Criterios y heurísticas de diseño

(versión 4 agosto 2019)

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 objeto, 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 y pegar) estoy metiendo la pata.

Para evitarlo: ¿No puedo generalizar ese comportamiento en una clase y heredarlo? ¿No puedo llevarlo a otro objeto y re-utilizarlo por composición? ¿No puedo extraerlo en un método en la misma clase y re-usarlo?

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 aunque los objetos tengan 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 así, si no 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 que sea responsabilidad 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 automáticamente 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 un 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: sólo 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 SÍ. 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 son 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 no 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.

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, cuando 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, ...

Instancio e inicializo en varias líneas: Me olvidé de definir un constructor de una clase y para poder utilizar sus instancias me doy cuenta de que luego de invocar el #new **siempre** debo setearle las variables inmediatamente para poder utilizar la instancia. Por ejemplo, tenemos que setear individualmente el combustible y el precio del surtidor para poder vender 50 litros de nafta:

```
surtidor:= Surtidor new.
surtidor combustible:’Nafta super’.
surtidor precio:38.
surtidor venderLitros: 50.
```

Para evitarlo: Defino nuevos constructores que reciban como parámetro todos los valores que necesito para hacer la inicialización y conseguir como resultado una instancia lista para ser usada. De esta forma simplifica la tarea a quien crea los objetos. Garantizan buenas inicializaciones. Y en caso que los setters anteriores no fuesen necesarios y los hice automáticamente evito perder encapsulamiento.

Programo un constructor usando varios setters: Cuando programo el constructor para solucionar el mal olor anterior (Instancio e inicializo en varias líneas), debo evitar utilizar un método setter por cada una de las variables de instancia que debo inicializar y en cambio programar un método de inicialización que reciba como parámetro todas las variables. Por ejemplo, si queremos definir el constructor de un Surtidor donde se indica el nombre del combustible y el precio por litro de ese combustible. A continuación detallamos un ejemplo con el mal olor:

```
Surtidor (clase)>>combustible: unNombre precio:unPrecioPorLitro
  |instancia|
  instancia:=self new.
  instancia combustible: unNombre.
  instancia precio: unPrecioPorLitro.
  ^instancia.
```

Para evitarlo: realizo la inicialización mediante un método de instancia específico que reciba todos los parámetros. Desde el cuerpo del constructor invoco a este método de inicialización luego del mensaje new.

```
Surtidor (clase)>>combustible: unNombre precio:unPrecioPorLitro
  ^ self new inicializar:unNombre precio: unPrecioPorLitro.
Surtidor (instancia)>>inicializar:unNombre precio: unPrecioPorLitro
  nombre:= unNombre.
  precio:= unPrecioPorLitro.
```

Estilo de programación

Los siguientes son patrones de estilo y buenas prácticas de programación que deberían respetar nuestros programas.

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".

Ofrecer constructores (también llamados métodos de creación completos). Simplifica la tarea de 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.

Los nombres de las variables deben indicar su rol. Elige los nombres de las variables para que quede claro qué rol cumplen en 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)

Algunas recetas útiles

Para el manejo de fechas puede utilizar la clase Date. La misma, devuelve una instancia que representa al día actual si se le envía el mensaje de clase #today:

```
Date today
```
“Devuelve una instancia de Date para la fecha 8 de Octubre del 2018”.

Si desea comparar dos fechas, simplemente puede usar el mensaje de instancia #=
```
unaFecha = Date today
```

“Esta sentencia da como resultado true si unaFecha es una instancia de Date que representa al día de hoy”.

Para construir una fecha específica, el mensaje #fromString puede ser útil:

```
Date fromString: '1/1/2019'.
```

“Esta sentencia da como resultado la fecha que representa al primero de enero de 2019”

Sobre Colecciones

Tenga en cuenta también los mensajes #select: y #reject:, #sum:, #collect: , #sort y #sort:, que entienden las instancias de colecciones.

El mensaje #select: permite obtener los objetos de una colección que cumplen con una condición. Supongamos que la colección <coleccionDeFechas> contiene instancias de la clase Date, entonces:

```
coleccionDeFechas select: [ :fecha | fecha < Date today ]
```

retornaría una colección de las fechas contenidas en <coleccionDeFechas> que sean menores a la fecha de hoy. Si usamos, en la sentencia anterior, el mensaje #reject: en lugar del #select:, obtendríamos una colección de las fechas mayores (o iguales) a la fecha de hoy.

Por otro lado, el mensaje #sumNumbers: nos permite hacer una sumatoria de los objetos contenidos en una colección, o bien de lo que devuelve algún mensaje enviado a estos objetos. Supongamos que <coleccionDeCajasDeAhorro> contiene varias instancias de la clase CajaDeAhorro, entonces:

```
coleccionDeCajasDeAhorro sumNumbers: [ :caja | caja saldo ]
```

retornaría la suma de todos los saldos de las cajas de ahorro de la colección.

El mensaje #collect: permite aplicar un bloque a una colección y retornar otra con la misma cantidad de elementos pero donde cada uno es el resultado de aplicar el bloque al de la colección que recibió el #collect: Por ejemplo, supongamos que <coleccionDeCajasDeAhorro> contiene varias instancias de la clase CajaDeAhorro, y una CajaDeAhorro sabe responder a los mensajes #cliente, #fechaDeLaUltimaExtraccion, #saldo, entre otros.

```
coleccionDeCajasDeAhorro collect: [ :caja | caja saldo ].
```

retornaría una nueva colección, con los saldos de cada una de las cajas de ahorro de coleccionDeCajasDeAhorro.

```
coleccionDeCajasDeAhorro collect: [ :caja | caja fechaDeLaUltimaExtraccion ].
```

retornaría una nueva colección, con las fechas de la última extracción de cada una de las cajas de ahorro de coleccionDeCajasDeAhorro.

```
coleccionDeCajasDeAhorro collect: [ :caja | caja cliente ].
```

retornaría una nueva colección, con los clientes de cada una de las cajas de ahorro de coleccionDeCajasDeAhorro.

Al momento de ordenar una colección, los mensajes #sort y #sort: nos pueden simplificar la tarea. El mensaje #sort retorna la colección ordenada, pero ¡OJO! Lo hace con el criterio #<=; esto quiere decir que los objetos que tengo dentro de la colección deben entender el mensaje #<=, de otro modo, no funcionaría. Por ejemplo, si tuviéramos en coleccionDeNumerosAlReves a los números del 10 al 1 (10, 9, 8, 7, 6, 5, 4, 3, 2, 1), la siguiente sentencia retornaría la colección con los números del 1 al 10.

```
coleccionDeNumerosAlReves sort.
```

Supongamos que queremos ordenar la coleccionDeCajasDeAhorro, de menor a mayor, por su saldo.
¿Que ocurriría si ejecutaramos la siguiente sentencia?

```
coleccionDeCajasDeAhorro sort.
```

MessageNotUnderstood!!! Las instancias de cajas de ahorro no entienden el mensaje #<=.

¿Cómo lo solucionamos? Simple, utilizamos el mensaje #sort:, indicando como parámetro un bloque que nos indique sobre qué queremos ordenar las cajas de ahorro (OJO, el bloque debe dar como resultado un objeto que entienda el mensaje #<=).

```
coleccionDeCajasDeAhorro sort: [:cajaDeAhorro | cajaDeAhorro saldo].
```

Cómo programar en papel (y no morir en el intento)

Primero declare la clase y sus variables de instancia, por ejemplo, para declarar la clase CajaDeAhorro como subclase de Object y con dos variables de instancia saldo y cliente, debe escribir lo siguiente:

Object subclass: #CajaDeAhorro
v.i.: saldo cliente

Luego declare los métodos de la clase. Si el método es de clase, indique en forma clara antes de la signatura del nombre que el mismo es de clase como muestra el ejemplo:

class >> para: unCliente
    "Aquí va el código del método de clase, por ejemplo un constructor"

Si el método es de instancia, no debe agregar la palabra class antes de >>, como muestra el siguiente ejemplo:

>> initialize
    "Aquí va el código de este método de instancia"

Otros criterios y buenas prácticas

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

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