Encapsulamiento

Encapsular, como su nombre lo sugiere, implica la acción de poner juntas ciertas cosas dado que hay una razón para ello. En la POO aquellas cosas serán los datos y los métodos que operan sobre esos datos. *Mediante el encapsulamiento es que creamos las entidades que deseamos manejar en nuestros sistemas.*
Como puede leerse en una de las fuentes:

> "Preguntar '¿qué es el encapsulamiento?' y recibir como respuesta 'se trata de hacer los atributos y los métodos privados', es generalmente una indicación de que si bien los programadores están empleando clases y objetos, esto no significa que estén utilizando Orientación a Objetos".
> Shaun Smith

El encapsulamiento tiene dos sentidos: el de especialización, y el de completitud:

- *Especialización*, ya que el propio objeto es aquél que sabrá cómo manejar los datos que contiene. Es por eso que el mejor lugar para hacerlo, es dentro del mismo objeto (con los métodos asociados a los datos dados).
- *Completitud*, ya que nos permite descansar en que la abstracción construída representa a la entidad, y a aquella responsabilidad que tendrá asignada dentro de nuestro sistema.

Por supuesto no debemos perder de vista que el objetivo de trabajar con objetos es el de aprovechar el pasaje de mensajes y la colaboración entre los mismos: *no nos dejemos tentar por la idea de que programar orientado a objetos es simplemente utilizar estructuras de datos y funciones asociadas.* Es por ello que nos interesa mucho más la interfaz pública de los objetos que sus representaciones internas y datos asociados.

Se define como **interfaz pública** de una clase al conjunto de responsabilidades que los objetos de esa clase estarán brindando, desde el punto de vista externo a la misma. Estas responsabilidades deben ser *cohesivas*, pero eso lo veremos más adelante.

Ocultamiento de la información

El encapsulamiento muchas veces se interpreta como **ocultamiento de la información**, lo que sería incorrecto.

> "El diseñador de cada módulo debe seleccionar un subconjunto de las propiedades del módulo como la información oficial acerca del módulo, para hacerla disponible a los autores de módulos cliente".
> *Bertrand Meyer [MEYER-97]*

Meyer define por contraposición (la interfaz pública) y deja entender que habrá algunas propiedades que no se darán a conocer a los autores de otros módulo.

Meyer también afirma que el ocultamiento de la información nos permite cumplir con el *principio de continuidad*, que al asumir que todo módulo cambiará a lo largo del tiempo prevee que aquellos cambios se concentren en la parte "escondida" del código, sin afectar la interfaz pública (y como consecuencia, no cambiando el código de los módulos cliente).

Muchas veces este ocultamiento de información no podrá ser físico ya que frecuentemente el código será visible y otros programadores podrán leerlo directamente: debemos lograr el ocultamiento lógico, es decir, *que no se pueda* crear un módulo que dependa de información que está oculta en otro módulo.
¿Cómo logramos esto? Buenas prácticas de programación, y una regla fundamental: *al escribir un módulo se deberá depender exclusivamente de la interfaz pública de otros módulos y* ***nunca*** *de los detalles de implementación.*

Encapsulamiento no es oculatamiento de información

El encapsulamiento nos permite armar nuestros objetos, juntando aquellas responsabilidades que necesitan dentro del sistema con los datos para poder hacerlo.
Ocultar la información es utilizar las técnicas que nos brinda el lenguaje para abstraer a nuestros módulos cliente de los detalles de implementación.

El encapsulamiento nos habla de límites: "esta responsabilidad es mía, la llevaré a cabo con estos datos".
El ocultamiento de información nos indica buenas prácticas de programación: "necesito de este otro objeto, pero no me importa cómo resuelva sus responsabilidades mientras lo haga por mí".

Un concepto no implica el otro, ni viceversa. Un ejemplo muy simplificado de cada caso podemos verlo en una de las fuentes, que nos tomamos la licencia de transcribir:

```
class NoEncapsulationOrInformationHiding {
  public ArrayList widths = new ArrayList();
}
```

Podemos ver que nuestra clase simplemente deja visibles los atributos, devela detalles de implementación (como ser que es un ArrayList) y no provee métodos para interactuar con esos datos (la esencia del objeto).

```
class EncapsulationWithoutInformationHiding {
  private ArrayList widths = new ArrayList();
  public ArrayList getWidths(){
    return widths;
  }
}
```

Aquí, en cambio, provee los datos para interactuar, impide el acceso externo

al atributo (los clientes no saben de su existencia) pero, sin embargo, devela detalles de implementación al retornar un ArrayList y permitir, de cierto modo, manipular la colección de *anchos*

```
class InformationHidingWithoutEncapsulation {
  public List widths = new ArrayList();
}
```

En este caso, la implementación se encuentra protegida mediante la abstracción a la interfaz (List), lo que nos permitirá en un futuro cambiar nuestra colección de un ArrayList a, por ejemplo, una LinkedList

```
class EncapsulationAndInformationHiding {
  private ArrayList widths = new ArrayList();
  public List getWidths(){
    return widths;
  }
}
```

Por último, tenemos ambas cualidades, garantizadas por el método que representa la responsabilidad de los objetos, retornando List (suficientemente genérico) y la posesión del atributo privado.

Nota: ¡Cuidado! El manejo de referencias tiene como consecuencia que al retornar la lista original, aquél que la utilice pueda cambiar su contenido. Hay alternativas a esto, como ser el retornar una copia inmutable, no exponer la lista u otorgar un iterador propio en lugar de permitir el acceso al atributo (pero a los efectos de ejemplificar, bien vale el código proporcionado)

Reglas

En el muy recomendable artículo de Wm. Paul Rogers, en JavaWorld, podemos encontrar una serie de reglas para asegurar encapsulamiento y ocultamiento de información.

Encapsulamiento

1. Ubicar los datos y las operaciones que trabajan sobre esos datos en la misma clase.
2. Utilizar diseño guiado por las responsabilidades para determinar la agrupación de datos y operaciones dentro de clases.

Ocultamiento de información

1. No exponer atributos.
2. No exponer diferencia entre atributos propiamente dichos y atributos calculados.
3. No exponer la estructura interna de una clase.

4. No exponer detalles de implementación de una clase.

Getters y Setters

Dado que bajo ciertas circunstancias es necesario instruir a un objeto la necesidad de cambiar algún valor interno del mismo, es importante proporcionar un mecanismo para poder hacerlo sin romper el encapsulamiento (en términos rudimentarios, sin hacer públicos algunos -o todos- sus miembros).
Es por ello que existen dos tipos de métodos muy simples que se denominan **accesores**, y serán los que nos permitan acceder a esos miembros privados *de una manera controlada por el diseñador de la clase.*
Los getters nos servirán para obtener el valor de un miembro, y los setters para establecerlo. Su raíz en el idioma inglés hace juego con las palabras "get" y "set" (obtener y establecer).
Típicamente, un getter tiene esta estructura:

```
public Integer getEdad() {
  return this.edad;
}
```

Un setter, en cambio, responde a la siguiente estructura:

```
public void setEdad(Integer edad) {
  this.edad = edad;
}
```

Cuando estamos a cargo de la definición de una clase, definimos nosotros mismos el nivel de complejidad que estos métodos encapsulan, y qué tan directamente permitimos que un agente externo acceda a los miembros de la misma.

> **Nota:** Para utilizar algunas tecnologías Java, es necesario escribir setters y getters de todos los atributos *persistentes* de una clase. Ya llegaremos a eso, pero por el momento es interesante saberlo.

Tell, don't ask

> El código procedural obtiene información y luego toma decisiones.
> El código orientado a objetos instruye a los objetos para hacer cosas.
> Alec Sharp [SHARP-97]

En este sentido, deberíamos pedirle cosas a los objetos en lugar de preguntarle al respecto de su estado, tomar una decisión en función a ello, y luego decirles qué hacer.
Violaríamos el encapsulamiento al hacerlo de otro modo: si tomásemos decisiones en función del estado de un objeto, estaríamos conociendo *demasiada* información y es muy probable que la decisión que estemos tratando de tomar desde el módulo

cliente corresponda al objeto consultado en lugar del módulo que estamos escribiendo.

Con esto volvemos a los conceptos de **comandos** y **consultas** que conocemos de capítulos anteriores, y la preferencia de los primeros por sobre los segundos.

Veamos un ejemplo práctico de lo que queremos decir. Podríamos tener el siguiente código (correcto, pero póbremente diseñado):

```
public void verificarSobrecalentamiento(Monitor monitor) {
  if (monitor.getTemperatura() > 100) {
    monitor.sonarAlarmas();
  }
}
```

En cambio, podríamos reescribir esa pieza teniendo en cuenta los conceptos que acabamos de volcar:

```
public class Monitor {
  public void verificarSobrecalentamiento() {
    if (this.temperatura > 100) {
      this.sonarAlarmas();
    }
  }
  //...
}
```

Como podemos ver, la decisión de cuánto se considera "sobrecalentamiento" corresponde más al monitor que al cliente. Adicionalmente, éste tiene toda la información y los elementos necesarios para llevar a cabo esta responsabilidad: vigilar el sobrecalentamiento, y sonar las alarmas si corresponde.

Recursos

Bibliografía

- [MEYER-97] **Object-oriented software construction** *Bertrand Meyer* - Prentice Hall PTR - 1997
- [SHARP-97] **Smalltalk by example: the developer's guide** *Alec Sharp* - McGraw Hill - 1997

Links

- Encapsulation Definition - c2 wiki
- Encapsulation is not information hiding - JavaWorld
- Encapsulation is not information hiding - c2 wiki
- Tell, don't ask - The Pragmatic Bookshelf

- Tell, don't ask - Thoughtbot
- Tell, don't ask - c2 wiki

Por Lucas Videla. Versión original disponible en delucas/notas