Criterios y heurísticas de diseño

(versión 3.0)

El objetivo de este documento es ayudarnos a evaluar críticamente y en detalle nuestros diseños y programas orientados a objetos, en términos de los conceptos vistos en Orientación a Objetos 1. Podemos utilizarlo como un checklist cada vez que resolvemos un ejercicio, cuando ayudamos a un compañero a analizar sus programas, o incluso cuando nos preparamos para un exámen.

No todos los ítems nos van a sonar claros al principio. Los iremos encontrando y entendiendo gradualmente. Ante la duda, preguntamos.

Comprobaciones básicas

Las siguientes comprobaciones enfocan principios básicos de la programación orientada a objetos.

- La sintaxis debe ser objeto-mensaje (siempre hay un receptor del mensaje, nunca lo olvido aunque en algunos lenguajes de programación no sea necesario).
- En Smalltalk, todas las líneas terminan con un punto (.). Esto es opcional para la última línea.
- En Smalltalk, la asignación es con dos puntos e igual (:=).
- Si quiero enviar varios mensajes a un mismo objetos, uso punto y coma (;) antes de cada nuevo mensaje y punto (.) para terminar.
- No olvidar retornar explícitamente un objeto cuando eso es lo que se espera (caso contrario, mi programa se comportará de forma extraña y me costará encontrar el problema).
- No se puede acceder directamente a las variables de instancia de otro objeto, aunque algunos lenguajes lo permitan (el ocultamiento de información es uno de los principios más importantes de la POO).
- No confundir mensajes de clase con mensajes de instancia.
- No se puede acceder a las variables de instancia desde los métodos de clase (por ejemplo desde los constructores).
- Al programar en papel no olvidar especificar superclase y variables de instancia. Si hay métodos de clase, hacerlo explícito.

Malos olores de diseño

Los siguientes son malos olores en el diseño OO que no deberían estar presentes en nuestros programas una vez que completamos Orientación a Objetos 1. Debemos saber reconocerlos y evitarlos.

Envidia de atributos: soy un objeto que pido cosas a otros objetos para hacer algo (por ejemplo un cálculo) yo mismo ...

Para evitarlo: la tarea la debe hacer el objeto que tiene las cosas que se necesitan; delegárselo a él.

Clase Dios: Una clase que hace todo y las demás están todas anémicas (ver clases anémicas). Una clase así no cumple el principio de "una sola responsabilidad". Seguramente, si me pregunto qué hace, tengo que decir "hace tal cosa y además... ". Probablemente también haya 'envidia de atributos' si es que otros objetos, al menos, tienen información.

Para evitarlo: Ver qué otros objetos podría hacer aparecer, que se puedan encargar de alguna de las responsabilidades de éste. Ver, de los objetos que este objeto conoce, cuál podría ser responsable por algo que ahora hace él.

Código duplicado: si hago Ctrl+C Ctrl+V (copiar & pegar) estoy metiendo la pata.

Para evitarlo: ¿No puedo generalizar ese comportamiento en una clase y heredarlo? ¿No puedo llevarlo a otro objeto y reutilizarlo por composición?

Clase larga: tengo una clase muy grande en comparación al resto.

Para evitarlo: ¿No será que esa clase puede delegar algo en otros objetos a los que conoce? No será que esa clase modela más de una cosa (puedo pensarla como una composición de varios objetos?)

Método largo: si un método tiene más de 10 renglones, es mala señal. Si debo incluir comentarios en medio de un método, es mala señal.

Para evitarlo: Identificar dentro del método largo, partes que podría considerar comportamientos individuales. Llevar cada parte a un nuevo método (con un buen nombre) y cuando necesite llevar a cabo uno de esos comportamientos, enviar mensajes a self.

Objetos que conocen el id de otro: Nunca relacionar objetos por medio de claves o ids!!

Para evitarlo: Cuando un objeto se relaciona con otro, lo hace con una referencia. Nunca conoce su id (incluso si los objetos tienen id)

Eso debería ser un objeto (obsesión por los primitivos): A veces modelamos como strings o números, cosas que deberían ser objetos. Cuando hacemos eso, el comportamiento que debería tener ese objeto termina estando en un lugar que no corresponde.

Para evitarlo: Pensar si eso que estoy modelando con un string o número (un primitivo) no debería ser modelado con una clase específica.

Switch statements: Debería sentir mal olor cuando veo que se usa un ***if*** *(o algo que parece un* ***case*** *o un* ***switch*** *o* ***ifs anidados****)* para determinar de qué forma se resuelve algo. Esto es más evidente si la variable que uso en el *if* tiene un nombre que suena a "tipo". Algo como "if (tipo = esto) entonces lo hago así, pero si (tipo = aquello) entonces lo hago asá" tiene muy feo olor. El caso más extremo es preguntar por la clase del objeto (si es de esta clase lo hago asi, sino lo hago asá).

Para evitarlo: Aplico adecuadamente polimorfismo!

Variables de instancia que en realidad deberían ser temporales: Si una variable de instancia deja de tener sentido en algún momento de la vida del objeto, entonces es probable que sea temporal o es reponsabilidad de otro.

Para evitarlo: pensar si esa variable es realmente un atributo del objeto, que lo acompaña siempre, o es algo que necesito temporalmente dentro de un método.

Romper encapsulamiento: Romper el encapsulamiento de un objeto es muy malo**.** Nos hace perder la gran mayoría de las ventajas de la OO. En Smalltalk no podemos modificar "desde afuera" las variables de instancia de un objeto, pero podemos romper el encapsulamiento de manera más sutil. Por ejemplo, si automaticamente agregamos setters y getters para todas las variables de instancia de nuestros objetos, estamos invitando a otros a que las modifiquen cuanto quieran (como si no existiera en ocultamiento de información) al modificar objetos o colecciones que son de otros. Si modificamos una colección que no es nuestra (es de otro objeto) también atentamos contra el encapsulamiento.

Para evitarlo: solo agregar getters y setters cuando es necesario; nunca modificar una colección que no es nuestra, delegar las tareas a los que tienen la información que se necesita.

Clase de datos o clase anémica: una clase que parece un registro de datos debería dar mala espina. A veces los enunciados son simplificaciones que hacen que algunas clases terminen siendo así, pero por lo general sospecho cuando una clase solo tiene datos y no tiene comportamiento.

Para evitarlo: asegurarse que no hay comportamiento en el sistema que debería estar haciendo esa clase y lo hace otro objeto (el cual seguramente muestre envidia de atributos).

No es-un: Una relación de herencia (clase B hereda de clase A) siempre debe respetar el principio **es-un**. Si me pregunto "¿un B, es un A?, la respuesta debe ser SI. Si la respuesta es NO, eso tiene mal olor.

Para evitarlo: Siempre preguntarme "es_un" cuando defino una subclase. Si la respuesta es no, pensar un poco más. A veces el problema es que elegí mal los nombres de las clases. A veces es

señal de que tanto A como B con subclase de otra clase que todavía no apareció. A veces es señal de que la relación de subclasificación no es la correcta para ese caso, y debo pensar otras alternativas (como composición).

No quiero mi herencia: cuando encontramos un método que redefine a uno heredado pero hace algo totalmente diferente, debemos desconfiar. Si sirve el comportamiento heredado tal vez no se cumpla el principio "es-un". El caso extremo es redefinir un método heredado, para indicar un error o no hacer nada.

Para evitarlo: pensar si no puedo reorganizar la jerarquía de clases para que ninguna clase herede comportamiento que no quiere

Me olvidé que lo heredaba: redefino un método que heredo, cuando quiero agregar comportamiento al que se define en la superclase. Por ejemplo, hacer algo antes de lo que heredo y/o hacer algo después de lo que heredo. En cualquiera de esos casos, debo incluir el comportamiento heredado por medio de la pseudo-variable "super". Si me olvido de "super" quiere decir que "no quiero mi herencia", y eso es malo.

Para evitarlo: no olvidar de incluir el comportamiento heredado, utilizando la pseudo-variable "super" (que no es otra cosa que enviar el mensaje a self, pero iniciando el method lookup un poco más arriba).

Reinventando la rueda: un principio fundamental de la POO es que las cosas se escriben una sola vez y donde corresponde. De esa manera, mis módulos (objetos/métodos) son más fáciles de mantener y reutilizar. Tiene mal olor cuando defino comportamiento que sospecho que ya está programado en algún lado (hay algún objeto que ya sabe hacer eso). El ejemplo más común en Smalltalk es utilizar siempre el #do: de collection, cuanto existen otros métodos que ya hacen lo que necesito (#select: , #detect:, #collect: , #sumNumbers:)

Para evitarlo: investigo y aprendo las clases y protocolos que ofrecen las librerías de objetos a mi disposición. Intento siempre utilizar comportamiento que ya fue definido. Presto especial atención cuando utilizo colecciones, fechas, ...

Estilo de programación

Los siguientes son patrones de estilo y buenas prácticas de programación que deberían respetarse en los programas hechos por los alumnos que aprueben la materia.

Ofrecer constructores (también llamados métodos de creación completos). Simplifican a quien crea los objetos. Garantizan una buena inicialización.

Nombre de mensaje que revela la intención: Que el nombre del mensaje comunique lo que se quiere hacer, no cómo.

Delegación a self: Permite descomponer un método en partes que el mismo objeto resuelve. Cada método hace una cosa. Su nombre indica lo que hace. Quedan todos cortos. Permite que la subclase redefina/extienda solo un paso.

Métodos cortos: Siempre prefiere tener métodos cortos. Para lograrlo utiliza delegación a self. Para que sean más fáciles de leer, utiliza nombres de mensajes que revelen la intención (servirán como documentación de lo que hace el código)

Cada cosa se hace una sola vez: Para ello es importante aprender el protocolo de colecciones y otros objetos frecuentemente utilizados. Es recomendable explorar el protocolo de los objetos que voy a utilizar antes de comenzar a programar.

Inicializa los objetos de manera explícita: Siempre que sea posible incluye un método #initialize que inicializa todos aquellos atributos que pueden tomar valores preestablecidos. Si se requieren parámetros, define un método de inicialización completo (que recibe todos los parámetros necesarios) y clasificarlo en el protocolo "initialize"

Los nombres de las variables deben indicar su rol. Elige los nombres de las variables que quede claro qué rol cumplen el el método / clase. Los nombres de variables siempre comienzan con minúscula. No temas a los nombres de las variables largos, con varias palabras y sintaxis de camello.

Piensa bien los nombres de las clases. Estos siempre inician con mayúscula y singular. No temas a los nombres de clase largos, con varias palabras y sintaxis de camello. Si se puede, que el nombre de la subclase ayude a reconocer que es un caso particular de la superclase (por ejemplo, agregando alguna palabra al nombre de la superclase para definir un caso especial: EmpleadoDePlanta subclase de Empleado)

Otros criterios y buenas prácticas

(utilice este espacio para documentar otros criterios que haya escuchado en clase)

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....