ENTORNOS DE DESARROLLO

2013

Egibide

Entornos de Desarrollo

2015

Departamento de Informática

Desarrollo a Capas

Introducción

La programación por capas es una arquitectura cliente-servidor en el que el objetivo primordial es la separación de la lógica de negocios de la lógica de diseño; un ejemplo básico de esto consiste en separar la capa de datos de la capa de presentación al usuario.

La ventaja principal de este estilo es que el desarrollo se puede llevar a cabo en varios niveles y, en caso de que sobrevenga algún cambio, sólo se ataca al nivel requerido sin tener que revisar entre código mezclado.

Además, permite distribuir el trabajo de creación de una aplicación por niveles; de este modo, cada grupo de trabajo está totalmente abstraído del resto de niveles, de forma que basta con conocer la API que existe entre niveles.

En el diseño de sistemas informáticos actual se suelen usar las arquitecturas multinivel o Programación por capas. En dichas arquitecturas a cada nivel se le confía una misión simple, lo que permite el diseño de arquitecturas escalables (que pueden ampliarse con facilidad en caso de que las necesidades aumenten).

El diseño más utilizado actualmente es el diseño en tres niveles (o en tres capas).

Capas y Niveles

Capa de presentación

Es la que ve el usuario (también se la denomina "capa de usuario"), presenta el sistema al usuario, le comunica la información y captura la información del usuario en un mínimo de proceso (realiza un filtrado previo para comprobar que no hay errores de formato). También es conocida como interfaz gráfica y debe tener la característica de ser "amigable"(entendible y fácil de usar) para el usuario. Esta capa se comunica únicamente con la capa de negocio.

Capa de negocio

Es donde residen los programas que se ejecutan, se reciben las peticiones del usuario y se envían las respuestas tras el proceso. Se denomina capa de negocio (e incluso de lógica del negocio) porque es aquí donde se establecen todas las reglas que deben cumplirse. Esta capa se comunica con la capa de presentación, para recibir las solicitudes y presentar los resultados, y con la capa de datos, para solicitar al gestor de base de datos almacenar o recuperar datos de él. También se consideran aquí los programas de aplicación.

Capa de datos

Es donde residen los datos y es la encargada de acceder a los mismos. Está formada por uno o más gestores de bases de datos que realizan todo el almacenamiento de datos, reciben solicitudes de almacenamiento o recuperación de información desde la capa de negocio.

Todas estas capas pueden residir en un único ordenador, si bien lo más usual es que haya una multitud de ordenadores en donde reside la capa de presentación (son los clientes de la arquitectura cliente/servidor). Las capas de negocio y de datos pueden residir en el mismo ordenador, y si el crecimiento de las necesidades lo aconseja se pueden separar en dos o más ordenadores. Así, si el tamaño o complejidad de la base de datos aumenta, se puede separar en varios ordenadores los cuales recibirán las peticiones del ordenador en que resida la capa de negocio.

Si, por el contrario, fuese la complejidad en la capa de negocio lo que obligase a la separación, esta capa de negocio podría residir en uno o más ordenadores que realizarían solicitudes a una única base de datos. En sistemas muy complejos se llega a tener una serie de ordenadores sobre los cuales corre la capa de negocio, y otra serie de ordenadores sobre los cuales corre la base de datos.

En una arquitectura de tres niveles, los términos "capas" ó "niveles" no significan lo mismo ni son similares.

El término "capa" hace referencia a la forma como una solución es segmentada desde el punto de vista lógico:

- Presentación.
- Lógica de Negocio.
- Datos.

En cambio, el término "nivel", corresponde a la forma en que las capas lógicas se encuentran distribuidas de forma física. Por ejemplo:

- Una solución de tres capas (presentación, lógica del negocio, datos) que residen en un solo ordenador (Presentación+lógica+datos). Se dice que la arquitectura de la solución es de tres capas y un nivel.
- Una solución de tres capas (presentación, lógica del negocio, datos) que residen en dos ordenadores (presentación+lógica por un lado; lógica+datos por el otro lado). Se dice que la arquitectura de la solución es de tres capas y dos niveles.

Introducción a la POO

Abstracción

Todos los lenguajes de programación utilizan abstracciones. El lenguaje ensamblador es una leve abstracción de la máquina subyacente. Muchos de los lenguajes imperativos que le siguieron (como COBOL o C) eran abstracciones del ensamblador. Estos lenguajes fueron un gran avance, pero todavía requieren que el programador piense en términos de la estructura del ordenador en vez de centrarse en la estructura del problema que quiere resolver. La programación orientada a objetos va un paso más allá ofreciendo al programador herramientas para representar elementos en el espacio problema. Esta representación es lo suficientemente general para que el programador no se vea limitado a ningún tipo concreto de problema. Nos referimos a los elementos en el espacio problema como "objetos" . Cada objeto tiene un estado y unas operaciones que podemos pedirle.

La POO permite describir el problema en los términos propios del problema, en lugar de términos propios de la máquina en la que funcionará la solución.

Algunas de las características básicas de un lenguaje orientado a objetos fueron resumidas por Alan Kay:

- **Todo son objetos.** Pensemos en un objeto como en una variable "inteligente"; almacena datos, pero se le pueden "hacer peticiones" para que ejecute operaciones sobre sí misma. En teoría se puede tomar cualquier componente conceptual del problema (perros, edificios, servicios, etc.) y representarlo como un objeto en el programa.
- Un programa es un grupo de objetos diciéndose unos a otros qué deben hacer por medio de **mensajes**. Para hacer una petición a un objeto "se le envía un mensaje" a ese objeto. Más concretamente, podemos imaginarlo como una llamada a un método que pertenece a un objeto en particular.
- Cada objeto posee su propia memoria construida de otros objetos (**herencia**). Dicho de otro modo, creamos nuevos objetos empaquetando objetos existentes. Por tanto, se pueden construir estructura complejas de programación mientras las ocultamos dentro de objetos aparentemente simples.
- Cada objeto tiene un tipo. En la jerga se dice que "cada objeto es una instancia de una clase" siendo "**clase**"sinónimo de "tipo". La característica diferenciadora más importante de una clase es "¿qué mensajes se le pueden enviar?"
- Todos los objetos de un tipo concreto pueden recibir los mismos mensajes. Ya que un objeto de tipo "circulo"es también un objeto de tipo "figura", es seguro que aceptará los mensajes enviados a un objeto de tipo figura. Esto significa que podemos escribir código que maneje figuras que será capaz de manejar automáticamente cualquier cosa que encaje en la descripción de una figura. Esta capacidad de sustitución es uno de los conceptos más potentes de la POO.

Booch ofrece una descripción algo más breve de un objeto:

"Un objeto tiene estado, comportamiento e identidad."

Esto significa que un objeto puede tener datos internos (que le dan un estado), métodos (para producir un comportamiento) y que cada uno de ellos puede ser identificado unívocamente de cualquier otro objeto (se encuentra en una localización de memoria concreta).

Un objeto tiene un interfaz

Aristóteles fue posiblemente el primero en iniciar un estudio detallado sobre el concepto de tipo; habló de la *clase de los peces y la clase de los pájaros*. La idea de que todos los objetos, aun siendo únicos, también son parte de una clase de objetos que tienen características y comportamientos comunes fue utilizada directamente en el primer lenguaje orientado a objetos, Simula-67, con la palabra reservada "class" que introduce un nuevo tipo en el lenguaje.

Simula, como representa su nombre, fue creado para desarrollar simulaciones como la del "cajero de banco". En ella tenemos cajeros, clientes, cuentas, transacciones y unidades monetarias; un montón de objetos. Los objetos que son idénticos salvo por su estado durante la ejecución del programa se agrupan en "clases de objetos" y de ahí surgió la palabra "clase". Poder crear tipos abstractos de datos o clases es un concepto fundamental en POO. Los tipos de datos abstractos funcionan casi igual que los tipos primitivos: podemos crear variables a partir de ellos (llamados objetos o instancias) y manipular estas variables (enviar mensajes o peticiones). De este modo, lo que hacemos en POO es crear nuevos tipos de datos, prácticamente todos los lenguajes orientados a objetos usan la palabra "class". Cuando aparezca la palabra "tipo" tendremos que pensar en "clase" o mejor, en "class".

Una vez que la clase está construida, podemos crear tantas instancias de ella como queramos y manipular estos objetos como si fueran los elementos que existen en el problema que intentamos resolver. De hecho, uno de los retos de la POO es conseguir una equivalencia de uno a uno entre los objetos del espacio problema y los del espacio solución.

Posteriormente debe existir una forma de hacer peticiones al objeto de modo que haga algo, como completar una transacción, dibujar algo en la pantalla o encender un interruptor. Además, cada objeto puede aceptar sólo ciertas peticiones. Este conjunto de peticiones que pueden hacérsele a un objeto componen su interfaz.

El interfaz define que peticiones se le pueden hacer a un objeto concreto. Sin embargo tiene que haber código en algún lugar para satisfacer esa petición. Esto, además de datos ocultos, implica una implementación. Desde un punto de vista procedimental no resulta tan complicado: un tipo tiene una función asociada a cada posible petición y cuando se le solicita, se ejecuta esa función. Normalmente este proceso se resume en que "enviamos un mensaje" (hacemos un petición) a un objeto y el objeto identifica qué hacer en consecuencia (ejecuta código).

Un ejemplo sencillo sería el de una bombilla:

```
Bombilla b = new Bombilla ();
b.encender();
```

En el ejemplo, el nombre del tipo/clase es "Bombilla", el nombre del objeto concreto es "b" y la petición que podemos hacer a un objeto de tipo "Bombilla" es "encender()" o "apagar()". Creamos un objeto Bombilla definiendo una "referencia" ("b") para ese objeto y llamando a "new" para solicitar un nuevo objeto de ese tipo. Para enviar el mensaje utilizamos en nombre del objeto y lo conectamos a la petición o mensaje con un punto (".").

Desde el punto de vista del usuario de una clase predefinida este es todo el trabajo a realizar para programar con objetos.

El diagrama de la figura sigue el formato establecido por el Lenguaje Unificado de Modelado (UML) para una clase. Cada clase se representa con nombre de tipo en la parte superior, datos miembro en la parte media y métodos (funciones que pertenecen a esta clase) en la parte inferior. A veces solamente se muestra en nombre de la clase y los métodos de la parte inferior que forman el interfaz e incluso en ocasiones solamente el nombre, si no es necesario dar demasiados detalles.

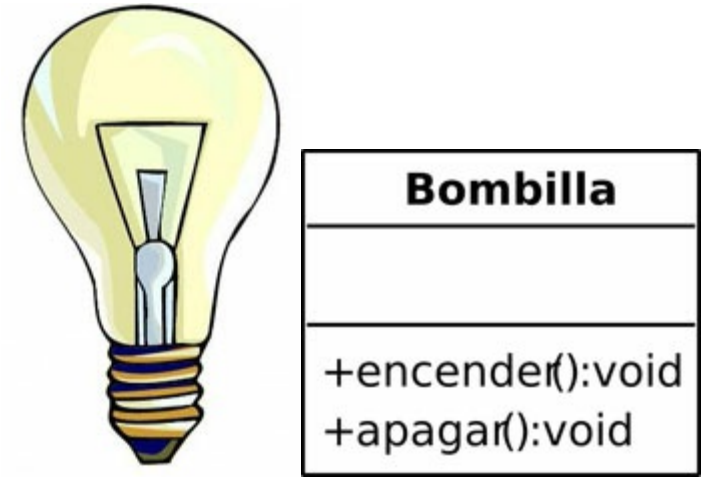


Figura 3.1: Dos maneras de ver una bombilla.

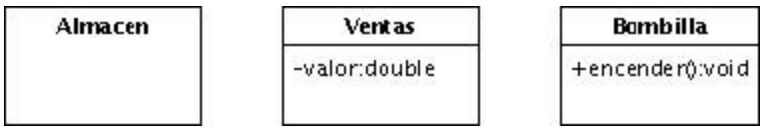


Figura 3.2: Tres clases con diferentes compartimentos visibles.

Un objeto ofrece servicios

Una de las mejores formas de pensar en los objetos es como "proveedores de servicios". Nuestro programa en sí mismo ofrece un servicio al cliente/usuario y logrará este objetivo utilizando servicios de otros objetos. El truco está en producir (o mejor en encontrar en las librerías existentes)

un conjunto de objetos que nos ofrezcan los servicios ideales para resolver nuestro problema.

Una manera de empezar a hacer esto es pensar "si pudiera sacarlos de una chistera, ¿que objetos resolverían mi problema ahora mismo?". Puede que algunos de estos objetos ya existan y para los que no existan podemos preguntarnos como deberían ser y que objetos necesitarían para cumplir su cometido. Si continuamos el proceso, alcanzaremos un punto en el que podremos afirmar "este objeto es bastante simple como para construirlo" o "seguro que tiene que existir ya". Es una manera razonable de descomponer un problema en un conjunto de objetos.

Cada objeto ofrece un conjunto cohesivo de servicios. En lenguaje común: cada objeto realiza una tarea bien, pero no intenta hacer demasiadas. Esto no solo ayuda a encontrar objetos que podemos conseguir (o comprar) ya hechos, sino que abre la posibilidad de reutilizar un objeto concreto en otro lugar o aplicación.

Tratar a los objetos como proveedores de servicios es una gran herramienta de simplificación y no sólo es útil durante el proceso de diseño, sino que también facilita que alguien que intente leer nuestro código o reutilizar un objeto lo entienda o lo encaje en su diseño con más facilidad.

La implementación oculta

Es interesante dividir el terreno de juego entre "**creadores de clases**" y "**programadores usuarios**". El objetivo del programador usuario es reunir un conjunto de clases para utilizarlas en el desarrollo rápido de las aplicaciones. El del creador de clases (o arquitecto) es construirlas de modo que sólo expongan al exterior los mecanismos necesarios para permitir su uso correcto por parte del programador cliente y esconder todo lo demás. ¿Por qué? Porque si se oculta el programador cliente no puede acceder a ello y así el arquitecto puede cambiar su contenido sin preocuparse por el impacto que tendrá en ningún usuario. Además, la parte oculta normalmente representa la "maquinaria" que hay detrás de un objeto y suele ser sensible a programadores descuidados o poco mañosos, así que ocultándola reducimos problemas.

El concepto de ocultación de la implementación es importantísimo. Cuando se crea una relación es importante establecer ciertas "reglas de juego". Si todo estuviera al alcance de todo el mundo, el programador usuario podría saltarse las reglas y producir perturbaciones a otros usuarios. Sin reglas de acceso, los objetos quedarían "desnudos".

Por tanto la primera razón para controlar el acceso es impedir que los programadores pongan las manos encima a porciones de código de las operaciones internas de un objeto que no deberían tocar. En realidad, es una ventaja para ellos, porque pueden ver fácilmente que es lo que pueden utilizar del objeto y que pueden ignorar a través del interfaz que éste expone al exterior.

La segunda razón es que permite al arquitecto cambiar la implementación interna por otra, más eficiente o impuesta por otras condiciones, sin preocuparse porque afecte a los programadores usuarios. Se puede implementar una clase de una forma sencilla mientras desarrollamos la aplicación y más tarde descubrir que el proceso no era tan sencillo. Si el interfaz y la implementación están

claramente separados y protegidos, podremos realizar el cambio sin problemas.

Reutilizando la implementación

Una vez que la clase ha sido construida y probada debería (idealmente) ser una unidad útil de código. Resulta que esta capacidad no es tan fácil de conseguir como muchos esperarían; requiere práctica y reflexión producir un diseño reutilizable. Esta es una de las grandes ventajas que proveen los lenguajes orientados a objetos.

Una de las formas más simples de reutilización es utilizar un objeto de una clase directamente, pero también podemos colocar un objeto dentro de otra nueva clase. A eso se lo denomina "crear un objeto miembro". La nueva clase estará compuesta de un número variable de otros objetos combinados para conseguir la funcionalidad deseada. A esté concepto se le llama composición (y si ocurre dinámicamente agregación). Normalmente a la composición se le denomina relación **contiene un** o **se compone de** como en "un coche contiene un motor".

La herencia, concepto que explicaremos ahora, es un concepto tan importante en POO que muchas veces se intenta utilizar en todas partes. Sin embargo primero deberíamos contemplar la posibilidad de la composición, ya que es un mecanismo más flexible. Con un poco de práctica resultará obvio cuando utilizar una u otra.

Herencia: reutilizando el interfaz

Parece un poco raro tener que hacer un esfuerzo importante para construir una clase y después tener que reconstruirla para crear otra que tenga una funcionalidad similar. Sería bueno poder "clonar" esa clase y entonces añadir o modificar elementos al clon. Esto es en esencia lo que se consigue con la herencia, con la excepción de que si se modifica la clase original (llamada clase base, superclase o clase padre), el "clon" modificado (llamado clase derivada, subclase o clase hija) también sufrirá esos cambios.

Un tipo o clase hace algo más que describir restricciones para un conjunto de objetos; también tiene relación con otros tipos. Puede que dos tipos tengan características o comportamientos comunes, pero uno de ellos tenga alguna característica más o maneje más mensajes (o de forma diferente). La herencia expresa esta similitud entre tipos utilizando el concepto de tipos base y tipos derivados. Un tipo base contiene todas las características y comportamientos compartidos entre los diferenes tipos derivados.

Un ejemplo típico es el de la figura jerarquia_figura, el tipo base es "Figura" que tiene un tamaño, color, etc. Cada figura puede dibujarse, borrarse, moverse. De aquí se obtienen tipos específicos como círculo, triángulo. cada uno con sus características y comportamientos adicionales.

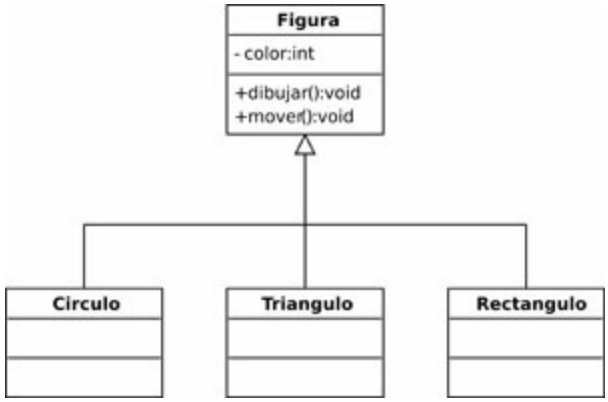


Figura 3.3: Jerarquía de clases para Figura.

Debido a que la clase base y la clase derivada comparten el mismo interfaz, debe haber código que se ejecute cuando el objeto reciba un mensaje concreto. Si simplemente se deriva el objeto y no se hace nada más, los métodos de la clase base serán exactamente iguales a los de la clase derivada, con lo que no se consigue nada interesante.

Hay dos formas de solucionar este problema: la primera es obvia, podemos añadir nuevos métodos a la clase derivada, porque encontramos que los puede necesitar. Hay que ser cuidadosos y observar además, si la clase base también necesita esos métodos.

Aunque la herencia a veces implica (especialmente en Java donde la palabra clave para la herencia es "extends") que vayamos a añadir nuevos métodos al interface, esto no es necesariamente cierto. La segunda forma de diferenciar la nueva clase es alterar el comportamiento de alguno de los métodos de la clase base. A esto se le llama sobrecarga.

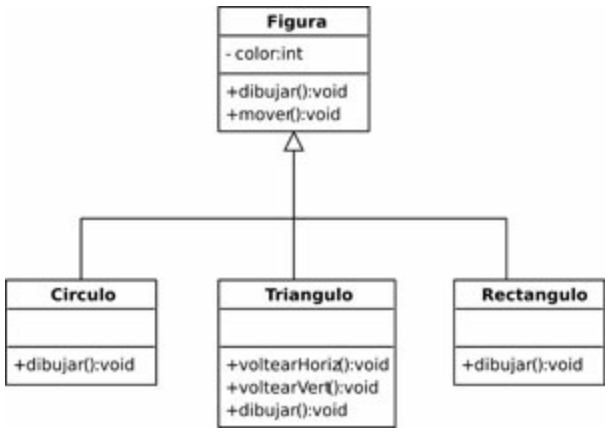


Figura 3.4: Jerarquía con nuevos métodos y sobrecarga.

Objetos intercambiables

Cuando se trabaja con jerarquías de tipos, a veces interesa manejar un objeto no como del tipo concreto al que pertenece, sino como su tipo base genérico. Esto permite escribir código que no depende de tipos específicos. En el ejemplo de las figuras, un método que manipule "figuras" no tiene que preocuparse de si son triángulos, rectángulos o incluso figuras que no hayan sido definidas todavía. Todas las figuras se pueden dibujar, borrar y mover por lo que estos métodos simplemente

envían un mensaje a un objeto de tipo figura; no se preocupan de qué objeto lo trata.

Este código no se ve afectado por la adición de nuevos objetos, y esta es precisamente la forma más común de trabajar en el desarrollo de sistemas orientados a objetos a la hora de afrontar nuevas situaciones. Por ejemplo, podríamos derivar un objeto de la clase pentágono, sin modificar los que trabajan con figuras.

Por ejemplo en el diagrama controla_pajaros tenemos un "ControlaPajaros" que no necesita ni quiere saber el tipo de pájaro al que está pidiendo que se mueva.

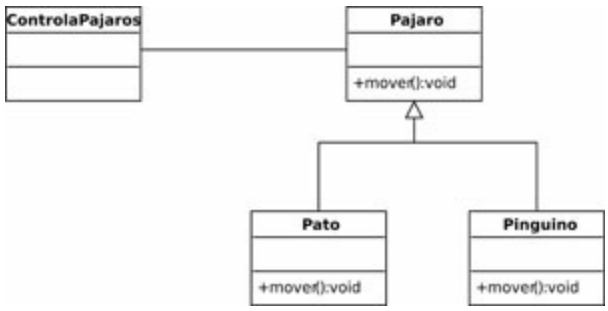
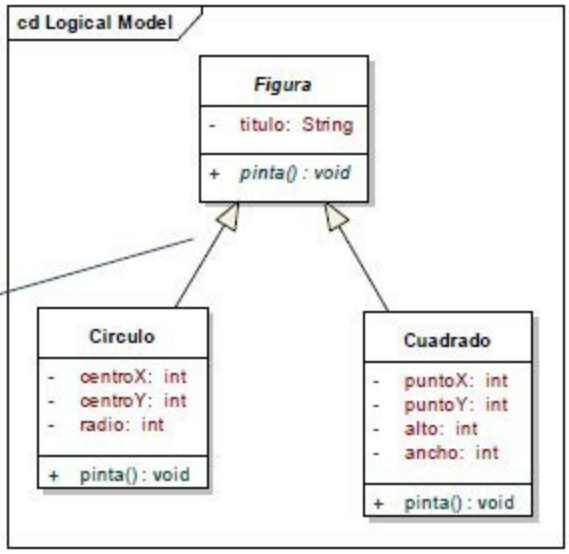


Figura 3.5: Un ejemplo de sustitución en una jerarquía de clases.

CLASE

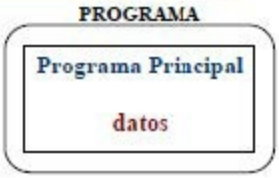
HERENCIA

```
public class Circulo extends Figura
{
private int centroX;
private int centroY;
private int radio;

public void pinta(){...}

}
```



Programación NO estructurada

- Un sólo bloque principal
- Típica de los lenguajes de bajo nivel
- Variables globales a todo el programa



Programación Orientado a Objetos

objeto1
color rojo
tipo deportivo
matricula 1234 ABC
velocidadMax 300

objeto2
color azul
tipo policiaco
matricula 0000 PM
velocidadMax 250

objeto3
color amarillo
tipo elegante
matricula 9876 DEF
velocidadMax 150

objeto4
color rojo
tipo utilitario
matricula 9235 FCB
velocidadMax 200