
# TP 1

Programación Orientada a Objetos II
Comisión 2
2º Cuatrimestre de 2024

Nicolás Salvanescki

# 1 - Evaluación de protocolos de una clase

Elegiría la opción 2, ya que presenta al menos dos grandes ventajas. En primer lugar, las tres esquinas que no se definen explícitamente con los métodos, pueden calcularse a partir de la esquina superior izquierda (punto origen) y el ancho y alto del rectángulo. Además, tener el método reubicarEsquinaSuperiorIzquierdaEn permite reubicar el rectángulo de manera automática, ya que se calculan las esquinas automáticamente. Esto evita posibles errores que podrían darse al calcular y setear cada esquina individualmente. Por ejemplo, en un desplazamiento al calcular mal las esquinas podría quedar una figura que no sea un rectángulo.

# 2 - Delegación

La opción 2 parece una mejor alternativa, ya que rompe un poco menos el encapsulamiento. En la opción 1, el Jefe conoce el fichero del Secretario y utiliza el método buscar sobre el mismo, en vez de delegar esta responsabilidad al Secretario. En la opción 2, en cambio, el Jefe no conoce el fichero, es el Secretario, proveyendo un método buscarEnFichero, quien se encarga de operarlo.

# 3 - Polimorfismo

Los defectos en cada opción son:

1. La opción 1 utiliza una única clase CuentaBancaria, lo cual impide usar polimorfismo. En cambio, para saber en qué tipo de cuenta se quiere extraer, se usa un condicional. Dicho condicional, si se cumple, modifica la variable local rojo, esto supone un problema para la legibilidad del código.
   Otro problema es que se utiliza el estilo Allman en un lenguaje como wollok que tiene por estándar el Estilo K&R (Kernighan & Ritchie).

2. La opción 2 es muy similar a la opción 1. En este caso lo que varia es que se crea un método en CuentaBancaria llamado getTipo que supondré que utiliza un condicional para validar el tipo de cuenta. Esto es el mismo problema que antes, solo que se delega en un método con condicionales que validan strings, en vez de usar directamente el método class.
   Acá ya se utiliza el estilo K&R que corresponde al usado en wollok.

3. En la opción 3 ya se aplica polimorfismo, CuentaBancaria solo aporta el protocolo (interfaz) del método extraer, son sus clases hijas CajaDeAhorro y CuentaCorriente las que aplican comportamiento al mismo. Sin embargo, en ambos métodos extraer de las hijas, se hace el chequeo del saldo disponible además de la extracción, cuando puede ser una mejor práctica delegar el chequeo en un método distinto y llamarlo en el condicional.
   Además, hay redundancia de código al setear el saldo, ya que es exactamente la misma línea de código en ambos casos "self.setSaldo(self.getSaldo() – unMonto)". Esto se podría solucionar, llevando dicha línea a la clase CuentaBancaria, pero para eso hay que hacer la validación previa allí también.

4. La opción 4 es la mejor alternativa por lo mencionado anteriormente. Se creó un método para chequear los saldos, lo cual permite a las clases hijas sobreescribir solo ese método con su propio comportamiento. Toda la lógica que antes era redundante, está implementada en el método extraer de CuentaBancaria.

# Actividad de Lectura #1

Voy a usar más que el solo el texto para responder las preguntas.

1. El acceso directo a las variables es, como lo indica su nombre, la interacción con los atributos de forma directa, es decir, sin métodos de por medio. Esto solo se puede realizar en lenguajes de programación que permiten ser públicos a los atributos, de otra forma no se podrían acceder así. Un ejemplo de acceso directo a variables sería:

```
// Si tenemos la siguiente clase
class 3DVector {
    public int x
    public int y
    public int z

    public 3DVector(int x, int y, int z){
        this.x = x
        this.y = y
        this.z = z
    }
}

3DVector unVector = new 3DVector(3,4,5)

// Podemos acceder directamente a sus atributos
unVector.x // -> 3

// Tambien cambiar su valor
unVector.x = 10
```

2. El acceso indirecto a las variables es interactuar con los atributos a través de métodos llamados getters y setters. Los primeros obtienen el valor, mientras que los segundos lo establecen y/o cambian. Se pueden utilizar getters y setters sin importar la visibilidad de los atributos, es decir que pueden ser privados, protegidos o públicos, y los getters/setters seguirán cumpliendo su función. Siguiendo con el ejemplo anterior:

```
// Si tenemos la siguiente clase
class 3DVector {
    private int x
    private int y
    private int z

    public 3DVector(int x, int y, int z){
        this.x = x
        this.y = y
        this.z = z
    }

    //getters

    public int getX(){
        return this.x;
    }

    public int getY(){
        return this.y;
    }

    public int getZ(){
        return this.z;
    }

    //setters
```

```
28
29      public int setX(int x){
30          this.x = x
31      }
32
33      public int setY(int y){
34          this.y = y
35      }
36
37      public int setZ(int z){
38          this.z = z
39      }
40  }
41
42  3DVector unVector = new 3DVector(3,4,5)
43
44  // Para acceder a los valores, ahora usamos el getter correspondiente
45  unVector.getX() // -> 3
46
47  // Para cambiar el valor, el setter correspondiente
48  unVector.setX(10)
```

3. Las ventajas y desventajas que presenta cada estrategia son:

   Acceso Directo:

   Ventajas:

   - Al no tener que definir métodos adicionales, requiere escribir menos código.
   - En mi opinión personal, se lee más fácil el código (ya que no hay que escribir los paréntesis del getter, y al setear el valor es como una asignación).
   - Puede que sea imperceptiblemente más rápido al apilar menos cantidad de métodos en el stack, ya que el getter/setter que estés llamando dentro de un método va a apilar por un momento su propio frame.

   Desventajas:

   - Expone las variables internas, violando el encapsulamiento. Esto también puede llevar a un acoplamiento mayor, es decir que dependan mucho los módulos entre sí y no puedas cambiar uno sin cambiar el otro, y menos control sobre como se modifica el estado del objeto.
   - Lo que mencioné en el item anterior lleva a que el código sea muy difícil de mantener y escalar.

   Acceso Indirecto:

   Ventajas:

   - Encapsulamiento.
   - Permite agregar validaciones/conversiones en los métodos getters/setters, y, por ende, un mayor control de acceso.
   - Flexibilidad, ya que te permite modificar la estructura interna de la clase sin impactar a las partes que la utilicen.
   - Mayor Escalabilidad y Mantenibilidad, a causa de lo anteriormente mencionado.

   Desventajas:

- Complejiza el código de la clase, al agregar los getters/setters.
- Como mencioné anteriormente, puede afectar un poco el rendimiento.

## Actividad de Lectura #2

Según entendí leyendo el fragmento, es conveniente utilizar dicho método cuando en los métodos setter de las variables de instancia hay cierta lógica extra, ya que en el método constructor, al tener que establecer el valor inicial de dichas variables, hay que hacer lo mismo que en el setter. Por lo tanto, la solución sería llamar al setter en el constructor.

## Actividad de Lectura #3

Como dejar un getter a la colección sería exponer demasiado la implementación de la colección y podría traer problemas de consistencia con otras variables (como contadores al borrar elementos), una solución sería limitar el acceso y uso del usuario a la colección creando métodos que deleguen en los de la colección. Por ejemplo, crear un método para agregar elementos a la colección que, dentro, llame al método de agregar elementos propio de la colección.

El único problema que tiene esta solución es que hay que definir más métodos que si solo expusieras la colección con un getter. Pero, aún así permite un acceso a las colecciones más controlado, limpio y fácil de mantener.

## Actividad de Lectura #4

Hay dos razones por las cuales tener dos métodos que asignen el estado a una propiedad booleana es mejor que tener solo un toggle.

La primera razón es que, en el caso del toggle, es muy difícil saber quien y en que parte del código, asigno en valor en False. Esto sucede porque no se explicita cuando se cambia al valor False, sino que solo sabes que cambió de un valor a otro.

La segunda razón es que se muestra parte de la implementación al usuario. Si en un futuro se tuviese que cambiar la implementación, habría que modificar la firma del toggle. Esto se puede evitar al usar dos métodos, ya que no es necesario pasar un parámetro, solo llamar al método, es decir, se puede abstraer de la implementación.