dispositivos de modularidad de nivel superior a las clases.

▷ Agrupan clases con funcionalidades similares, lo cual:

- ◇ Favorece la comunicación entre las clases.
- ◇ Facilita la localización.
- ◇ Favorece que se vea claramente que están relacionadas.

▷ Previene conflictos de nombre (*nombrepaquete.nombreclase*), lo que favorece la reutilización[6].

▷ Organizan los ficheros fuente.

3.4. Jerarquía

Definición

Una *jerarquía* es una *ordenación* o *clasificación* de las abstracciones.

Existen dos tipos de jerarquías:

○ Generalización / especialización: **herencia** ES_UN.

○ Agregación (objetos que contienen otros objetos): ES_PARTE_DE.

[6]También suelen usarse los dominios de Internet al revés: *com.borland...*.

3.4.1. Herencia

La *herencia* es un tipo de jerarquía de abstracciones en la que existen una serie de *subclases* (abstracciones especializadas) que van a heredar características de una o varias *superclases* (abstracciones generalizadas).

El objetivo fundamental de la herencia es, como el de la POO en general, la *economía de expresión* (reutilización).

La herencia define una estructura en forma de *árbol*. En muchos lenguajes existe un nodo raíz común (clase Object de Java, Visual Basic,...).

No obstante, la herencia presenta un conflicto con la encapsulación, puesto que la viola sobre todo a través de la característica protected. Existe un principio, el **principio abierto-cerrado** que dice que los módulos han de ser abiertos y cerrados a la vez. Esto que puede parecer una contradicción, no lo es en realidad porque se refiere a cosas distintas:

> Un módulo es *abierto* si está disponible para ser *extendido* con facilidad.
>
> Un método es *cerrado* si su interfaz es *estable* y está bien definido.

Es decir, una vez creado, se pretende que sea modificado lo menos posible; está abierto, sin embargo, a ser modificado mediante la extensión[7].

Ventajas

- ♠ Favorece la reutilización de código al extender clases que ya existen.
- ♠ Incrementa la fiabilidad del mismo código al extender clases ya probadas.
- ♠ Permite la compartición de código fuente.
- ♠ Permite la consistencia entre las interfaces.
- ♠ Favorece el desarrollo de librerías de componentes (RAD, rapid application development).
- ♠ Permite y favorece el polimorfismo.

Inconvenientes

- ♣ Debemos recordar siempre que medida que se añaden complejidades, tanto a los lenguajes como a nuestras propias aplicaciones) disminuye la velocidad de ejecución.
- ♣ Al incluir librerías enteras, metemos en realidad más código del que realmente necesitamos, con lo cual aumenta el tamaño del nuestro.
- ♣ Aumenta la complejidad de comprensión: *problema del yo-yó* (uso excesivo).

[7] Extensión implica tanto definición de más métodos como redefinición de métodos.

La herencia puede ser **simple** (desciende de una única superclase) o **múltiple** (desciende de varias superclases).

Cuando se tiene *herencia múltiple* la estructura deja de ser en árbol y pasa a ser un *grafo dirigido acíclico* (GDA).

Entre las ventajas de la herencia múltiple tenemos que es un mecanismo muy potente que nos proporciona mucha más flexibilidad.

Entre sus inconvenientes, los conflictos que genera su utilización, que da más problemas que soluciones: el código se vuelve más confuso.

¿Qué método usar? En el caso de la izquierda suele *sobreescribirse* uno por otro; en el de la derecha, o bien se escoge uno arbitrario en el orden en que se indican, o se le pregunta al usuario, o se heredan ambos métodos y se renombra uno de ellos.

El problema en realidad no es que haya conflictos, sino que cada lenguaje los resuelve a su manera[8]; por ello lenguajes como Object Pascal, Java, C# (surgidos de C++) han eliminado la herencia múltiple y simplemente la *simulan* mediante implementación (y compartición) de interfaces.

Clases abstractas e interfaces

La característica principal de una *clase abstracta* es, como ya sabemos, que no puede ser instanciada, de modo que su único *objetivo es ser extendida* por la herencia.

Un *interfaz* supone un paso más, una clase abstracta "total". Sus características son que incluye una serie de métodos que por defecto son públicos y abstractos. Las variables son implícitamente static y final, es decir, contantes de clase. Siguen entre ellos una jerarquía paralela a las clases, pero permitiendo en esta caso la herencia múltiple.

Minimizan los conflictos, pues ahora se hereda la cabecera de modo que los problemas que analizábamos antes desaparecen (salvo si devuelven tipos distintos).

Se permite una *conexión* entre ambas jerarquías (una clase puede extender una superclase y al mismo tiempo implementar algún intefaz):

[8] Java no soporta herencia múltiple.

```
class A extends B implements IX, IM { ... }
```

Además, al no introducir código alguno, los interfaces son utilizables en más ámbitos:

♢ Para capturar similitudes sin forzar relaciones de clases.

♢ Para revelar la interfaz de un objeto (métodos de acceso) sin revelar la clase.

♢ Para definición de nuevos tipos de datos. Sépase que de un interfaz no se pueden crear objetos, aunque sí puede haber objetos que los implementen:

```
interface Celda... { }

        class Imagen implements Celda { }

        ...

           Celda [][] = matriz;
           matriz [0][0] = new Imagen();
```

♢ Se pueden usar como marcadores de clase (interfaces vacíos); en el API de Java tenemos el ejemplo:

```
class MiClase implements Serializable { }

        ...

           MiClase x = new MiClase();
           if (x instance of Serializable) ...
```

♢ Se puede usar como contenedores de constantes globales:

```
interface Constantes {
   public double PI=3.1416;
   public int NUMFILAS = 15;
   ...
}
```

Tipos de herencia

La herencia lleva implícito el **principio de sustitución**: el tipo indicado en la declaración de una variable puede no coincidir con el tipo que contiene la variable en el momento actual (se puede declarar como de la clase y luego ser de la subclase, donde ahora va un tipo luego puede ir otro).

Hay que diferenciar:

subtipo: relación de tipos que reconocen explícitamente el principio de sustitución . Para que esto se cumpla, para poder decir que **B es subtipo de A** son necesarias dos condiciones:

1. Una instancia de B puede ser legalmente asignada a una variable de tipo A.
2. Una instancia de B asignada legalmente a una variable de tipo A puede ser utilizada por la variable de tipo A sin un cambio observable en su conducta.

subclase: B es subclase de A si la extiende a través de la herencia (de los supuestos anteriores sólo se cumple el primero).

La mayoría de las veces las subclases son subtipos, aunque no están obligadas. Los lenguajes de POO suelen entender SUBCLASE = SUBTIPO y permiten por tanto el principio de sustitución.

Herencia de Especialización

Es el tipo más común de herencia; en ella, las *subclases* son *versiones especializadas* de las superclases.

Herencia de Especificación

Una superclase declara unos métodos (abstract) que son implementados por las subclases.

Herencia de Construcción

No existe relación lógica entre subclases y superclases (no existe relación de tipo es_un), pero en las *subclases* se *aprovechan funcionalidades* de las superclases.

Ejemplo típico:

```
class Stack extends Vector {
  public Object push (Object item) {
    addElement(item);
    return item;
  }
  public Object pop () {
    Object obj = peek();
    removeElementAt(size()-1);
    return obj;
  }
  public Object peek () {
    return elementAt(size()-1);
  }
}
```

No se puede decir, empero, que una pila sea un subtipo de Vector, no es lógico que donde tenemos un vector podamos meter una pila, podría hacerse y no fallaría, pero dejarían de respetarse las directivas que hacen de la pila lo que es (por ejemplo, eliminar un elemento del medio). No cumple, por tanto, el principio de sustitución.

Herencia de Limitación

La conducta de la *subclase* es *más restrictiva* que la de la superclase.

En el ejemplo anterior, podría usarse, por ejemplo, para evitar efectos no deseados del tipo:

```
...

public int elementAt (int index)
 throws IllegalOperation {
  throw new IllegalOperation(‘‘elementAt’’);
}

...

class IllegalOption extends Exception {
  IllegalOperation (String str) {
    super(str);
  }
}
```

Aunque soluciona las pegas de la herencia de construcción, esta táctica no es muy "eficiente", pues imaginemos que hubiese que hacer esto para casi cada uno de los métodos de la superclase. Nos plantea si realmente se debería estar usando la herencia o no.

Herencia de Combinación

Consiste en *heredar de más de una* superclase (herencia múltiple).

Java y otros lenguajes, como hemos dicho, lo simulan con interfaces, aunque realmente no es lo mismo porque no heredamos código.

Herencia de Extensión

Se da cuando se hereda de una superclase pero no se redefine nada, sólo se añaden métodos. Es un *caso particular* de la *herencia de especialización*.

3.4.2. Alternativas a la herencia: la Composición

En los casos en los que la herencia no parece la solución más adecuada, puede optarse por emplear la **composición**, que implementa la filosofía ES_PARTE_DE, aunque

la distinción entre ésta y la ES_UN no está demasiado clara[9]. Aparece cuando un objeto está compuesto o *agrega* (*jerarquía de agregación*) varios objetos.

Por ejemplo:

```
class Stack {
  private Vector datos;
  public Stack () {
    datos = new Vector();
  }
  public boolean isEmpty() {
    return datos.isEmpty();
  }
  public void push (Object nuevoValor) {
    datos.addElement(nuevoValor);
  }
  public Object peek(){
    return datos.lastElement();
  }
  public Object pop() {
    Object resultado = datos.lastElement();
    datos.removeElementAt(datos.size()-1);
    return resultado;
  }
}
```

De este modo se utiliza lo que se quiere de la "superclase" y no se heredan métodos o atributos innecesarios.

Ventajas de la herencia sobre la composición

- El principio de sustitución a veces no es deseado, pero en otras ocasiones se requiere; con la composición no es posible nunca.
- Con la herencia tenemos menos código que mantener, ya que no hay *métodos de enlace*, es decir, métodos del estilo:

```
public boolean isEmpty() {
  return datos.isEmpty();
}
```

- En relación con lo anterior, la herencia evita tener que llamar a dos funciones (*delegación*).
- Los elementos protegidos no pueden usarse si se utiliza la composición[10].

[9] Una Stack no *es* ni tampoco *es parte de* un Vector.

[10] En Java sí, siempre que estén en el mismo paquete.

Ventajas de la composición sobre la herencia

- Puesto que *componiendo* no se heredan métodos no deseados, se indican con claridad las operaciones a utilizar.
- Con la composición se puede reimplementar la clase para utilizar otro mecanismo de almacenamiento[11].
- La composición, al no cumplir el principio de sustitución, impide polimorfismos no deseados[12] (lo cual, como ya vimos, es también un inconveniente).
- La composición permite simular la herencia múltiple mediante varios objetos privados internos.

Lo determinante para decidir entre herencia y composición será, como podemos deducir, si queremos consentir o no el mencionado principio de sustitución.

3.5. Polimorfismo

El **polimorfismo** es, como su propio nombre indica, la característica de algo que puede adoptar o tiene varias formas. El problema es que agrupa varios conceptos. Generalmente suele aplicarse a funciones que aceptan parámetros de diferentes tipos, variables que pueden contener valores de distintos tipos,...

Distinguimos algunas variantes que exponemos a continuación clasificadas según la *División de Cardelli y Wegner*:

- Polimorfismo
  - Universal o Verdadero
    - Paramétrico
    - De Inclusión
  - Ad-Hoc o Aparente
    - Sobrecarga
    - Coacción

En el **Polimorfismo Universal** existen valores (atributos) con *distintos tipos* (es decir, pueden tener un tipo declarado y almacenar otro) y se utiliza el *mismo código para todos*[13].

El **Polimorfismo Ad-Hoc** se llama también **Aparente** (y en contraposición el *Universal* es *Verdadero*) porque da la impresión que lo hay pero en realidad no es así. Se utiliza *distinto código* para tratar los *distintos tipos*, de modo que una función que pueda parecer polimórfica en realidad estará implementada a través de varias funciones monomórficas.

3.5.1. Polimorfismo Universal Paramétrico

Este tipo de polimorfismo es propio de los lenguajes funcionales (Lisp, Ml,...)[14] y consiste en *definir una función pero no el tipo de sus parámetros*, de suerte que se puede aplicar sobre diferentes tipos. En C++ se le llama *generidad* ó *genericidad* y está basado en la definición de "plantillas":

[11] Por ejemplo, en el caso particular de las pilas, utilizar otro elemento que no fuese un Vector.

[12] Que sobre la pila puedan aplicarse métodos de la clase Vector.

[13] Ejemplo: Clase Perro, subclase de Animal, con método getNombre que si no se redefine es el mismo para ambos.

[14] No existe en Java.

```
template <classT> class Box
{ ... }

Box <int> a_Box(5);
Box <shape> a_Box(circle);
```

3.5.2. Polimorfismo Universal de Inclusión

Este tipo de polimorfismo es el que se consigue a través de la *herencia*, típico y más común en los L.O.O.[15], que permite que una *instancia sea asignada a una variable declarada de otro tipo.*

3.5.3. Sobrecarga

La **Sobrecarga** consiste en utilizar el mismo nombre para funciones distintas. También es posible la sobrecarga entre clases[16].

La ventaja de usar nombres iguales para hacer los mismo sobre diferentes clases de objetos es la coherencia, aunque no siempre pueden querer significar lo mismo[17].

Sobrecarga Paramétrica

Dentro de una misma clase pueden existir varios métodos con el mismo nombre pero con distinto número de parámetros o parámetros de distintos tipos. El caso más usual es el de los constructores:

```
public Esfera () { ... }
public Esfera (double x, double y) { ... }
public Esfera (int x, double y) { ... }
public Esfera (double y, int x) { ... }
```

Lo que nunca puede ser igual es el *tipo de retorno*, no pueden existir dos métodos con el mismo nombre, número y orden de parámetros, que devuelvan dos tipos diferentes:

```
int abrirFichero ();
void abrirFichero();
```

Si el resultado no se recoge en una variable ¿cuál usamos?

Existen lenguajes *no* orientados a objetos que permiten este tipo de polimorfismo, que suele ser corriente fundamentalmente en *sobrecarga de operadores.*

[15] Y por tanto también en Java.

[16] Las clases Hashtable, Vector,... tienen un método isEmpty(), ¿cuál ha de ejecutarse cuando se le llama? Se sabe por el objeto que hace la llamada.

[17] Si tenemos un metodo isEmpty() sobre un Rectangle, ¿qué quiere decir?

Sobreescritura

Es un caso particular de sobrecarga que se da cuando se *sobrecarga un método que está en la superclase*, siempre que los parámetros sean iguales, ya que si son distintos se considera simplemente sobrecarga.

La **sobreescritura** se puede clasificar a su vez en:

de Reemplazo Se sustituye el método de la superclase por otro.

de Refinamiento Se refina el comportamiento del método de la superclase.

Veamos un ejemplo de cada una:

```
abstract class Personal {
  protected long base;
  protected long años;
  long sueldo() {
    return (base+10000*años);
  }
}

class A extends Personal {
  long numProyectos;
  long sueldo() {
    return (10000*numProyectos);
  }
}

class B extends Personal {
  long extra;
  long sueldo() {
    return (super.sueldo()+extra);
  }
}
```

Siempre que hay **sobreescritura** $\Rightarrow$ hay **sobrecarga** (entre clases). Lo contrario no siempre es cierto.

La **sobrecarga** se decide en *tiempo de compilación*; la **sobreescritura**, en *tiempo de ejecución*:

```
Personal P = new A();
P.sueldo();
```

En tiempo de ejecución se mira qué tipo hay en P y ejecuta el método correspondiente. A esto se le llama *ligadura dinámica* y lo veremos en la sección siguiente (3.6, página 28).

3.5.4. Coacción

La **coacción**[18] es, básicamente, una *conversión implícita de tipos* que se da en casos como:

```
real x;
int i;

x=i+8.3;
```

Lo que se hace es en primer lugar convertir `i` a real y luego realizar la operación.
No obstante, a veces no se sabe distinguir si lo que hay es sobrecarga o coacción:

```
3+4          3.0+4          3+4.0          3.0+4.0
```

¿Es esto un operador sobrecargado 4 veces? ¿O son dos funciones, suma de enteros y suma de reales, combinadas con la coacción de enteros si hay reales presentes? ¿O hay sólo una función suma y siempre coacción de enteros a reales?

3.6. Tipificación y ligadura

Las últimas características que nos quedan por ver es la **tipificación** y la **ligadura**. Cada lenguaje las interpreta y presenta de forma particular y específica, pero definiremos unos rasgos generales.

3.6.1. Tipificación

> Un **tipo** es una *caracterización* precisa de las propiedades estructurales o de comportamiento (atributos, métodos) que comparten una serie de entidades.

Con los *tipos* se consiguen 3 objetivos:

- Un **mecanismo de abstracción**, pues las entidades de un tipo se agrupan según sus propiedades y conducta.
- Un **mecanismo de protección**, que evita errores y garantiza corrección.
- Un **mecanismo de composición**, pues a partir de los tipos existentes se pueden crear tipos nuevos.

Lo que se persigue con los tipos es una *idea de congruencia.*

¿Qué relación hay entre tipos y clases? Los lenguajes O.O. hacen (por sencillez) que no haya prácticamente ninguna diferencia, no obstante, formalmente, la diferenciación que se suele hacer subraya que un **tipo** *especifica una semántica y una estructura* y una **clase** es una *implementación concreta de un tipo*:

[18] Del inglés, coercion.

```
PilaArray                          PilaLista
    |                                  |
    |--> push()                        |--> push()
    |--> pop()                         |--> pop()
    |--> peek()                        |--> peek()
```

PilaArray y PilaLista son dos clases distintas pero son el mismo tipo (pila).

En los L.O.O. las clases suelen definir un tipo, y a parte existen los *tipos básicos*, que no son ni se implementan a través de clases (int, boolean, char,...). En Java, para que podamos tratarlos como clases cuando lo necesitamos (por ejemplo, para poder meterlos donde se requiere un Object), se definen clases *contenedoras* (del inglés wrapped classes): Integer, Boolean, Character,...

```
class Integer {
  private int i;
  ...
  Object getValue();
  void setValue();
  ...
}
```

Comprobación de tipos

Hay varias forma de hacerlo:

- **Forma Estática**
- **Forma Estricta o Fuerte**
- **Forma Dinámica o Débil**

Aunque en ocasiones las dos primeras se agrupan bajo el segundo nombre.

Comprobación de tipos Estática

El tipo exacto se determina en tiempo de compilación. Es bastante restrictivo, y es la que se lleva a cabo siempre en los lenguajes no orientados a objetos.

Comprobación de tipos Estricta o Fuerte

Requiere que todas las expresiones de tipos sean **consistentes** en tiempo de compilación. La *consistencia* exige que se cumplan 3 restricciones:

- *Restricción de Declaración*: Todas las entidades del lenguaje han de tener un tipo declarado, las funciones y sus parámetros también.

- *Restricción de Compatibilidad*: En una llamada x=y o func(x) donde se pase y en lugar de x, se requiere que y sea *compatible* con x (y es *compatible* con x a través de la *herencia*; en otros lenguajes, siempre que sean *subtipos*).
- *Restricción de llamada a característica*: Para que sea posible hacer una llamada x.f (x clase; f método, atributo, característica o variable) el método f ha de estar definido en x o en alguno de sus antecesores.

Esta comprobación es más flexible que la estática y es la que tienen Java, C++, Object Pascal,.... En caso de "duda", aplica el *pesimismo*:

```
Personal P = new A();
P.numProyect();
```

En realidad esto podría funcionar, puesto que P contiene un objeto de tipo A, que posee el atributo numProyect, pero si no lo tuviese (suponer que la asignación a P estuviese en un if o algo así), sería un error, de modo que Java exige una *conversión explícita*:

```
((A) P.)numProyect();
```

Comprobación de tipos Dinámica o Débil

Todas las comprobaciones de tipos se hacen en tiempo de ejecución (se dan, por tanto, un mayor número de excepciones). Es decir, la flexibilidad es mucho mayor, pero nos topamos con muchos más problemas a la hora de ejecutar. Ejemplo: Smalltalk.

Ventajas de la tipificación

Las **ventajas** de ser estrictos con los tipos son:

◇ *Fiabilidad*, pues al detectar los errores en tiempo de compilación se corrigen antes.

◇ *Legibilidad*, ya que proporciona cierta "documentación" al código.

◇ *Eficiencia*, pueden hacerse optimizaciones.

3.6.2. Ligadura

En ocasiones puede confundirse con la *tipificación*, pero la **ligardura** consiste en *ligar* o *relacionar* la *llamada de un método con el cuerpo* del método.

Se da en casos de *sobreescritura*:

```
Personal P;
if ...
   P = new A();
else
   P = new B();
...
P.sueldo();
```

Podría optarse por una solución estática y ejecutar el método **sueldo()** de la clase Personal, pero probablemente no sería lo deseado, de modo que se escoge en tiempo de ejecución (en base al tipo actual, no al tipo declarado), por eso recibe el nombre de **ligadura dinámica**[19].

No todos los métodos permiten ligadura dinámica. Por ejemplo, en Java, los métodos final, que no permiten ser sobreescritos, usan siempre *ligadura estática*. Igual situación se da con los métodos privados. Recordemos que si una clase es final, todos sus métodos lo son. El resto de los métodos en Java emplean *ligadura dinámica* por defecto. En otros lenguajes, como C++ u Object Pascal, se necesita definir los métodos como virtual para que empleen esta técnica (por razones de eficiencia, ya que lógicamente la ejecución es más lenta), pues lo predeterminado es que se liguen estáticamente[20].

Todos los objetos tienen una **tabla de métodos virtuales** (VMT, virtual method table). Para decidir qué código se utiliza ante una determinada llamada, se siguen los pasos:

- Se comprueba el tipo actual correspondiente al objeto que solicita la ejecución del procedimiento.
- Se estudia la VMT de los objetos de dicho tipo y se localiza la entrada correspondiente al método. Es importante hacer notar que el orden de éstos en dicha tabla NO VARÍA en una jerarquía.
- Se sigue el puntero y se ejecuta el código al que apunta.

El principal problema que se le achaca a la ligadura dinámica, como ya hemos dejado entrever, es la **eficiencia**[21], aunque la penalización que ésta impone es constante, no depende del nivel de una jerarquía en que se encuentre,...

Hay compiladores que incluyen *llamadas "inline"*, lo que consiste en incluir el código de un método en el lugar de la llamada (sólo con procedimientos pequeños), algo que evita el paso de parámetros por la pila, ...

No obstante, la ligadura estática también tiene inconvenientes, ya que no permite la *reutilización* e *incumple el principio abierto-cerrado*.

La ligadura estática es válida si produce los mismos resultados que la ligadura dinámica.

Esto puede hacernos pensar que quizás lo mejor fuese que todo lo decidiese el compilador.

[19] En contraposición con la **ligadura estática**, que se decide en tiempo de compilación.

[20] Debe tenerse especial cuidado en no confundir la *ligadura dinámica* con la *comprobación de tipos*.

[21] Es el motivo de que otros lenguajes intenten eludirla.

Capítulo 4

Evolución de los Lenguajes Orientados a Objetos

Lo más importante a la hora de analizar algún diagrama que muestre esta evolución[1] es que distingamos entre los llamados **lenguajes puros**, como Java, Eiffel,... que representan la tendencia actual (lenguajes totalmente orientados a objetos que evitan, que eliminan las cosas molestas como la herencia múlitple) y los **lenguajes híbridos** como C++, Object Pascal, etc.

El resto, como decimos, podemos observarlo en un gráfico dirigido que plasme esta historia más o menos reciente:

ANTECEDENTES HISTÓRICOS

El concepto de *Orientación a Objetos* surge a partir del lenguaje Simula-67, un programador de sucesos discretos. En 1970, con Smalltalk, se acuña el término *Orientación a Objetos*.

El padre de la Orientación a Objetos es **Alan Kay**, creador de una máquina universal (Flex o Dynabook) basada en los conceptos de clase y herencia del Simula-67 y del Lisp.

En los años 70 la Orientació a Objetos y la Inteligencia Artificial comienzan a interrelacionarse de forma estrecha:

- ↬ Aparecen extensiones a lenguajes (Common Lisp Object System o CLOS), entornos de programación (Kee, ART),...
- ↬ Las teorías de herencia de la Orientación a Objetos influyen en el desarrollo de esquemas de representación del conocimiento.
- ↬ Se integran la Orientación a Objetos y la Composición Concurrente.

Ya en los 80, tiene su auge el desarrollo de interfaces de usuario **WIMP**: **w**indows, **i**cons, **m**ouse and **p**ointers (Xerox, Apple). También se desarrolla Smalltalk, y permite ahora la reutilización del software.

Aparecen también los primeros problemas: como la mayoría de los lenguajes Orientados a Objetos estaban muy influenciados por el desarrollo de interfaces de usuario no eran eficaces en otros campos de programación. Para combatirlo,

[1] Como el que tenemos en transparencias, por ejemplo.

nacen lenguajes como Eiffel, C++,...

En la década de los 90 la investigación se centra en:

1. El enfoque de la Ingeniería en el desarrollo del software: CASE, IPSE,...
2. Disposición de componentes de software reutilizables (lenguajes Orientados a Objetos, Ada,...).
3. Sistemas abiertos (UNIX, X-Window,...).

Nacen las Bases de Datos Relacionales, que incluyen extensiones post-relacionales precursoras de las Bases de Datos Orientadas a Objetos.

Aparecen los primeros *estándares*, nace el **OMG** (Object Management Group), las grandes empresas crean los suyos propios: Microsoft, IBM, AT&T,... Este agrupamiento (**OMG**) crea una guía de arquitectura para la integración de la aplicación Orientada a Objetos (Corba), así como el **UML** (Unified Modelling Language).

	Fase I **Década 70**: *Invención*	Fase II **Década 80**: *Confusión*	Fase III **Década 90** : *Madurez*
	Simulación de sucesos discretos	Interfaces: WIMP	Análisis, Diseño
	SIMULA	XEROX, APPLE	Sistemas Abiertos
	Kay: FLEX	Entornos LISP	Aplicaciones
	DYNABOOK	Entornos IA	B.D. Orientadas a Objetos
	Smalltalk	Nuevos lenguajes: Eiffel, C++	Estándares

Cuadro 4.1: Tabla resumen de la evolución de la Orientación a Objetos

Definiciones

P.O.O. Método de implementación en el que los programas se organizan como colecciones cooperativas de *objetos*, cada uno de los cuales representa una instancia de algún tipo de *clase*, tipos miembros de una jerarquía unidos mediante relaciones de herencia.

Diseño O.O. Método de diseño que abarca el proceso de *descomposicón* Orientada a Objetos y una *notación* para describir los modelos lógico y físico, así como los modelos estático y dinámico del sistema que se diseñe.

Análisis O.O. Método de análisis que examina los requisitos desde la perspectiva de las clases y objetos del dominio.

Capítulo 5

UML

Lo principal que veremos en este tema son *diagramas de clase* y cómo permutar entre un diagrama UML y un lenguaje como Java, pero sin meternos en detalles de diseño.

Definición

Un **modelo** es una simplificación de la realidad que busca o permite una mejor comprensión del sistema que estamos analizando.

La utilidad de un *modelo* es la posibilidad de ser usado para:

- VISUALIZAR cómo queremos que sea el sistema
- ESPECIFICAR su estructura y comportamiento
- servir de PLANTILLA que ayude al diseño
- hacer funciones de DOCUMENTACIÓN

UML es también una *metodología de desarrollo*, igual que los modelos *en cascada*:

o *en espiral o incremental*:

La tendencia actual es esta última, más "moderna" y realista.

UML nace para solucionar la diversidad en metodologías orientadas a objetos (Booch, OMT - Object Modeling Technique -, OOSE - Object Oriented System Engineering -, Coad-Yourdon), generando un estándar, pues a pesar de las diferencias el objetivo es común: diseñar la estructura de clases de un sistema, esto es, identificar y establecer las relaciones entre sus clases componentes.

Para ello son muy importantes las dos primeras etapas: el *análisis*, fase preliminar en la que se examinan los requisitos y se identifican los objetos, y el *diseño*, fase más específica en la que se describen los modelos lógico y físico, la estructura de relación entre los objetos[1].

UML es una metodología amplia, pretendiendo abarcar todas las ténicas de modelado, pero sencilla a la vez, y fue desarrollada en origen por Booch, Rumbaugh y Jacobson. En la actualidad, el **O.M.G.** (Object Management Group), que desarrolló los estándares en orientación a objetos, ha dejado paso a la **R.T.F.** (Revision Task Force) en lo que a revisión y evoluciones se refiere.

UML se integra en el proceso de:

- *Análisis Orientado a Objetos*, que se ocupa, entre otras cosas, de pormenorizar los detalles que genera el estudio de los *casos de uso*.
- *Diseño Orientado a Objetos*, que refina el análisis hasta situarse suficientemente próximo al programa. Consecuentemente, está más cerca del código que el análisis. Se distinguen dos tipos:
  - *Lógico - físico.*
  - *Estático - dinámico.*
- *Programación Orientada a Objetos.*

y formalmente puede decirse que es:

Definición

UML es un *lenguaje de modelado* basado en la representación gráfica mediante *diagramas* con el objetivo de:

- Visualizar.
- Especificar.
- Construir.
- Documentar.

Para pasar de los *diagramas* (de clase, de interacción,...), surgidos del análisis y diseño, a un programa a un lenguaje orientado a objetos, existen una serie de **reglas**, que afectan a:

Nombres Definen cómo hay que llamar a los elementos y a las relaciones entre ellos.

Alcance Definen el ámbito en el que es amplicable un nombre.

[1] Un análisis muy detallado se puede tomar como diseño.

Visibilidad Definen cómo se pueden "ver" los nombres.

Integridad Definen las relaciones de unos elementos con otros.

Ejecución Definen una simulación de un sistema / modelo dinámico.

5.1. Mecanismos Comunes

Son *mecanismos comunes* de UML:

* Especificaciones: de *nombre*, *atributos*, *métodos*,....

* Adornos; los típicos son los de *visibilidad*, de *multiplicidad*,...

* Divisiones comunes:

  - ☐ División entre *bloque* e *instancia del bloque*.
  - ☐ División entre *interfaz* e *implementación*.

* Mecanismos de extensibilidad:

  - ☐ Estereotipos, nuevos bloques definidos a partir de los existentes.

  - ☐ Valores etiquetados, usados para añadir información.

  - ☐ Restricciones, que suelen usarse para escribir cosas de un lenguaje concreto que el estándar no implementa.

5.2. Modelado Estructural: Diagrama de clases

Llamamos **ingeniería directa** al paso:

UML $\longrightarrow$ código

Debe tenerse cuidado de no *limitar el UML* tratando de ajustarnos al lenguaje que usaremos después para implementarlo, pues se corre el riesgo de esbozar un diagrama demasiado *dependiente del lenguaje*.

Por contrapartida, tampoco se debe *no limitarlo* en absoluto, ya que en ese caso la *correspondencia* posterior con el *lenguaje* de programación puede hacerse muy *difícil*.

Lo ideal es, por supuesto, encontrar el equilibrio entre ambas situaciones.

También se permite la **ingeniería inversa**, que es el paso:

código $\longrightarrow$ UML

El gran problema que presenta es que suele aportar demasiada información que no es interesante para nosotros.

Capítulo 6

Patrones de Software

Definición

Un **patrón** es una regla con 3 partes que expresa una relación entre un contexto, un problema y una solución.

El objetivo es que exista una literatura de problemas característicos ya resueltos para cuando se nos presente alguno similar acudir a ella y utilizar esa solución, ya que es común en P.O.O. que para ciertos problemas distintos la forma o estructura de la solución que adoptan sea similar. Por tanto, se pretende que haya una serie de soluciones *efectivas* (que ya se sabe que funcionan) y *reutilizables* (que puedan ser usadas en contextos similares).

Esta definición, acuñada por el arquitecto Christopher Alexander, es perfectamente aplicable al desarrollo de software, o al menos así lo entendieron Ward Cunningham y Kent Beck, que la adoptaron por primera vez en la OOPSLA'87, un congreso sobre Orientación a Objetos, en su disertación *Using Pattern Languages for O.O. Programs*.

Su iniciativa sería seguida por más personajes, como Jim Coplien (*Avanced C++ Programming Styles and Idioms*[1], 1991) y E. Gamma, R. Helm, R. Johnson y J. Vlissides, los *GoF* (*Design Patterns Elements of Reusable O.O. Software*, 1995) y M. Grand y J. W. Cooper.

6.1. Tipos de Patrones

Tenemos varios:

Arquitectónicos Expresan la estructura fundamental del sistema software, mostrando sus subsistemas y las relaciones entre éstos. Son patrones de alto nivel y se usan en la fase de análsis.

De Diseño Desempeñan el mismo papel que los anteriores pero a más bajo nivel, pues entran en cómo son las comunicaciones entre los elementos del sistema. Son propios de la fase de diseño.

De Idiomas De nuevo son análogos, aunque más específicos aún porque están relacionados con un lenguaje particular. Se tiene un patrón específico para cada lenguaje, con las diferencias que ello implique.

[1] En este caso el autor llama *idiomas* a los tipos de patrones.

En general suelen usarse los *patrones de Diseño*.

Además de una clasificación de *patrones* se tiene otra división: de **antipatrones**. Mientras que un *patrón* nos muestra una práctica correcta de programación, el *antipatrón* hace lo contrario, nos enseña las cosas que no se deben hacer, como por ejemplo los grandes programas sin modulación, algunas especificaciones de herencia,...

Tenemos dos tipos de antipatrones:

- Los que definen un determinado problema y describen una mala solución que lleva a una mala situación.
- Otros indican, si nos encontramos en una mala situación, cómo transitar a una buena situación, cómo arreglarlo (*refactoring*).

6.2. Patrones de Diseño

Se reconocen fundamentalmente cuatro, los tres primeros definidos por el *GoF* y el cuarto por M. Grand:

Creacionales Tratan sobre cómo crear instancias de los objetos de cara a hacer programas más flexibles y más generales intentando abstraer lo máximo posible esta operación: cuando una clase ofrece unos servicios y otra los usa, se requiere el mínimo acoplamiento para que se produzcan las menores molestias posibles en caso de modificarse la primera.

Estructurales Tratan de cómo realizar combinaciones para formar estructuras mayores (agrupaciones de clases).

Comportamiento Tratan de aspectos relacionados con las comunicaciones entre objetos, cómo llevarlas a cabo para obtener las mayores ventajas,...

Fundamentales Son los patrones de diseño más básicos, tratan del uso de interfaces, la composición, la herencia,...

Capítulo 7

Objetos Distribuidos

7.1. Partes de una aplicación

Son principalmente tres:

- ***Interfaz del usuario***: parte que, como su propio nombre indica, interactúa con el usuario, gestiona la información de entrada (cómo pedirla) y la de salida (cómo mostrarla).
- ***Lógica de la aplicación***: toma la información de entrada del interfaz y la procesa para transformarla y obtener la de salida. Es lo que distingue a unas de otras, en realidad. Suele llamarse también **lógica de negocio** ó **empresarial**.
- ***Servicios***: servicios que se ofrecen a la lógica de la aplicación (por ejemplo bases de datos, archivos, comunicación, impresión,...).

7.2. Tipos de aplicaciones

Distinguimos, según cómo se estructuran las partes anteriores:

* ***Aplicaciones monolíticas ó monocapa***.
* ***Cliente/Servidor***.
* ***Tres capas***.

7.2.1. Aplicaciones Monolíticas o Monocapa

Cada computador que ejecuta la aplicación guarda en sí mismo las tres partes:

Ventajas

- ◦ Fácil diseño e implementación (no hay que tener en cuenta nada relativo a la comunicación entre distintas computadoras...).
- ◦ Adecuadas para un número reducido de usuarios.

Inconvenientes

- ♭ Difícil de mantener (si hay un cambio hay que ir computador por computador actualizando la aplicación en todos...).
- ♭ Poco escalable (si aumentan las necesidades computacionales o de espacio, supone cambiar las máquinas,...).
- ♭ Precisa cierta capacidad de proceso (todos los ordenadores que la ejecutan deben poder hacerlo en su conjunto).

7.2.2. Cliente/Servidor

Tienen el siguiente aspecto (forma más común):

Ventajas

- ♮ Se reparte la potencia de proceso (la máquina cliente no tiene que hacer todo).
- ♮ Al tener un *servidor dedicado* se tiene una gestión de los datos más eficiente.
- ♮ Hay independencia entre la aplicación y los datos (que así pueden utilizarse en otras). Esto también beneficia que la aplicación en su conjunto sea más fácil de mantener, escalar y reutilizar.

Inconvenientes

- √ Aplicaciones más complejas.
- √ Aumento del coste.

7.2.3. Tres Capas

Las tres capas de la aplicación residen en tres computadores distintos. El interfaz se convierte así en un servicio más al servicio de la lógica.

Ventajas

- Simplicidad de mantenimiento (cambios en la lógica no hace falta distribuirlos en todos los clientes).
- Escalabilidad.

Inconvenientes

- Aplicaciones más complejas (que sean más fáciles de mantener no implica que sean más fáciles de construir).

Cada vez hay más aplicaciones de este tipo, sustituyendo a las de tipo *Cliente/Servidor*.

Variantes: n capas

Nos encontramos con esta variante cuando hay un servicio que cubre otro servicio... En realidad seguimos teniendo las tres capas, aunque alguna de ellas se divida en subcapas. Ello no quiere decir que haya n computadores, pueden estar varias capas en el mismo.

Definición

Hiddleware: Es el software que se utiliza entre las distintas capas para favorecer la comunicación entre las mismas.

Es aquí donde entran en juego los *objetos distribuidos*.

7.3. Comunicación entre aplicaciones

Partimos de que existen una infraestructura y un protocolo de red (TCP/IP,...). Podemos comunicar unas aplicaciones con otras:

- Mediante lo que se conoce como **envío de mensajes**,
- mediante **llamadas a procedimientos remotos**, o
- mediante **objetos distribuidos**.

7.3.1. Envío de mensajes

Esta es una técnica de bajo nivel en la que el programador se encarga de gestionar todos los detalles de la comunicación (un ejemplo: *sockets*).

7.3.2. Llamadas a procedimientos remotos

También denominada **RPC** (Remote Procedure Call), Se trata de que una aplicación llama a un procedimiento de un proceso que está en otra máquina. El problema en este caso es el paso de parámetros, que suele ser dependiente del lenguaje utilizado. Para solucionarlo, se desarrolló el **DCE** (Distributed Computing Environment), que define los datos independientemente del lenguaje mediante la **NDR** (Network Data Representation).

Las aplicaciones que usan **DCE RPC** tienen un esquema:

7.3.3. Modelos de objetos distribuidos

Están basados en DCE, RPC... y la ventaja que aportan es que aprovechan la tecnología ya existente y, al utilizar objetos, se tienen llamadas a métodos de éstos y no a procedimientos de librerías, algo que permite aprovechar las utilidades de la orientación a objetos.

Son ejemplos: **DCOM**, **Java RMI**, **CORBA**,....

DCOM

Desarrollado por Microsoft, es la evolución del modelo **COM** (Component Object Model) integrado en **OLE** (Object Linking and Embedding)[1], y sus siglas responden a las iniciales de **Distributed COM**. Lo que se hizo fue aplicar la misma tecnología, la misma idea, para objetos distribuidos.

Ventajas

- Es utilizable desde cualquier lenguaje (siempre que éste tenga las capacidades necesarias.

[1] Aunque también forma parte de ActiveX, por ejemplo.

Inconvenientes

- Está restringido a la plataforma WINDOWS 32-bits.

Java RMI

Desarrollado por los mismos creadores de Java, son las iniciales de **Remote Method Invocation**.

Ventajas

∗ Utiliza tecnología 100 % Java, de modo que se puede utilizar allí donde haya una máquina virtual Java, esto es, no está ligado a una plataforma concreta.

∗ Gran facilidad de uso: los objetos remotos son muy fáciles de crear y utilizar.

Inconvenientes

◦ Está ligado a Java.

CORBA

La **Common Object Request Broker Arquitecture** es la arquitectura más común para los intermediarios en las peticiones de objetos. Está gestionado por el OMG[2].

Ventajas

♭ Puede utilizarla cualquier lenguaje, hay soporte para lenguajes desde los más antiguos (¡como COBOL!) hasta los más modernos.

♭ Pueden alojarse en cualquier plataforma (hay servicios CORBA para hasta una treintena de plataformas: Windows, Linux,...).

Inconvenientes

♮ Su uso es complicado, a veces es más cómoda alguna de las anteriores *soluciones naturales*, que cuentan con facilidades para su manejo...

7.4. Java RMI

7.4.1. Servidor

Entre sus tareas se encuentran:

√ Crear objetos remotos.

√ Hacer disponibles las referencias a dichos objetos remotos.

√ Permanecer esperando a que llamen los clientes.

[2]Object Management Group, grupo de empresas dedicado al desarrollo de estándares, sobre todo en orientación a objetos.

7.4.2. Clientes

Deben poder:

- Obtener una referencia de un objeto remoto (en un determinado servidor).
- Llamar a métodos de esos objetos remotos.

El esquema es más o menos el siguiente:

Skeleton y **Stub** son sendos *proxies* que hacen que la comunicación por la red entre clientes y servidores sea transparente a ambos.

7.4.3. Creación y ejecución (cliente y servidor)

[3]

(1) **Definición del interfaz remoto**.
El servidor ofrece unos servicios encarnados en los métodos de un objeto remoto. Para poder usar esos métodos, por supuesto, hay que conocerlos, de modo que se especifican en un *interfaz*:

```
import java.rmi.*;

public interface InterfazServidor extends Remote {

   public String Consulta (String Criterio)
          throws RemoteException;

}
```

(2) **Implementación del interfaz**.

(3) **Compilación de la clase Servidor**.

(4) **Crear STUB y SKELETON**.
Se lleva a cabo con la orden:

```
$ rmic claseServidor
```

[3] Ver transparencias.

El *rmci* genera dos clases: claseServidor_STUB, que ha de ponerse a disposición del cliente, y claseServidor_SKELETON.

(5) **Ejecutar el registro RMI**.
Es tan sencillo como:

```
$ start rmiregistry
```

(6) **Creación de los objetos del Servidor**.
Para ello simplemente se ejecuta el archivo *Servidor.class*, que en su método main crea una instancia del servidor.

(7) **Registro de los objetos del Servidor**.
El constructor de la clase Servidor llama a la función Naming.rebind, que sirve para llevar a cabo el registro.

(8) **Implementación de la clase Cliente**.
El constructor de la clase Cliente llama a Naming.lookup para obtener la referencia del objeto remoto que quiere manejar. Una vez "traída", se ejectua como si fuera local.

7.5. CORBA

Definido por la OMG, es en realidad una especificación formal del **OMA** (Object Management Arquitecture) al que las empresas se ajustan para fabricar los productos comerciales, lo que hace que no esté ligado a ninguna plataforma.

Los pilares fundamentales de CORBA son tres:

(1) **IDL** (Interface Definition Language): lenguaje de definición de interfaces, para crearlas de manera sencilla e independiente del lenguaje.

(2) **ORB** (Object Request Brokers): gestores de peticiones de objetos, intermediarios entre clientes y servidores.

(3) **GIOP** (General Inter-ORB Protocol): protocolo de comunicación entre ORB's.

7.5.1. IDL

IDL es un lenguaje de definición de interfaces, que son el *contrato* entre clientes y servidores. La ventaja que presenta es que es un *lenguaje neutro* (no es C, ni Java...). A la par existen "puentes" de comunicación entre éstos e IDL, *compiladores IDL*, que a partir de una interfaz IDL obtiene una interfaz en el lenguaje destino:

Así, la forma de crear los interfaces es independiente del lenguaje.

7.5.2. ORB

Los **ORB** añaden una capa más a la comunicación entre cliente y servidor, proporcionando más independencia de la plataforma y del lenguaje al mecanismo CORBA. Son específicos tanto para el lenguaje en que se codifica el servidor (o el cliente), como para la plataforma en que está corriendo.

Su tarea es realizar el *marshalling/unmarshalling* de forma transparente para clientes y servidores.

Los ORB de cada extremo pueden diferir en lenguaje y plataforma, lo que los distingue de *skeleton* y *stub*, que tenían que estar codificados en el mismo lenguaje (Java RMI); lo que habilita esto es el uso de un protocolo estándar como es GIOP.

Son ejemplos: el Visibroker de Inprise (Delphi), Java2 de Java...

7.5.3. GIOP

Este protocolo no existía en la versión 1.0, se definió en la versión 2.0. Es una especificación genérica de cómo conectar dos ORB's, a la que luego se le ha de dar una implementación concreta dependiendo del protocolo de transporte (TCP/IP, PX/SPX...). El estándar obliga a que como mínimo se haga una implementación de **IIOP** (Internet Inter-ORB Protocol), la implementación GIOP para TCP/IP.

7.5.4. Servicios CORBA

Pertenecientes al OMA, sólo algunas implementaciones de CORBA los implementan. Los más comunes son:

- **Nombres**: facilita la resolución de referencias a objetos. Este servicio se estará ejecutando en un lugar conocido en la red, a donde cualquier cliente puede ir a consultar. Los clientes lo usan para obtener las referencias de los objetos remotos.

- **Eventos**: para gestionar los distintos tipos de eventos que se puedan dar entre os objetos CORBA.

Los servicios CORBA son a su vez objetos CORBA que los proporcionan, de modo que pueden ser usadas de modo sencillo por los clientes.

7.5.5. Uso de CORBA desde Java (Ejemplo)

(1) **Crear el interfaz IDL**.

(2) **Compilar el fichero IDL** a nuestro lenguaje:

```
$ idl2java -fno -cpp Consulta.idl
```

Las opciones son para que no se use el preprocesador de C. Esta orden crea una carpeta, SvrConsultas, en la que se guardan varios ficheros:

- Interfaz en Java.
- Esqueleto.
- Stub.
- Dos archivos (clases) auxiliares.

1. Esqueleto del servidor
   Es la clase abstracta **_IConsultaImlBase**, que sirve como base para la implementación del servidor.
2. Stub del cliente
   Es la clase **_IConsultaStub**, que implementa el interfaz **IConsulta**. No se llama a este método, sino que gestiona la llamada remota para llamar al procedimiento definido en el servidor.
3. Clases auxiliares
   Son dos: **IConsultaHelper.java** e **IConsultaHolder.java**. Contienen métodos estáticos (que se pueden llamar sin crear una instancia). La primera contiene el método narrow, que hace un cast a una referencia CORBA para devolver un objeto remoto como un objeto Java, y la segunda controla el paso de parámetros in y out que existen en otros lenguajes como C++.

(3) **Activar el servicio de nombres**, que nos permitirá obtener las referencias a los objetos que queramos. En java2 se denomina tnameserver y generalmente corre en el puerto 900 de la máquina.

(4) **Implementar el servidor**. Se tienen dos clases en el archivo:

1. Clase **ConsultaServant**, que implementa el interfaz.
2. Clase **ServidorConsulta**, que registra el servidor, inicializa el ORB y espera las llamadas de los clientes.

(5) **Implementar el cliente**, que inicializa también su ORB y obtiene referencias de los objetos remotos a través del servidor de nombres.

7.5.6. Acercamiento de Java a CORBA

Se produjo mediante dos acciones principales:

- La inclusión de un ORB en java2.
- La implementación de Java RMI sobre IIOP.

7.6. DCOM

Es el principal competidor de CORBA. Desarrollado por Microsoft, con todo lo que ello supone, destaca entre sus características la de ser independiente del lenguaje.

7.6.1. Interfaces DCOM

También se basa en el concepto de los interfaces que luego son implementados.

En DCOM, los objetos se identifican por medio de un número de 128 bits que se llama **GUID** (*Globally Unique Identifier*) ó **CLSID** (*Class ID*). Existe un método que proporciona estos números garantizando que son únicos tanto en espacio (ya que utiliza la dirección IP ó la dirección Ethernet para calcularlo) como en tiempo (utiliza también la fecha y hora de creación).

Los problemas con que nos encontramos residen en que los objetos DCOM *no son objetos al uso*, esto es, un usuario puede conectarse, llamar a procedimientos del objeto, desconectar y más tarde reconectarse y volver a llamar a más procedimientos, y nada garantiza que la instancia del objeto sea la misma, (eso sí, será de la misma clase), es decir, *no mantienen el estado entre conexiones*.

Los objetos DCOM implementan interfaces, y siempre van a implementar uno denominado **IUnknown**. Este interfaz tiene los métodos:

↪ `AddRef`.

↪ `Release`.

↪ `QueryInterface`.

Los dos primeros se usan para saber cuántos clientes están accediendo al objeto (poseen referencia de él) en un determinado momento. El tercero comprueba qué otras interfaces implementa el objeto en cuestión.

7.6.2. Servidores DCOM

La clase **IClassFactory** permite crear instancias del tipo del objeto, es una clase que hace la función de "constructor".

El equivalente al *registro de nombres* de CORBA es el **registro de Windows**.

7.6.3. Acceso a un objeto DCOM

Hay tres formas de acceder a un objeto DCOM:

(a) **Servidores Internos**. Tanto las aplicaciones como los objetos internos (cuyas referencias se tienen en librerías dll) corren en un mismo espacio de nombres.

(a) **Servidores Locales**. En este caso los objetos se buscan en un exe, por tanto, en distinto espacio de nombres, aunque residan en la misma máquina y mismo sistema operativo. La comunicación se lleva a cabo gracias al **LRPC** (*Lightweight RPC*), una versión más sencilla de RPC.

(a) **Servidores Remotos**. Cliente y servidor se hallan en máquinas distintas, utilizan el protocolo RPC para comunicarse. Si en los dos anteriores se podían usar objetos COM, en este caso es obligado utilizar DCOM.

Índice general