Heurísticas de Diseño

El diseño orientado a objetos es un proceso de toma de decisiones, y cada decisión tiene pros y contras. No existe tal cosa como "la mejor solución" por eso se habla de **heurísticas**: prácticas que nos **pueden** llevar a un **buen** diseño.

Observar cambio en el comportamiento esencial de los objetos

Siempre es importante observar el protocolo (conjunto de mensajes) que comprende cada objeto. Si vemos que está creciendo a una velocidad muy alta ante cada cambio, es probable que ese comportamiento pertenezca a otro/s objetos

No utilizar Null/Nil/Undefined

Si en algún momento nos toca interactuar con este tipo de objetos, estamos en una posición muy desfavorable desde el punto de vista del polimorfismo, ya que no podemos controlar qué mensajes entienden, por representar una noción muy abstracta de algo ausente/inexistente. En general se termina necesitando de condicionales para preguntar si el objeto es de este tipo para hacer algo particular.

Crear objetos Válidos y Completos

El paso más importante de la construcción de un objeto es asegurarse que esté "listo" para ser utilizado. Esto quiere decir que esté inicializado con toda la información necesaria. En la realidad no existen "cosas incompletas" por eso cuando diseñamos nuestros objetos tienen que existir completos, o no existir.

Desaconsejado el uso de Setters

Cada vez que utilizamos un setter, muy probablemente estemos rompiendo con el encapsulamiento del objeto en cuestión. Sí es cierto que es natural que necesitemos generar algún efecto de modificación en un objeto, pero no necesariamente se debe implementar con un setter, sino más bien con algún otro mensaje que capture la esencia de lo que se intenta modificar.

No tomar comportamiento de las colecciones

Las colecciones como cualquier otro objeto, tiene varias responsabilidades que hacen muchas de las tareas que necesitamos más fáciles. No tiene sentido, por ejemplo, iterar usando for/forEach una colección para elegir los elementos que cumplen con una condición; en ese caso, estaríamos hablando de algo que se puede lograr con el mensaje #select: en Smalltalk o filter() en JS.

Identificar objetos polimórficos o intercambiables

Es muy común que en nuestro programa necesitemos interactuar con diferentes objetos pero que comparten una esencia; hay algo en común entre ellos. En ese caso, estamos hablando de objetos polimórficos (o intercambiables), lo que quiere decir que debería ser sencillo y no requerir cambios en el código intercambiar un objeto por otro para obtener un resultado diferente.

Identificar protocolo común mediante jerarquías

Una vez hayamos identificado objetos intercambiables, el próximo paso lógico es enfatizar ese conocimiento y reflejarlo en una jerarquía (puede ser de clases o de objetos concretos según el lenguaje).

Cuestionar uso de condicionales (IF)

Si nos vemos en la necesidad de escribir un IF, podemos hacerlo en una etapa inicial pero debemos volver sobre esa decisión y analizar lo siguiente: cómo un cambio en el dominio afecta ese condicional? debo cambiar la condición o también debo agregar ramas/caminos alternativos? Pensemos que cada condicional tiene 3 cosas, que en POO se traducen a 3 responsabilidades: determinar la condición, qué hacer cuando la condición es verdadera, y qué hacer cuando la condición es falsa. Que un objeto tome esas 3 responsabilidades podría ser una señal de un diseño pobre.

Ocultar detalles de implementación de subclases

La gracia de tener una jerarquía de clases es que no necesitemos depender de ningún mensaje que esté en una subclase o conjunto de subclases, sino que como usuarios, dependamos de mensajes que estén en la superclase. Esto nos permitirá maximizar el polimorfismo.

Mantener número de colaboraciones bajo en métodos

Por una cuestión de división de responsabilidades, es bueno detectar cuando un método está "haciendo demasiado", para o bien delegar en el mismo objeto que tiene el método, o en otro objeto que pueda hacerse cargo de una parte de la responsabilidad. Obviamente el criterio de "demasiado" no es exacto, sino que depende del tipo de tarea que estemos haciendo, del lenguaje/herramienta con la que estemos trabajando, y de qué tan legible o no sea el código. En la práctica métodos con menos de 5 colaboraciones se consideran simples, mientras que los que tienen más de 10 podrían ser señal de que algo puede ser dividido.

1 objeto = 1 clara responsabilidad en el dominio

Cada objeto que definimos existe por una razón y viene a cumplir un único propósito en el dominio. Un objeto que tiene conjuntos de responsabilidades distintos termina siendo poco cohesivo, y es señal de que podríamos en realidad estar necesitando más de un objeto.

Agregados al dominio deberían ser objetos nuevos

Es normal que al desarrollar de manera iterativa e incremental tengamos que ir agregando progresivamente "cosas" nuevas en el dominio (por ejemplo, una regla nueva en el calendario de feriados). Cuando eso ocurre, pueden pasar dos cosas: debo cambiar un objeto existente o agregar un objeto nuevo. Lo segundo es un indicador de buen diseño puesto que estamos respetando la relación 1 a 1 entre algo en la realidad y un objeto.

Nombrar en base a roles

Siempre que tenemos que enviarle un mensaje a un objeto debemos conocerlo a través de un nombre. Este nombre debe representar el rol que cumple el objeto con el que estamos colaborando en el contexto donde estamos. Ejemplo: si podemos saber la ubicación en la que se encuentra un satélite, desde el punto de vista del satélite tiene sentido que hablemos de “ubicacion actual“. Un nombre tipo “ubicacion” se queda corto, mientras que otro tipo “una coordenada” es muy implementativo.