5-12-2016

INFORME

TRABAJO GRUPAL

PRINCIPIOS SOLID

TALLER DE PROGRAMACIÓN

INTEGRANTES:

Delgado Soto, Renato

Flores Panaifo, Josselyn

**INDICE**

1. Resumen
2. Antecedentes
3. Fundamentos Teórico
4. Caso Demostrativo
5. Discusión
6. Conclusiones
7. Recomendaciones
8. Bibliografía utilizando norma APA

**RESUMEN**

Comenzamos con el último de los principios S.O.L.I.D., el de inversión de dependencias.

Fue postulado inicialmente por Robert C. Martin (Uncle Bob) a mediados de los noventa en un artículo de diseño orientado a objetos en C++.

Es la base teórica sobre la que se basa la inyección de dependencias usada por muchos frameworks, como Spring.

Fue un concepto bastante innovador, porque supuso romper con el diseño conocido hasta esa fecha, darle la vuelta a las dependencias entre objetos para reducir su acoplamiento y que puedan reutilizarse y cambiarse más fácilmente.

Son cinco principios fundamentales, uno por cada letra, que hablan del diseño orientado a objetos en términos de la gestión de dependencias. Las dependencias entre unas clases y otras son las que hacen al código más frágil o más robusto y reutilizable. El problema con el modelado tradicional es que no se ocupa en profundidad de la gestión de dependencias entre clases sino de la conceptualización.

**ANTECEDENTES**

Robert C. Martín nos propuso en el año 2000 unos principios para que nuestro código orientado a objeto sea más flexible. Estos principios están recogidos sobre el acrónimo SOLID.

**FUNDAMENTOS TEORICOS**

En ingeniería de software, SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion) es un acrónimo mnemónico introducido por Robert C. Martin, a comienzos de la década del 2000 que representa cinco principios básicos de la programación orientada a objetos y el diseño. Cuando estos principios se aplican en conjunto es más probable que un desarrollador cree un sistema que sea fácil de mantener y ampliar en el tiempo. Los principio SOLID son guías que pueden ser aplicadas en el desarrollo de software para eliminar código sucio provocando que el programador tenga que refactorizar el código fuente hasta que sea legible y extensible. Debe ser utilizado con el desarrollo guiado por pruebas o TDD, y forma parte de la estrategia global del desarrollo ágil de software y programación adaptativa.

**SOLID** representa cinco principios básicos de la programación orientada a objetos y el diseño:

**1. Single Responsability Principle (Principio Responsabilidad Única)**

Cada clase debe tener una única responsabilidad. Una clase debe tener una, y solo una, razón de cambio, también aplica a nivel de métodos de una clase.

**2. Open Closed Principle (Principio Abierto/Cerrado)**

Una clase debe ser abierta para extender, pero cerrada para modificar. Uno debe ser capaz de extender el comportamiento de una clase, sin modificar su contenido.

**3. Liskov Substitution Principle (Principio Sustitución Liskov)**

Una clase derivada puede ser reemplazada por cualquier otra que use su clase base sin alterar su correcto funcionamiento.

**4. Interface Segregation Principle (Principio Segregación de Interfaz)**

Define interfaces pequeñas que resuelvan un problema específico (Role Interface) en lugar de tener interfaces grandes que hagan muchas cosas (Head Interface). Los clientes no deben ser forzados a implementar interfaces que ellos no necesiten.

**5. Dependency Inversion Principle (Principio Inversión de Dependencia)**

Los módulos de alto nivel no deben depender de los módulos de bajo nivel ambos deben depender de abstracciones (Interfaces, Clases abstractas), no de clases concretas.

**CUÁNDO USAR SOLID:**

SOLID nació como una reacción a un conjunto de design smells. Por design smells hablamos de síntomas, introducidos en el diseño, que indican la violación de algún principio del diseño orientado a objetos. Se recomienda aplicar los principios SOLID si tu código sufre alguno de los siguientes síntomas:

1. Rigidez:

Es la tendencia a que el software sea difícil de cambiar, aun con cambios simples. Ocurre cuando un simple cambio causa que se tengan que volver a compilar los módulos dependientes. Mientras más módulos se tengan que cambiar, más rígido es el diseño. A medida que la rigidez de un módulo aumenta los tiempos asignados para los cambios también aumentan.

2. Fragilidad:

Es la tendencia a que el software se rompa en muchas partes cuando se hace cambio. Ocurre cuando un cambio hecho en un módulo externo genera problemas en lugares que no tienen relación. A medida que la fragilidad de un módulo aumenta la probabilidad que un cambio introduzca nuevos problemas también aumenta.

3. Inmovilidad:

Significa que el diseño de un sistema es difícil de reutilizar. Ocurre cuando un sistema contiene partes que pueden ser reutilizadas por otros, pero el esfuerzo para separarlas es muy grande. A medida que la inmovilidad de un sistema aumenta la probabilidad de volver a cometer los mismos errores también aumenta.

4. Viscosidad:

El diseño hace que sea difícil hacer lo correcto a nivel de software y ambiente de desarrollo. Seguir la estrategia propuesta versus aplicar soluciones alternativas. Ocurre cuando es más fácil hacer mal las cosas (usar soluciones alternas) que hacerlas bien (seguir la estrategia propuesta). A medida que la viscosidad de un sistema aumenta ocasiona que existan muchas formas de hacer lo mismo.

5. Complejidad Innecesaria: (El más importante de todos)

Son los elementos que no se utilizan actualmente en el diseño y fueron agregados para soportar necesidades que no existen, ni van a existir. Ocurre cuando se quiere desarrollar una librería de propósito general y se empieza a soñar con cualquier tipo de cosas extrañas que probablemente nunca se van a necesitar. A medida que la complejidad aumenta el código se vuelve más difícil de entender.

6. Repetición innecesaria:

Usar Ctrl+v y Ctrl+c a través de todo el sistema. Ocurre cuando al corregir un bug se tiene que buscar en toda la solución para encontrar cada implementación hecha con el mismo código. A medida que el código repetido aumenta en un módulo se vuelve más complicado darle mantenimiento.

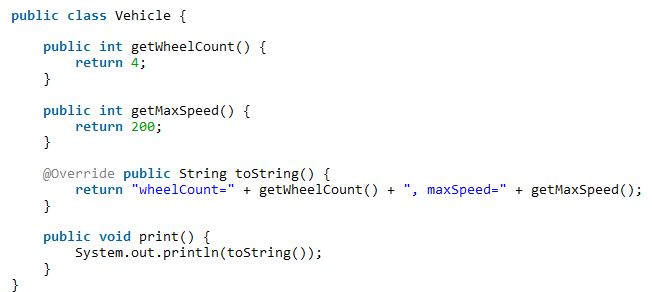
7. Opacidad:

La tendencia de un módulo a ser difícil de entender. Ocurre cuando el código evoluciona con el tiempo y se vuelve más difícil de entender. A medida que la opacidad aumenta en un sistema se vuelve más complicado darle mantenimiento y agregar nuevas funcionalidades.

**CASO DEMOSTRATIVO**

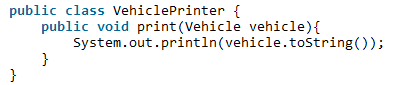
# **Principio de Responsabilidad Única**

Un ejemplo típico es el de un objeto que necesita ser renderizado de alguna forma, por

Ejemplo, imprimiéndose por pantalla. Podríamos tener una clase como esta:

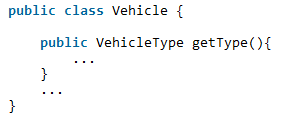
Aunque a primera vista puede parecer una clase de lo más razonable, en seguida podemos detectar que estamos mezclando dos conceptos muy diferentes: la lógica de negocio y la lógica de presentación.

Donde la solución simple para el caso mostrado seria implementar una clase que se encargue de imprimir:



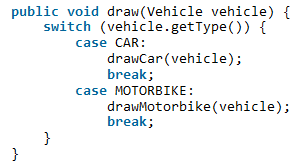
# **Principio Open/Closed**

Imaginemos que tenemos una clase con un método que se encarga de dibujar un vehículo por pantalla. Por supuesto, cada vehículo tiene su propia forma de ser pintado. Nuestro vehículo tiene la siguiente forma:

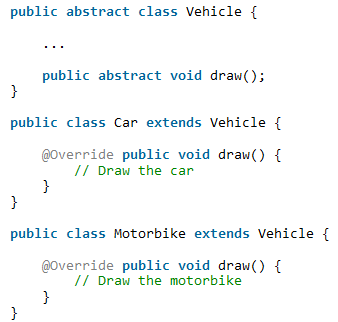


Básicamente es una clase que especifica su tipo mediante un enumerado. Podemos tener por ejemplo un **enum** con un par de tipos:



Y éste es el método de la clase que se encarga de pintarlos:

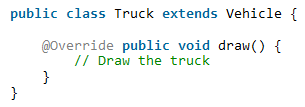
Si lo solucionamos mediante herencia y polimorfismo, el paso evidente es sustituir ese enumerado por clases reales, y que cada clase sepa cómo pintarse:



Ahora nuestro método anterior se reduce a:

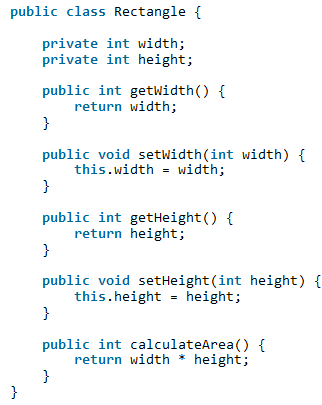


Añadir nuevos vehículos ahora es tan sencillo como crear la clase correspondiente que extienda de **Vehicle**:

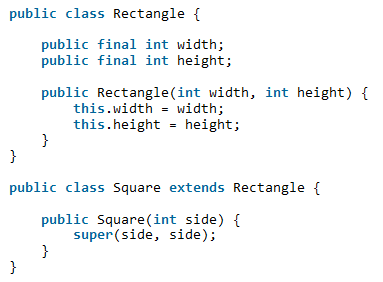


# **Principio de Sustitución de Liskov**

Si intentamos modelar un cuadrado como una concreción de un rectángulo, vamos a tener problemas con este principio:



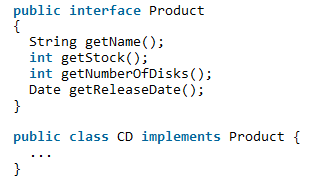
Aplicando el principio de sustitución, consideramos el siguiente código:



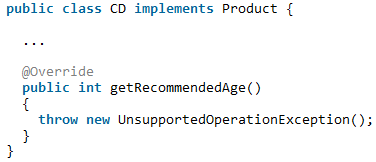
Desde el momento de la instanciación del objeto, todo lo que hagamos con él será válido, ya usemos un rectángulo o un cuadrado.

# **Principio de Segregación de Interfaces**

Imagina que tienes una tienda de CDs de música, y que tienes modelados tus productos de esta manera:



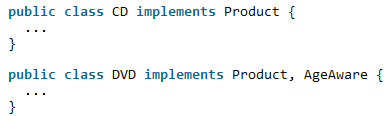
Nos vemos obligados a implementar **getRecommendedAge(),** pero no van a saber qué hacer con ello, así que lanzarán una excepción:



Con todos los problemas asociados que hemos visto antes. Además, se forma una dependencia muy fea, en la que cada vez que añadimos algo a **Product,** nos vemos obligados a modificar CD con cosas que no necesita. Podríamos hacer algo tal que así:



Y ahora nuestra clase DVD implementará las dos interfaces:

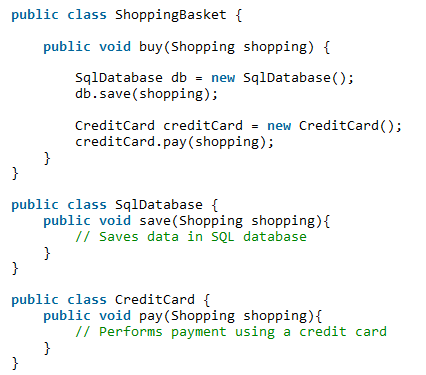


La ventaja de esta solución es que ahora podemos tener código **AgeAware**, y todas las clases que implementen esta interfaz podrían participar en código común.



# **Principio de Inversión de Dependencias**

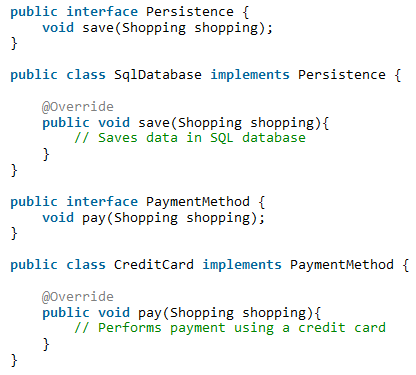
Imaginemos que tenemos una cesta de la compra que lo que hace es almacenar la información y llamar al método de pago para que ejecute la operación. Nuestro código sería algo así:



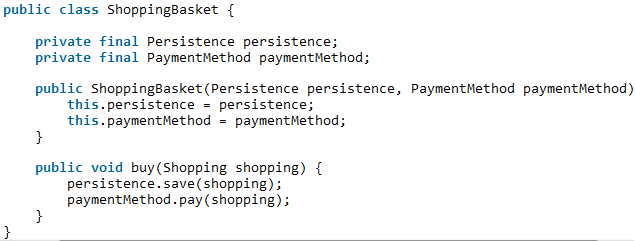
Aquí estamos incumpliendo todas las reglas que impusimos al principio. Una clase de más alto nivel, como es la cesta de la compra, está dependiendo de otras de alto nivel, como cuál es el mecanismo para almacenar la información o para realizar el método de pago.

**Solución**.

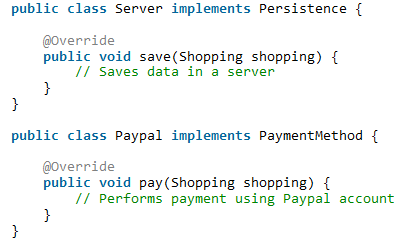
Vamos a crear interfaces que definan el comportamiento que debe dar una clase para poder funcionar como mecanismo de persistencia o como método de pago:



Nuestro segundo paso es invertir las dependencias. Vamos a hacer que estos objetos se pasen por constructor:



Definimos las concreciones específicas para este caso, y se las pasamos por constructor a la cesta de la compra:



**DISCUSIÓN**

Estos principios se discuten a menudo en el contexto de lenguajes clásicos, de tipado estático, y orientados a objetos, y aunque JavaScript es un lenguaje basado en prototipos y de tipado dinámico, tiene mezcla de ambos paradigmas, el orientado a objetos y el funcional, por lo que los programadores aún puede cosechar los beneficios de la aplicación de estos principios a JavaScript. El primero de estos principios: el principio de responsabilidad única.

El principio de responsabilidad única se refiere a la relación funcional de los elementos de un módulo. El principio declara:

Una clase debe tener sólo una razón para cambiar.

Indica que un objeto sólo debe hacer una cosa. Sin embargo, lo que quiere decir esta afirmación es que un objeto debe tener un conjunto coherente de comportamientos entorno a una única responsabilidad de tal forma que si esta responsabilidad cambia implica cambiar la definición del objeto. De una forma más simple, la definición de un objeto sólo debería tener que ser modificada debido a cambios en una única responsabilidad dentro del sistema.

La adhesión al principio de responsabilidad única ayuda a mejorar la mantenibilidad limitando las responsabilidades de un objeto a únicamente aquellas que cambien por razones relacionadas. Cuando un objeto encapsula múltiples responsabilidades, los cambios en una de las responsabilidades del objeto pueden afectar negativamente a las otras.

**CONCLUSIONES**

Al mencionar SOLID hablamos de un conjunto de principios, no de ningún framework o librería. SOLID no está amarrado a ninguna tecnología y puede ser usado en cualquier lenguaje de programación. SOLID son 5 principios: Responsabilidad única, Open/Closed, Sustitución Liskov, Segregación de interfaces e Inyección de dependencias. Sirve aplicar SOLID solo cuando suframos de alguno de estos síntomas en el diseño: Rigidez, Fragilidad, Inmovilidad, Viscosidad, Sobre diseño, Repetición innecesaria u Opacidad.

**RECOMENDACIONES**

**Recomendaciones para prevenir los design smells:**

* Tomar decisiones de arquitectura junto a otros compañeros.
* Asegurar que las partes más críticas del sistema cuenten con pruebas unitarias.
* Contar con una guía de buenas prácticas de codificación para el equipo.
* Para algún cambio o una nueva funcionalidad, se debe invertir tiempo en refactorizar el código.
* Hacer programación en pares cuando se implemente un cambio complejo.
* Revisar el código de los demás.
* Lo perfecto es el enemigo de lo bueno, no tratar de crear diseños perfectos que con llevan a una complejidad innecesaria.

**BIBLIOGRAFIA**

* sábado, 3 de marzo de 2012 - el blog de adrián m. paredes - arquitectura de software
* september 29, 2016 by arthur antunes-solid, 5 principios básicos de la programación orientada a objetos/bemobile
* 2010-2013 carlos ble -diseño orientado a objetos - librow web.