Optimización del diseño de objetos

Ing. Luciano Straccia

Optimización del diseño

Optimizar no es hacerlo mejor en términos de mantenibilidad o performance (atributos en ocasiones considerados los más relevantes), sino mejorarlo para el cumplimiento de nuestros atributos de calidad del software (¡el atributo que sea!)

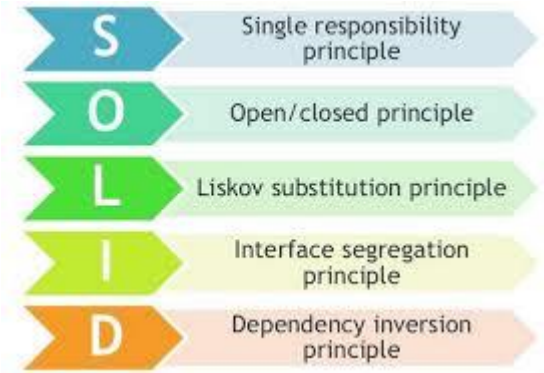
- Deuda técnica
- Principios SOLID
- Patrones de diseño
- Antipatrones de diseño
- Bad Smells

Deuda técnica

- Término acuñado en Ward Cunningham en 1992
- Es una analogía con la petición de un crédito bancario para atender aspectos de corto plazo, pero que el pago de intereses impacta a largo plazo

Principios solid

- Permiten mejorar el diseño
- SOLID =
  - Single Responsability
  - Open Closed
  - Liskov Substitution
  - Interface Segregation
  - Dependency Inversion



- Ver PPT adicional

Patrones de diseño

PATRONES DE DISEÑO

- Un patrón de diseño nomina, abstrae e identifica los aspectos clave de una estructura de diseño común, lo que los hace útiles para crear un diseño orientado a objetos reutilizable.
- Características:
  - Capturan la experiencia y la hacen accesible a los no expertos
  - El conjunto de sus nombres forma un vocabulario que ayuda a que los desarrolladores se comuniquen mejor (lenguaje de patrones)
  - Facilitan la reestructuración de un sistema tanto si fue o no concebido con patrones en mente
  - Reutilización
  - Los patrones pueden ser la base de un manual de ingeniería de software

TIPOS DE PATRONES DE DISEÑO

- Creacionales: Relacionados al proceso de creación de objetos
- Estructurales: Relacionados con la composición de objetos y clases
- De comportamiento: Como interactúan y se reparten responsabilidades los objetos y clases

Catálogo

Tabla 1.1: Patrones de diseño				
		Propósito		
		De Creación	**Estructurales**	**De comportamiento**
Ámbito	**Clase**	Factory Method (99)	Adapter (de clases) (131)	Interpreter (225) Template Method (299)
	Objeto	Abstract Factory (79) Builder (89) Prototype (109) Singleton (119)	Adapter (de objetos) (131) Bridge (141) Composite (151) Decorator (161) Facade (171) Flyweight (179) Proxy (191)	Chain of Responsibility (205) Command (215) Iterator (237) Mediator (251) Memento (261) Observer (269) State (279) Strategy (289) Visitor (305)

Antipatrones de diseño

Antipatrones de diseño

- Según (Brown et al, 1998) “Un antipatrón es una forma literaria que describe una solución recurrente que genera consecuencias negativas”.
- Analizando parte por parte esta definición se entiende:
  - Forma literaria: descripciones de problemas, no de código.
  - Recurrente: si no es un patrón, entonces no es un antipatrón. Se deben establecer diversas ocurrencias del mismo comportamiento erróneo preferentemente en diversos contextos.
  - Consecuencias negativas: el diseño debe producir impacto negativo

Antipatrones de diseño

- The Blob, God Class o God Object:
  - concentrar demasiada funcionalidad en una única parte del diseño (clase)
- Lava Flow o Dead Code:
  - presencia de código (o clases) ya inservible
- Poltergeists (Proliferación de clases):
  - las clases poltergeists (fantasmas) se caracterizan por tener pocas responsabilidades dentro del sistema y un ciclo de vida bastante breve, ya que “aparecen” solamente para iniciar algún método en alguna clase, a menudo en un determinado orden. Son de relativa facilidad de encuentro ya que sus nombres suelen llevar el sufijo “controller” o “manager”.

Antipatrones de diseño

- Golden Hammer (Silver Bullet):
  - se da cuando un equipo consigue un alto nivel de competencia en una determinada solución/tecnología, lo que hace que cada nuevo desafío que se tenga que afrontar sea mejor resuelto con la herramienta ya conocida.
- Spaghetti Code
- YAFL (yet another fucking layer, y otra maldita capa más) o 'Código Lasagna':
  - añadir capas innecesarias a un programa, biblioteca o framework. Esta tendencia se extendió bastante después de que se publicase el primer libro sobre patrones.
- Copy-And-Paste Programming
- Objeto sumidero (object cesspool):
  - reutilizar objetos no adecuados realmente para el fin que se persigue.

Antipatrones de diseño

- Problema del yoyó (yo-yo problem):
  - construir estructuras (por ejemplo, de herencia) que son difíciles de comprender debido a su excesiva fragmentación.
- Functional Decomposition o No object oriented AntiPattern:
  - Cuando desarrolladores experimentados en programación estructurada, quienes se hallan cómodos con una rutina principal (main) que llama a diversas subrutinas para realizar el trabajo, diseñan sistemas orientados a objetos suelen traducir sus diseños estructurados a orientados a objetos: traducen cada subrutina como una clase.
- Singletonitis:
  - abuso de la utilización del patrón singleton.

BAD SMELLS

Bad smells

- Code Smells (o Bad Smells) –olor en el código o mal olor- se refiere a síntomas en el código fuente (probablemente con origen en problemas de diseño) que posiblemente indica un problema

- Tipos de Bad Smells:

  - Bloaters
  - Object-Orientation Abusers
  - Change Preventers
  - Dispensables
  - Couplers

Bad smells

- Bloaters: son código, métodos y clases que han aumentado a proporciones gigantescas con las que son difíciles de trabajar. Por lo general no aparecen de inmediato, sino que se acumulan con el tiempo a medida que el programa evoluciona (y especialmente cuando nadie hace un esfuerzo para erradicarlos).
  - Long Method: código spaghetti
  - Large Class: una clase contiene muchos atributos, métodos o líneas de código
  - Primitive Obsession: representar con int, boolean, string o enumeraciones cosas que podrían ser objetos con comportamiento. (por ejemplo, código postal o direcciones; en ocasiones se definen como string y luego se realizan múltiples validaciones en lugar de crear los objetos pertinentes)
  - Long Parameter List: más de 3 o 4 parámetros para un método
  - Data Clumps (Aglomeraciones): diferentes partes del código contienen grupos idénticos de variables (x, y está representando un objeto Point; calle, altura, piso, departamento, etc. forman parte de un objeto Dirección; dni, nombre puede representar a una Persona). Estos grupos deben convertirse en sus propias clases.

Bad smells

- Object-Orientation Abusers: aplicación incompleta o incorrecta de los principios de programación orientados a objetos.
  - Switch Statements: «switch» complejo o secuencias de «if»
  - Temporary Field: campos que son utilizados sólo en algunas circunstancias, pero mayormente están vacíos (empty) (por ejemplo campos calculables como total de facturación de una persona, teniendo una clase Factura)
  - Refused Bequest: una subclase utiliza sólo algunos de los métodos y propiedades heredados. Esto puede ocurrir cuando se crea una subclase para reutilizar código de la superclase, pero clase y subclase son completamente diferentes.
  - Clases alternativas con interfaz diferente: dos clases tienen funciones idénticas, pero sus métodos llevan nombres diferentes. Generalmente ocurre cuando el programador que creó una de las clases no sabía que existía otra clase funcional equivalente.

Bad smells

- Change Preventers: si se necesita cambiar algo en un lugar en su código, se tienen que hacer muchos cambios en otros lugares también. El desarrollo del programa se vuelve mucho más complicado y costoso como resultado.
  - Divergent Change: hay que cambiar muchos métodos no relacionados cuando se hacen cambios en una clase. Por ejemplo, al agregar un nuevo tipo de producto, debe cambiar los métodos de búsqueda, visualización y pedido de productos.
  - Shotgun Surgery («Cirugía de escopeta»): al realizar un cambio en un método de una clase implica realizar modificaciones en otras clases
  - Jerarquías paralelas de herencia: cada vez que se crea una subclase para una clase es necesario crear una subclase para otra clase. Ejemplo: una clase Vehículo con subclases Auto y Aeroplano; y una clase Operador con Conductor y Piloto.

Bad smells

- Dispensables: algo inútil e innecesario cuya ausencia haría el código más limpio, más eficiente y más fácil de entender
  - Comments: comentarios explicativos sobre código que no son necesarios (recordar que «el mejor comentario es un buen nombre para el método o clase»)
  - Código duplicado
  - Lazy Class («clase perezosa»): clases difíciles de entender (generalmente por sus modificaciones a lo largo del tiempo) que ameritan ser eliminadas y vueltas a construir
  - Data Class: clases que sólo tienen atributos y métodos GET y SET. Resultan sólo contenedores para datos utilizados para otras clases, sin funcionalidad adicional ni pueden operar independientes.
  - Dead Code: código en desuso
  - Speculative Generality: clase, método o atributo en desuso

Bad smells

- Couplers: contribuyen a un acoplamiento excesivo entre las clases o muestran lo que sucede si se reemplaza el acoplamiento por delegación excesiva.
  - Feature Envy: un método accede a datos de otros objetos

```
>>Carta
public String getDestinatario() {
   return persona.getNombre() + " " + persona.getDireccion() + " "
           + persona.getEdad();
}
```

La clase Carta está pidiendo demasiada info a la persona, entonces debería ser responsabilidad de la persona devolver el domicilio completo:

```
>>Carta
public String getDestinatario() {
   return persona.asDestinatario();
}

>>Persona
public String asDestinatario() {
   return this.getNombre() + " " + this.getDireccion() + " " +
this.getEdad();
}
```

Bad smells

- Inappropiate Intimacy: una clase utiliza campos y métodos de otra clase
- Message Chains («cadena de mensajes»): ocurre cuando un cliente requiere un objeto y este objeto requiere otro y así sucesivamente: $a->b()->c()->d()
- Middle Man (Intermediario): una clase realiza una sola acción que es delegar trabajo a otra clase
- Incomplete Library Class: una librería (de sólo lectura, por ser externa) no cumple con las necesidades de nuestro código

FIN