Portafolio 1

Implementación de clases.

Jose Collado San Pedro

Sombrero Verde

Recomendaciones interesantes.

- El nombre de una clase debe describir las responsabilidades que le corresponden.
- Debemos mantener la privacidad en la medida de lo posible. Si un método necesita ser accedido por otra clase en el mismo paquete, lo declararemos *protected*.
- Una clase puede tener **una sola responsabilidad**, o como dice el *The Single Responsibility Principle*, solo debería tener una razón para cambiar.
- Esto es una clase con dos responsabilidades. No cumpliría con el principio anterior.

```
public class Portafolio {
public void setTitle(String title);
public int getBuildNumber();
}
```

- Un concepto interesante es el de **cohesión**. Una clase no debería tener muchas variables declaradas. Cuantas más variables de la clase use un método de ella, mas cohesionado es ese método respecto la clase.
- Una clase donde cada variable es usada por todos los métodos de ella es una clase cohesionada al máximo.
- Tener muchas clases pequeñas no hace que el sistema más fácil de entender. ¿Es preferible tener las herramientas ordenadas en cajones bien etiquetados o todas en un mismo cajón?
- A veces romper una función en otras más pequeñas nos puede forzar a tener que partir también la clase, pero esto es bueno.
- Una clase debe estar **abierta para el cambio**. Por eso, son buenas las clases cortas y con una sola responsabilidad.
- El **aislamiento** hace más fácil la comprensión de las clases y aumenta su facilidad al cambio en un momento dado.
- Un ejemplo útil sobre aislamiento podría ser crear una clase *Interface*, otra clase que implemente los métodos de la interfaz y la clase que va a usar esos métodos, en vez de depender de la clase que implementa, si depende solo de la interfaz aislamos los detalles específicos de la implementación de los métodos.
- Ten en cuenta que una clase puede sufrir erosión. Por culpa de cambios que vamos añadiendo, la clase puede perder el sentido inicial que tenía.

Opiniones al respecto.

Según sugieren ambos libros en diversas ocasiones, el uso de la herencia es preferible ante una extensa comprobación de tipos, ya que nos puede ahorrar mucho código a la vez que hacemos que sea más limpio y legible. Pero no siempre ocurre esta mejora, y lo podemos observar en el siguiente ejemplo basado en uno similar del libro *Code Complete*.

```
switch (futbolista.getPosicion()) {
    case "Defensa":
        futbolista.defender();
        break;

    case "Delantero":
        futbolista.marcaGol();
        break;
    ...
    }
```

En este caso vemos claramente que usando polimorfismo podríamos tener un solo método *realizarFuncion()* y que cada clase que hereda de una padre llamada Futbolista sobrescriba dependiendo del tipo de función que desempeña el jugador en el campo. Aquí vemos como si que es bastante útil y no ahorraría bastante código, a parte de hacerlo más simple y legible. Sin embargo, vamos a ver un ejemplo que nos demuestra que usar el polimorfismo no siempre nos va a aportar mejoría.

```
switch (ui.Command()) {
    case Command_OpenFile:
      OpenFile();
      break;
    case Command_Print:
      Print();
      break;
    case Command_Save:
      Save();
      break;
    case Command_Exit:
      ShutDown();
      break;
...
  }
```

Si usamos herencia aquí, deberíamos crear clases que heredasen de una clase *Command* y sobrescribir cada método *DoCommand()*. De esta forma, el método perdería totalmente su significado y sería imposible de entender su funcionamiento sin leer detenidamente el código del método.