1. Los objetos colaboran

En POO lo mejor ir encontrando relaciones a partir del comportamiento esperado en los escenarios.

Lo más importante en POO no es tanto la estructura de relacionamiento de los objetos, sino el comportamiento. Por lo tanto, deberíamos ocuparnos de, siguiendo el diseño por contrato, determinar los contratos de cada objeto a implementar (precondicionesa y postcondiciones, invariantes, etc.)

1.1. Relaciones entre objetos

1.1.1. Los objetos interactúan

Todo mensaje tiene un objeto **cliente** y uno **receptor**. Pero los objetos no tienen por qué ser siempre servidores o clientes, sino que su condición cambia según el mensaje. Por ejemplo, un objeto puede ser el receptor de un mensaje y, para poder responder ese mensaje envía a su vez una solicitud a objeto, en cuyo caso se convierte en cliente.

1.1.2. Yendo a lo práctico... o casi

En POO lo mejor no es pensar en relaciones entre objetos per se, sino más bien ir encontrando estas relaciones a partir del **comportamiento** esperado en los escenarios. De esa manera, no tenemos relaciones tan complejas sin una razón para ello.

1.1.3. ¿Cómo implementamos esto?

En POO lo importante no es tanto la estructura de relacionamiento de los objetos, sino el comportamiento.

Por lo tanto, deberíamos ocuparnos de, siguiendo el diseño por contrato, determinar los contratos de cada objeto a implementar.

1.2. Conceptualizando

1.2.1. Dependencia y asociación

Definición 1.1. Dependencia Un objeto depende de otro cuando debe conocerlo para poder enviarle un mensaje. Corolario: Todo objeto cliente depende de su servidor.

Esta dependencia puede venir dada de tres maneras:

- Porque el objeto servidor se envía como argumento.
- Porque el objeto servidor se obtiene como respuesta al envío de un mensaje a otro objeto.

- Porque el objeto cliente tiene una referencia al servidor. (Esto es asociación)

Definición 1.2. Asociación Una asociación es una forma de dependencia en la que el objeto cliente tiene almacenada una referencia al objeto servidor.

Tanto las dependencias más débiles como las asociaciones tienen un sentido: decimos que el cliente depende del servidor, y no al revés. Para un mismo mensaje, lo habitual es que las dependencias sean unidireccionales.

Ejemplo 1.1. Ejemplos de dependencia y asociación (preguntar si es correcto)

- Si tengo una clase A donde los atributos usan una clase B. Esto sería asociación por que el objeto cliente tiene almacenado una referencia al objeto servidor?
- Si tengo una clase A donde sus métodos usan una clase B. Esto sería dependencia por que el objeto cliente, depende objeto servidor?

Sí, tienes razón en ambos casos.

Si tienes una clase A donde los atributos usan una clase B, esto sería una asociación porque el objeto cliente (clase A) tiene almacenada una referencia al objeto servidor (clase B). Esto significa que el objeto cliente puede interactuar con el objeto servidor a través de esta referencia.

Si tienes una clase A donde sus métodos usan una clase B, esto sería una dependencia porque el objeto cliente (clase A) depende del objeto servidor (clase B). Esto significa que cualquier cambio en la definición de la clase B podría afectar a la clase A. Sin embargo, esta dependencia puede ser débil si se hace que el código dependa de interfaces o clases abstractas en lugar de clases concretas.

1.2.2. Interludio metodológico: ¿en qué orden probar cuando tenemos objetos que deben construirse luego?

Definición 1.3. (Digresión) Conceptualmente, si una prueba está probando el comportamiento de más de una clase, o incluso más de una responsabilidad de un objeto, es una prueba de integración, no una prueba unitaria.

1.2.3. Relaciones entre clases

1.2.4. Colaboración por delegación

Uno de los objetivos centrales de la POO era permitir la programación en base a componentes.

La manera más sencilla de vinculación es la simple **asociación** entre objetos. Esta vinculación nos sirve para la delegación de comportamiento.

1.2.5. Programación por diferencia: herencia

Una forma de vinculación de componentes es la herencia, o también relación de generalización-especialización.

Definición 1.4. (Herencia) La herencia es una relación entre clases, por la cual se define que una clase puede ser un caso particular de otra. A la clase más general la llamamos madre y a la más patricular hija.

Corolario: Cuando hay herencia, todas las instancias de la clase hija son también instancias de la clase madre.

Definición 1.5. (programación por diferencia) Programamos por diferencia cuando indicamos que parte de la implementación de un objeto está definida en otro objeto, y por lo tanto sólo implementamos las diferencias específicas.

Como regla, entonces, conviene hacer siempre este test a una relación para ver si aplicar o no la herencia: *¿necesitamos reutilizar la interfaz de una clase tal como está en otra clase, sin que nada me sobre?* Si es así, puedo usar herencia, haciendo que la clase que va a reutilizar sea hija de la clase que provee el código que nos interesa. Si no, conviene reutilizar por delegación.

Dejar la herencia solamente para aquellos casos en que la clase madre tenga una interfaz contenida en la interfaz de la clase hija. Es decir, que la clase madre no tenga métodos que sobren en la clase hija.

NOTA: La herencia se puede usar en cualquier tipo de clase pero comumente se nota más, su uso, en las clases abstractas y en las interfaces.

1.2.6. Programación por diferencia: delegación de comportamiento

Definición 1.6. Delegación de comportamiento Cuando deseamos programar por diferencia en JavaScript recurrimos a la delegación de comportamiento, por la cual un objeto delega su comportamiento en otro objeto prototípico sin necesidad de repetir las definiciones de los métodos.

1.2.7. Redefinición

Definición 1.7. (Redefinición) La redefinición existe para definir un mismo comportamiento en una clase derivada, para el mismo mensaje de la clase base. Por lo tanto, la semántica o significado del mensaje se debe mantener. Si así no fuera, conviene definir un método diferente, con nombre diferente.

En la refinición es importante que se mantenga la firma del método, las precondiciones y postcondiciones pueden variar ligeralmente.

1.2.8. Clases abstractas

Definición 1.8. (Clase abstracta) Una clase es abstracta cuando no puede tener instancias en forma directa, habitualmente debido a que sus clases descendientes cubren todos los casos posibles.

Cuando queramos indicar una clase que no es abstracta (que puede tener instancias) la llamaremos clase concreta.

1.2.9. Métodos abstractos

Definición 1.9. (Método abstracto) Un método es abstracto cuando no lo implementamos en una clase, pero sí deseamos que todas las clases descendientes puedan entender el mensaje.

Cuando queramos indicar un método que no es abstracto (que tiene implementación) lo llamaremos método concreto.

Como corolario, un método abstracto no va a tener implementación, sino que sólo se define su firma, para que las clases descendientes lo implementen en base a ella.

1.3. Cuestiones de implementación

1.3.1. Delegación y herencia en Smalltalk y Java

La delegación tiene una implementación muy directa en ambos lenguajes: basta recibir al objeto servidor como parámetro o como respuesta a un mensaje, o simplemente tener una referencia a él, para poder delegar.

1.3.2. Visibilidad

Hay tres niveles de visibilidad:

- **Pública**: el método, atributo o clase en cuestión se puede utilizar en cualquier parte del programa.
- **Privada**: el método, atributo o clase en cuestión se puede utilizar solamente dentro de su clase.
- **Protegida**: el método, atributo o clase en cuestión se puede utilizar solamente dentro de su clase o una clase descendiente de ella.
- **Paquete**: el método, atributo o clase en cuestión se puede utilizar solamente dentro de su paquete.

En Smalltalk hay menos control de la visibilidad por parte del programador: todos las clases y los métodos son públicos y los atributos son todos protegidos.

- **Smalltalk**: Todos los métodos son publicos, todos los atributos son protegidos (aunque se recomienda considerarlos privados).
- **Java**: Los métodos y atributos pueden ser publicos, privados o protegidos.

1.3.3. Métodos y clases abstractos

Smalltalk, las clases abstractas son aquellas que tienen algún método abstracto, sin que tengamos que declararlo en forma explícit

Smalltalk, como recién dijimos, maneja la cuestión en tiempo de ejecución, lanzando excepciones. En Java, en cambio, es el compilador el que chequea que no se llame un método abstracto o se intente instanciar una clase abstracta.

1.3.4. Constructores en situaciones de herencia y asociación

1.3.5. Herencia y compatibilidad en lenguajes con comprobación de tipos estática

1.4. Modularización

Principios de diseño:

- clases
- Paquetes
- Métodos

1.4.1. Cohesión y acoplamiento

Definición 1.10. (Cohesión) La cohesión es un principio de diseño que indica que los elementos de un módulo deben estar relacionados entre sí, y que el módulo debe tener una única responsabilidad.

Para mantener compresibles, mantenibles y reutilizables a los módulos.

1.5. Pricipios SOLID

1.5.1. Única responsabilidad (SRP)

Tiene que ver con la cohesión.

Cada clase o método debe tener una única responsabilidad, y por lo tanto una única razón para cambiar.

Por ejemplo: si tenemos una clase que se encarga de la lógica de negocio y de la persistencia (guardar), esta clase tiene dos responsabilidades y por lo tanto dos razones para cambiar.

1.5.2. Abierto-cerrado (OCP)

"Las clases tienen que estar cerradas para modificación, pero abiertas para reutilización".

- No modificar las clases existentes
- Extenderlas o adaptarlas por herencia o delegación.

1.5.3. Sustitución de Liskov (LSP)

En los clientes, debemos poder sustituir un tipo por sus subtipos.
Relación es .[es] un"La subclase sea un subconjunto de la clase. Las clase base no deben tener comportamiento que dependan de las clases derivadas.

1.5.4. Segregación de interfaces (ISP)

1.5.5. Inversión de dependencias (DIP)