Curso de desarrollo de software

Entendiendo los principios SOLID

Actividad grupal: 2 horas

Debes construir una carpeta llamada Actividad 12-CC3S2 en github y subir el proyecto desarrollado.

Java es un poderoso lenguaje de programación orientado a objetos y tiene muchas características. Si lo comparamos con los viejos tiempos, debemos decir que la codificación se ha vuelto más fácil con el apoyo de estas potentes funciones. Pero la dura verdad es que el simple uso de estas funciones en una aplicación no garantiza que las haya utilizado de la manera correcta. En cualquier requisito dado, es vital identificar clases, objetos y cómo se comunican entre sí. Además, tu aplicación debe ser flexible y extensible para cumplir con futuras mejoras. Este es uno de los principales objetivos del aprendizaje de patrones de diseño.

Esto a menudo se denomina reutilización de la experiencia porque obtienes beneficios de las experiencias de otras personas a medida que pasas por sus luchas y ves cómo resolvieron esos problemas y adoptan nuevos comportamientos en sus sistemas. Es posible que un patrón no encaje perfectamente en tu aplicación objetivo, pero si conoce las mejores prácticas de antemano, es más probable que haga una mejor aplicación. Con suerte, puede adivinar que comprender los diferentes patrones de diseño puede no ser muy fácil al principio. Debes conocer ciertos principios o pautas antes de implementar un patrón en tu código.

Estas pautas fundamentales no solo son comunes a todos los patrones, sino que también son útiles para producir software de buena calidad.

Robert Cecil Martin es un nombre famoso en el mundo de la programación. Es un ingeniero de software estadounidense y autor de best-sellers y también es conocido como "Tío Bob". Promovió muchos principios. El siguiente es un subconjunto de ellos:

- Principio de responsabilidad única (PRS)
- Principio abierto/cerrado (OCP)
- Principio de sustitución de Liskov (LSP)
- Principio de segregación de interfaz (ISP)
- Principio de Inversión de Dependencia (DIP)

Tomando la primera letra de cada principio, Michael Feathers introdujo el acrónimo SOLID para recordar fácilmente estos nombres. Los principios de diseño son pautas de alto nivel que puedes utilizar para crear un mejor software. No están vinculados a ningún lenguaje informático en particular. Entonces, si comprendes estos conceptos usando Java, puede usarlos con lenguajes similares como C# o C++.

Principio de responsabilidad única

Una clase actúa como un contenedor que puede contener muchas cosas, como datos, propiedades o métodos. Si ingresas demasiados datos o métodos que no están relacionados entre sí, terminarás con una clase voluminosa que puede crear problemas en el futuro. Consideremos un ejemplo. Supongamos que creas una clase con múltiples métodos que hacen cosas diferentes. En tal caso, incluso si realizas un pequeño cambio en un método, debes volver a probar toda la clase nuevamente para asegurarte de que el flujo de trabajo sea correcto. Por lo tanto, los cambios en un método pueden afectar a los otros métodos relacionados de la clase. Es por eso que el principio de responsabilidad única se opone a esta idea de poner múltiples responsabilidades en una clase. Dice que una clase debe tener una sola razón para cambiar.

Entonces, antes de hacer una clase, identifica la responsabilidad o el propósito de la clase. Si varios miembros te ayudan a lograr un solo propósito, está bien colocar a todos los miembros dentro de la clase.

Punto a recordar

Cuando sigues el SRP, tu código es más pequeño, más limpio y menos frágil. Entonces, ¿cómo se sigue este principio? Una respuesta simple es que puedes dividir un gran problema en partes más pequeñas en función de las diferentes responsabilidades y colocar cada una de estas partes pequeñas en clases separadas. La siguiente pregunta es, ¿qué entendemos por responsabilidad?

En palabras simples, la responsabilidad es una razón para un cambio. En su exitoso libro Clean Architecture (Pearson, 2017), Robert C. Martin nos advierte que no confundamos este principio con el principio que dice que una función debe hacer una y solo una cosa. También creo lo mismo, no solo para este principio sino también para otros principios.

Programa Inicial

La demostración 1 tiene una clase Empleado con tres métodos diferentes. Aquí están los detalles:

- displayEmpDetail() muestra el nombre del empleado y su experiencia laboral en años.
- El método generateEmpId() genera una identificación de empleado mediante la concatenación de cadenas. La lógica es simple: concateno la primera palabra del primer nombre con un número aleatorio para formar una identificación de empleado. Después de la demostración dentro del método main() (el código del cliente) creo dos instancias de Empleado y uso estos métodos para mostrar los detalles relevantes.
- El método checkSeniority() evalúa si un empleado es una persona mayor. Supongamos que si el empleado tiene más de 5 años de experiencia, es un empleado senior; de lo contrario, es un empleado junior.

Demostración 1 - Sin SRP

Empleado.java
Cliente.java

Pregunta 1 Realiza una salida de muestra. Ten en cuenta que la identificación(ID) de un empleado puede variar en tu caso porque genera un número aleatorio para obtener la identificación (ID) del empleado.

Pregunta 2 ¿Cuál es el problema con este diseño?

Mejor programa

En la siguiente demostración, se presentan dos clases más. La clase SeniorityChecker ahora contiene el método checkSeniority() y la clase GeneradorIDEmpleado contiene el método generateEmpId(...) para generar la identificación del empleado. Como resultado, en el futuro, si necesitas cambiar la lógica del programa para determinar el nivel de antigüedad o utilizar un nuevo algoritmo para generar una identificación del empleado, puedes realizar los cambios en las clases respectivas. Otras clases están intactas, por lo que no necesito volver a probar esas clases.

Para mejorar la legibilidad del código y evitar torpezas dentro del método main(), se utiliza el método estático showEmpDetail(...). Este método llama al método displayEmpDetail() de Empleado, al método generateEmpId() de GeneradorIDEmpleado y al método checkSeniority() de SeniorityChecker. Debes entender que este método no era necesario, pero hace que el código del cliente sea simple y fácilmente comprensible.

Demostración 2 Con SRP

Pregunta 3: realiza una demostración completa que sigue a SRP.

Empleado.java
GeneradorIdEmpleado.java
SeniorityChecker.java
Cliente.java

Importante : ten en cuenta que el srp no dice que una clase deba tener como máximo un método. Aquí el énfasis está en la responsabilidad individual. puede haber métodos estrechamente relacionados que pueden ayudarte a implementar una responsabilidad. Por ejemplo, si tienes diferentes métodos para mostrar el nombre, el apellido y el nombre completo, puedes poner estos métodos en la misma clase. Estos métodos están estrechamente relacionados y tiene sentido colocar todos estos métodos de visualización dentro de la misma clase.

Además, no debes concluir que siempre debes separar responsabilidades en cada aplicación que realices. Es necesario analizar la naturaleza del cambio. Esto es porque demasiadas clases pueden hacer que tu aplicación sea compleja y, por lo tanto, difícil de mantener. Pero si conoce este principio y piensas detenidamente antes de implementar un diseño, es probable que evite los errores discutidos anteriormente.

Principio abierto/cerrado

El Principio Abierto-Cerrado (OCP) fue acuñado en 1988 por Bertrand Meyer. Dice: Un artefacto de software debe estar abierto para extensión pero cerrado para modificación. En esta sección, examinaremos el principio OCP en detalle usando clases Java.

Al leer Object-Oriented Software Construction (Second Edition) de Bertrand Meyer, encontramos algunas ideas importantes detrás de este principio. Éstos son algunos de ellos:

- Cualquier técnica de descomposición modular debe satisfacer el Principio Abierto-Cerrado. Los módulos deben ser tanto abiertos como cerrados.
- La contradicción entre ambos términos es sólo aparente en cuanto corresponden a objetivos de diferente naturaleza.
- Se dice que un módulo está abierto si todavía está disponible para extensión. Por ejemplo, debería ser posible expandir su conjunto de operaciones o agregar campos a sus estructuras de datos.
- Se dice que un módulo está cerrado si está disponible para que lo utilicen otros módulos. Esto supone que al módulo se le ha dado una descripción estable y bien definida (su interfaz en el sentido de ocultar información). En el nivel de implementación, el cierre de un módulo también implica que puedes compilarlo, tal vez almacenarlo en una biblioteca y ponerlo a disposición de otros (sus clientes) para que lo usen.
- La necesidad de cerrar los módulos y la necesidad de que permanezcan abiertos surgen por diferentes motivos.
- Explica que la apertura es útil para los desarrolladores de software porque no pueden prever todos los elementos que un módulo puede necesitar en el futuro. Pero los módulos "cerrados" satisfarán a los gerentes de proyecto porque quieren completar el proyecto en lugar de esperar a que todos completen sus partes.

Debes comprender que la idea detrás de esta filosofía de diseño es que en una aplicación estable y en funcionamiento, una vez que crea una clase y otras partes de su aplicación comienzan a usarla, cualquier cambio adicional en la clase puede hacer que la aplicación en funcionamiento se rompa. Si necesitas nuevas características (o funcionalidades), en lugar de cambiar la clase existente, puedes ampliarla para adoptar los nuevos requisitos. ¿Cuál es el beneficio? Dado que no cambia el código anterior, sus funcionalidades existentes continúan funcionando sin ningún problema y puedes evitar probarlas nuevamente. En su lugar, solo prueba la parte "extendida" (o las funcionalidades).

En 1988, Bertrand Meyer sugirió el uso de la herencia en este contexto. Wikipedia (https://en.wikipedia.org/wiki/Open%E2%80%93closed_principle), el menciona su cita de la siguiente manera:

“Una clase está cerrada, ya que puede compilarse, almacenarse en una biblioteca, establecer una línea de base y ser utilizada por clases de clientes. Pero también es abierto, ya que cualquier clase nueva puede usarlo como padre, agregando nuevas funciones. Cuando se define una clase descendiente, no hay necesidad de cambiar el original o molestar a sus clientes.

Pero la herencia promueve un acoplamiento estrecho. En programación, nos gusta eliminar estos acoplamientos estrechos. Robert C. Martin mejoró la definición y la convirtió en OCP polimórfico. La nueva propuesta usa clases base abstractas que usan los protocolos en lugar de una superclase para permitir diferentes implementaciones. Estos protocolos están cerrados a la modificación y proporcionan otro nivel de abstracción que permite un acoplamiento débil. En el curso seguimos la idea de Robert C. Martin que promueve el OCP polimórfico.

Programa Inicial

Supongamos que hay un pequeño grupo de estudiantes que toman un examen semestral. (Para demostrar esto, elijo un pequeño número de participantes para ayudarlo a concentrarse en el principio,

no en detalles innecesarios). Irene, Jessica, Chalo y Claudio John y Kate son los cuatro estudiantes de este ejemplo. Todos ellos pertenecen a la clase Estudiante. Para crear una instancia de clase de Estudiante, debes proporcionar un nombre, un número de registro y las calificaciones obtenidas en el examen. También mencionas si un estudiante pertenece al departamento de Ciencias o al departamento de Artes. Para simplificar, supongamos que una universidad en particular tiene los siguientes cuatro departamentos:

- Ciencias de la Computación
- Física
- Historia
- Inglés

Entonces, verá las siguientes líneas de código en el siguiente ejemplo:

```
Estudiante irene = new Estudiante("Irene", "R1", 81.5, "Ciencia de la Computación.");
Estudiante jessica= new Estudiante("Jessica", "R2", 72, "Fisica");
Estudiante chalo = new Estudiante("Chalo", "R3", 71, "Historia");
Estudiante claudio = new Estudiante("Claudio", "R4", 66.5, "Literatura");
```

Cuando un estudiante opta por informática o física, decimos que opta por la rama de ciencias. De igual forma, cuando un estudiante pertenece al departamento de Historia o Inglés, es un estudiante de artes. Comenzamos con dos métodos de instancia en este ejemplo. displayResult() muestra el resultado con todos los detalles necesarios de un estudiante y el método de evaluateDistinction() evalúa si un estudiante es elegible para un certificado de distinción. Si un estudiante de ciencias obtiene una puntuación superior a 80 en este examen, obtiene el certificado con distinción. Pero el criterio para un estudiante de artes se relaja un poco. Un estudiante de artes obtiene la distinción si su puntaje es superior a 70.

Si entiendes SRP mencionado anteriormente no querrás colocar displayResult() y evaluateDistinction() en la misma clase, como la siguiente:

```
class Student {
        public void displayResult() {
        // código
        }
        public void evaluateDistinction() {
        // código
    }

  }
```

¿Cuáles son los problemas en este segmento de código?

- Primero, viola el SRP porque tanto displayResult() como el métodos evaluationDistinction() están dentro de la clase Student.
- Estos dos métodos no están relacionados. En el futuro, la autoridad examinadora puede cambiar los criterios de distinción. En este caso, tendrás que cambiar el método evaluationDistinction(). ¿Resuelve el problema? En la situación actual, la respuesta es sí. Pero una autoridad universitaria puede cambiar los criterios de distinción de nuevo. ¿Cuántas veces volverás a probar la clase Student debido a la modificación del método evaluateDistinction()?

- Recuerda que cada vez que modificas el método, cambias el que contiene la clase y también necesitas modificar los casos de prueba existentes.

Puede ver que cada vez que cambia el criterio de distinción, necesita modificar el método de evaluateDistinction()() en la clase Estudiante. Entonces, esta clase no sigue el SRP y tampoco está cerrada por modificación. Una vez que comprenda estos problemas, puedes comenzar con un mejor diseño que sigue el SRP.

Estas son las principales características del diseño:

- En el siguiente programa, Student y DistinctionDecider son dos diferentes clases
- La clase DistinctionDecider contiene el método evaluateDistinction() en este ejemplo.
- Para mostrar los detalles de un estudiante, puedes sobrescribir toString(), en lugar de usar el método separado displayResult(). Así que, dentro de la clase Estudiante, verás el método toString() ahora.
- Dentro de main(), verá la siguiente línea:

  List<Student> enrolledStudents = enrollStudents();

- El método enrollStudents() crea una lista de estudiantes. Tu usas esta lista para imprimir los detalles de los estudiantes uno por uno. También usas la misma lista. antes de invocar evaluateDistinction() para identificar a los estudiantes

Demostración 3 sin OCP

Estudiante.java

DistinctionDecider.java

Cliente.java

Pregunta 4. Realiza una salida de muestra.

Ahora estás siguiendo el SRP. Si en el futuro la autoridad examinadora cambia los criterios de distinción, no toca la clase de Estudiante. Por lo tanto, esta parte está cerrada por modificación. Esto resuelve una parte del problema. Ahora piensa en otra posibilidad futura:

- La autoridad del colegio puede introducir una nueva corriente como el comercio y establecer un nuevo criterio de distinción para esta corriente.

Pregunta 5: Modifica el método de evaluateDistinction() y agrega otra instrucción if para considerar a los estudiantes de comercio. ¿Está bien modificar el método evaluateDistinction()de esta manera?

Mejor programa

Para abordar este problema, puedes escribir un mejor programa. El siguiente programa muestra un ejemplo de este tipo. Está escrito siguiendo el principio OCP que sugiere que escribamos segmentos de código (como clases o métodos) que están abiertos para la extensión pero cerrados para la modificación. El OCP se puede lograr de diferentes maneras, pero la abstracción es el corazón de este principio. Si puedes diseñar tu aplicación siguiendo el OCP, tu aplicación es flexible y extensible. No siempre es fácil

implementar completamente este principio, pero el cumplimiento parcial de OCP puede generarle un mayor beneficio. También observa que comenzó la demostración 3 siguiendo el SRP. Si no sigues el OCP, puedes terminar con una clase que realiza varias tareas, lo que significa que el SRP no funciona.

Para la situación actual, puedes dejar la clase Estudiante como está. Pero quieres mejorar el código. Tu comprendes que en el futuro puedes necesitar considerar una corriente diferente, como el comercio. ¿Cómo eliges una corriente? Se basa en el tema elegido por un estudiante, ¿verdad? Entonces, en el siguiente ejemplo, haces que la clase Student sea abstracta.
ArteEstudiante y CienciaEstudiante son las clases concretas que amplían la clase Subject y se utilizan para proporcionar la información del "departamento" (en otras palabras, la materia cursada por un estudiante). El siguiente código muestra una implementación de muestra para tu rápida referencia:

```
abstract class Estudiante {
        String nombre;
        String regNumber;
        double puntuación;
        String departmento;
        public Student(String nombre,
        String regNumber,
        double puntuación) {
        this.name = nombre;
        this.regNumber = regNumber;
        this.score = puntuación;
        }
        public String toString() {
        return ("Nombre: " + número +
        "\nReg Nombre: " + regNumber +
        "\nDept:" + departamento +
        "\nMarks:"         + puntuación +
        "\n*******");
        }
}
public class ArteEstudiante extends Student{
        public ArteEstudiante(String nombre,
                String regNumber,
                double puntuación,
                String dept) {
        super(name, regNumber, puntuación;
        this.department = dept;
        }
}
// The ScienceStudent class no se muestra
```

La construcción anterior lo ayuda a inscribir estudiantes de ciencias y estudiantes de artes por separado dentro del código del cliente de la siguiente manera:

```
private static List<Estudiante> enrollScienceStudents() {
   Estudiante Irene = new CienciaEstudiante("Irene", "R1", 81.5,"Ciencia de la computación.");
        Estudiante jessica = new CienciaEstudiante("Jessica", "R2", 72,"Fisica");
        List<Estudiante> CienciasEstudiantes = new ArrayList<Estudiante>();
```

```
        CienciasEstudiantes.add(Irene);
        CienciasEstudiantes.add(jessica);
        return CienciasEstudiantes;
        }

        private static List<Estudiante> enrollArtsStudents() {
        Estudiante chalo = new ArteEstudiante("Chalo", "R3", 71,"Historia");
        Estudiante claudio = new ArteEstudiante("Claudio", "R4", 66.5,"Literatura");
        List<Estudiante> ArtesEstudiantes = new ArrayList<Estudiante>();
        ArtesEstudiantes.add(chalo);
        ArtesEstudiantes.add(claudio);
        return ArtesEstudiantes;
        }
```

Ahora centrémonos en los cambios más importantes. Debes abordar el método de evaluación para la distinción de una mejor manera. Por lo tanto, crea la interfaz DistinctionDecider que contiene un método llamado EvaluationDistinction. Aquí está la interfaz:

```
interface DistinctionDecider {
        void evaluateDistinction(Estudiante estudiante);
}
```

ArtsDistinctionDecider y ScienceDistinctionDecider implementan esta interfaz y sobreescriben el método de evaluateDistinction(...) para especificar los criterios de evaluación según sus necesidades. De esta forma, los criterios de distinción específicos de flujo se envuelven en una unidad independiente. Aquí está el segmento de código para ti. Los diferentes criterios para cada clase se muestran en negrita.

```
// ScienceDistinctionDecider.java
public class ScienceDistinctionDecider implements DistinctionDecider {
 @Override
   public void evaluateDistinction(Estudiante estudiante) {
            if (estudiante.score > 80) {
          System.out.println(estudiante.regNumber+" ha recibido una distinción en ciencias.");
            }
    }
}

// ArtsDistinctionDecider.java
public class ArtsDistinctionDecider implements DistinctionDecider{
   @Override
  public void evaluateDistinction(Estudiante estudiante) {
    if (estudiante.score > 70) {
      System.out.println(estudiante.regNumber+" ha recibido una distinción en Artes.");
     }
   }
}
```

Nota El método de evaluateDistinction(...) acepta un parámetro Estudiante. Significa que ahora también puede pasar un objeto ArtsStudent o un objeto ScienceStudent a este método.

El código restante es fácil y no debería tener ningún problema para comprender la siguiente demostración ahora.

Demostración 4

Estudiante.java

ArteEstudiante.java

CienciaEstudiante.java

DistinctionDecider.java

ScienceDistinctionDecider.java

ArtsDistinctionDecider.java

Cliente.java

Pregunta 6. Realiza una salida de muestra.

Pregunta 7. ¿Cuáles son las principales ventajas ahora?

Principio de sustitución de Liskov

El principio de sustitución de Liskov se introdujo inicialmente a partir del trabajo de Barbara Liskov en 1988. El LSP dice que deberías poder sustituir un tipo padre (o base) con un subtipo. Significa que en un segmento de programa, puedes usar una clase derivada en lugar de su clase base sin alterar la corrección del programa.

¿Cómo se usa la herencia? Hay una clase base y crea una (o más) clases derivadas de ella. Luego puedes agregar nuevos métodos en las clases derivadas. Siempre que uses directamente el método de clase derivada con un objeto de clase derivada, todo está bien. Puede ocurrir un problema si intentas obtener el comportamiento polimórfico sin seguir el LSP. ¿Cómo?

Déjame darte una breve idea. Supongamos que hay dos clases en las que B es la clase base y D es la subclase (de B). Además, suponga que existe un método que acepta una referencia de B como argumento, algo como lo siguiente:

```
public void someMethod(B b){
        // Código
        }
```

Este método funciona bien hasta que le pasas una instancia B. Pero, ¿qué sucede si pasas una instancia D en lugar de una instancia B? Idealmente, el programa no debería fallar. Es porque usas el concepto de polimorfismo y dices que D es básicamente un tipo B ya que la clase D hereda de la clase B. Puedes relacionar este escenario con un ejemplo común cuando decimos que un jugador de fútbol también es un jugador, donde consideramos la clase "jugador" es un supertipo de "jugador de fútbol".

¿Ves ahora lo que nos sugiere el LSP? Dice que someMethod() no debería comportarse mal/fallar si se le pasa una instancia D en lugar de una instancia B. Pero puede suceder si no escribes su código siguiendo el LSP. El concepto le resultará más claro cuando analice el siguiente ejemplo.

En los patrones de diseño, a menudo se ve código polimórfico. Aquí hay un ejemplo común. Suponga que tiene el siguiente segmento de código:

class B{}
class D extends B{}

Ahora puedes escribir B obB=new B(); con seguridad. Pero ten en cuenta que en este caso, también puede escribir B obB=new D(); // También está bien

Del mismo modo, puedes utilizar las interfaces como supertipo. Por ejemplo, si tienes

interfaz B{}
class D implements B{}

puedes escribir

B obB=nuevo D(); // También está bien

El código polimórfico muestra su experiencia, pero recuerda que es tu responsabilidad implementar el comportamiento polimórfico correctamente y evitar resultados no deseados.

Programa Inicial

Utilizo un portal de pago en línea para pagar una factura. Como soy un usuario registrado, cuando realizo una solicitud de pago en este portal, también muestra mis pagos anteriores. Consideremos un ejemplo simplificado basado en este escenario de la vida real.

Supongamos que también tienes un portal de pago donde un usuario registrado puede realizar una solicitud de pago. Utiliza el método newPayment() para esto. En este portal, también puede mostrar los detalles del último pago del usuario utilizando un método llamado previousPaymentInfo(). Aquí hay un segmento de código de muestra para esto:

```
interface Payment {
        void previousPaymentInfo();
        void newPayment();
}

public class RegisteredUserPayment implements Payment{
  String name;
  public RegisteredUserPayment(String userName) {
   this.name = userName;
    }
  @Override
    public void previousPaymentInfo(){
        System.out.println("Recuperando de "+ name+ ", últimos detalles de pagos.");
  }
```

```
   @Override
    public void newPayment(){
            System.out.println("Procesando de "+name+", la actual solicitud de pagos .");

    }
}
```

Además, crea la clase de ayuda PaymentHelper para mostrar todos los pagos anteriores y las nuevas solicitudes de pago de estos usuarios. Utiliza showPreviousPayments() y processNewPayments() para estas actividades.

Estos métodos llaman a previousPaymentInfo() y newPayment() en las respectivas instancias de pago. Utilizas una instrucción **for** mejorada (a menudo se denomina bucle for mejorado) para cumplir estos propósitos. Aquí está la clase PaymentHelper para su referencia instantánea:

```
import java.util.ArrayList;
import java.util.List;
public class PaymentHelper {
  List<Payment> payments = new ArrayList<Payment>();
            public void addUser(Payment user){
               payments.add(user);
            }
    public void showPreviousPayments() {
       for (Payment payment: payments) {
                payment.previousPaymentInfo();
                   System.out.println("------");
            }
    }
  public void processNewPayments()  {
      for (Payment payment: payments) {
            payment.newPayment();
            System.out.println("------");
      }
  }
}
```

Dentro del código del cliente, crea dos usuarios y muestra sus solicitudes de pago actuales junto con los pagos anteriores. Todo está bien hasta ahora.

Demostración 5

Payment.java
RegisteredUserPayment.java
PaymentHelper.java
Cliente.java

Pregunta 8. Realiza una salida de muestra.

Este programa parece estar bien. Ahora suponga que tiene un nuevo requisito que dice que necesita admitir usuarios invitados en el futuro. Puede procesar la solicitud de pago de un usuario invitado, pero no muestra su último detalle de pago. Entonces, crea la siguiente clase que implementa la interfaz de pago de la siguiente manera:

```
class GuestUserPayment implements Payment {
        String name;
        public GuestUserPayment() {
          this.name = "guest";
        }
        @Override
        public void previousPaymentInfo(){
          throw new UnsupportedOperationException();
}
 @Override
        public void newPayment(){
        System.out.println("Procesando de  "+name+ "pago actual
                request.");

        }
}
```

Dentro del método main(), ahora crea una instancia de usuario invitado e intenta usar su clase auxiliar de la misma manera. Aquí está el nuevo código de cliente (observa los cambios en negrita). Para su fácil comprensión, se agregan algunos comentarios para llamar su atención sobre el código que causa el problema ahora.

```
public class Cliente {
  public static void main(String[] args) {
    System.out.println("Demostracion sin LSP\n");
    PaymentHelper helper = new PaymentHelper();

    // Instanciando dos usuarios registrados
    RegisteredUserPayment pagoAbejita = new RegisteredUserPayment("Abejita");
    RegisteredUserPayment pagoChalito = new RegisteredUserPayment("Chalito");

    // Agregando los usuarios a los helper
    helper.addUser(pagoAbejita);
    helper.addUser(pagoChalito);

     GuestUserPayment guestUser = new GuestUserPayment();
        helper.addUser(guestUser);

    // Procesando el pago usando la clase helper
    // Encuentras algún problema?
    helper.showPreviousPayments();
   helper.processNewPayments();

  }
}
```

Pregunta 9. Realiza una salida de muestra y describe la excepción resultante. ¿Cuál es el problema?.

¿Cuál es la solución?

La primera solución obvia que se le puede ocurrir es introducir una cadena if-else para verificar si la instancia de pago es un pago de usuario invitado (GuestUserPayment) o un pago de usuario registrado (RegisteredUserPayment). Sin embargo, esta no es la mejor solución posible en el sentido de que si tiene otro tipo especial de usuario, vuelves a verificarlo dentro de esta cadena if-else. Lo más importante es que viola el OCP cada vez que modifica una clase existente que usa esta cadena if-else. Entonces, busquemos una mejor solución.

En el próximo programa, eliminaremos el método newPayment() de la interfaz de payment. Coloca este método en otra interfaz llamada NewPayment. Como resultado, ahora tiene dos interfaces con las operaciones específicas. Dado que todos los tipos de usuarios pueden generar una nueva solicitud de pago, las clases concretas de RegisteredUserPayment y GuestUserPayment implementan la interfaz NewPayment. Pero muestra el último detalle de pago solo para los usuarios registrados. Entonces, la clase RegisteredUser implementa la interfaz payment. Dado que Payment contiene el método previousPaymentInfo(), tiene sentido elegir un nombre mejor, como PreviousPayment en lugar de Payment. Entonces, ahora verá las siguientes interfaces:

```
interface PreviousPayment {
        void previousPaymentInfo();
}
interface NewPayment {
        void newPayment();
}
```

Ajuste estos nuevos nombres en la clase auxiliar también. Veamos el programa actualizado que se muestra en la siguiente sección.

Demostración 6

PreviousPayment.java

NewPayment.java

RegisteredUserPayment.java

GuestUserPayment.java

PaymentHelper.java

Cliente.java

Problema 10: ¿cuáles son los cambios clave?

Principio de segregación de interfaz

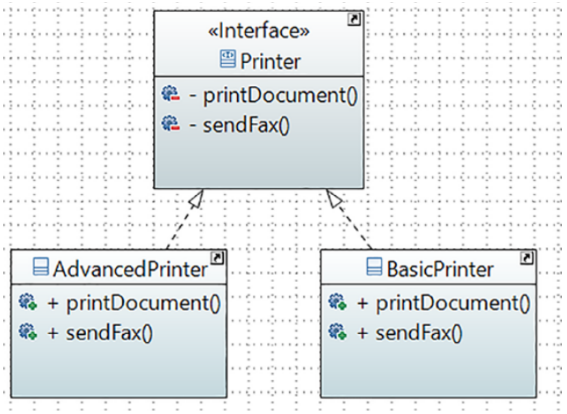
A menudo ves una interfaz voluminosa que contiene muchos métodos. Es posible que una clase que implemente la interfaz no necesite todos estos métodos. Entonces, ¿por qué la interfaz contiene todos estos métodos? Una posible respuesta es admitir algunas de las clases de implementación de esta interfaz. Esta es el área en la que se centra el Principio de Segregación de la Interfaz. Sugiere que no contamines una interfaz con estos métodos innecesarios solo para admitir una (o algunas) de las clases de implementación de esta interfaz. La idea es que un cliente no debe depender de un método que no utiliza.

Ten en cuenta los siguientes puntos antes de continuar:

- Un cliente significa cualquier clase que usa otra clase (o interfaz).
- La palabra “Interfaz” del Principio de Segregación de Interfaz no se limita a una interfaz Java. El mismo concepto se aplica a cualquier supertipo, como una clase abstracta o una clase principal simple.
- Muchos ejemplos a través de diferentes fuentes explican la violación del ISP con énfasis en lanzar una excepción como UnsupportedOperationException() en Java.
- El ISP sugiere que tu clase no debe depender de métodos de interfaz que no utiliza. Esta afirmación tendrá sentido para ti cuando analices el siguiente ejemplo y recuerda los puntos anteriores.

Programa Inicial

Supongamos que tiene la interfaz Impresora con dos métodos, printDocument() y sendFax(). Hay varios usuarios de esta clase. Para simplificar, consideremos solo dos de ellos: BasicPrinter y AdvancedPrinter. La figura muestra un diagrama de clases simple para esto.



Una impresora básica puede imprimir documentos. No es compatible con ninguna otra funcionalidad. Por lo tanto, BasicPrinter solo necesita el método printDocument(). Una impresora avanzada puede imprimir documentos y enviar faxes. Entonces, AdvancedPrinter necesita ambos métodos.

En este caso, un cambio en el método sendFax() en AdvancedPrinter puede obligar a la interfaz Printer a cambiar, lo que a su vez obliga al código de BasicPrinter a recompilarse. Esta situación no es deseada y puede causarle problemas potenciales en el futuro.

En este caso ISP sugiere que diseñes tu interfaz con los métodos adecuados que un cliente en particular pueda necesitar. ¿Por qué un usuario necesita cambiar una clase base (o una interfaz)?

Para responder a esto, supongamos que deseas mostrar qué el tipo de fax está utilizando en una fase de desarrollo posterior. Tu sabes que existen diferentes variaciones de métodos de fax, como LanFax, InternetFax (o EFax) y AnalogFax. Entonces, antes, el método SendFax() no usaba ningún parámetro, pero ahora necesita aceptar un parámetro para mostrar el tipo de fax que usa. Para demostrar esto aún más, suponga que tiene una jerarquía de fax que puede parecerse a la siguiente:

```
interface Fax {
	void faxType();
}
class LanFax implements Fax {
	@Override
	public void faxType() {
	System.out.println("Usando lan fax para enviar el fax.");
	}
}
class EFax implements Fax {
	@Override
	public void faxType() {
	System.out.println("Usando internet fax(efax) para el fax.");
	}
}
```

Para usar esta jerarquía de herencia, una vez que modificas el método sendFax() a sendFax(Fax faxType) en la clase AdvancedPrinter, exige que cambies la interfaz de Printer (sí, aquí también rompes el OCP). Cuando actualices printer, también debes actualizar la clase BasicPrinter para adaptarse a este cambio. ¡Ahora ves el problema!

Viste que un cambio en AdvancedPrinter provoca cambios en la interfaz Printer, lo que a su vez hace que BasicPrinter actualice su método de fax. Entonces, puedes ver que aunque BasicPrinter no necesita este método de fax en absoluto, un cambio en AdvancedPrinter hace que cambie y se vuelva a compilar. El ISP sugiere que se ocupe de este tipo de escenario.

Es por eso que cuando veas una interfaz voluminosa, pregúntate si estos métodos son necesarios para un cliente. Si no, divídelos en interfaces más pequeñas que sean relevantes para los clientes.

Si comprendes la discusión anterior, es posible que no comiences con el siguiente código en el que asume que es posible que necesite admitir diferentes dispositivos/impresoras en el futuro:

```
interface Impresora {
	void printDocument();
	void sendFax();
}
```

Si comienzas tu codificación considerando las impresoras avanzadas que pueden imprimir y enviar un fax, está bien. Pero en una etapa posterior, si tu programa también necesita admitir impresoras básicas, puede escribir algo como:

```
class ImpresoraBasica implements Impresora {
 @Override
   public void printDocument() {
    System.out.println("La impresora básica imprime un documento.");
   }

  @Override
   public void sendFax() {
     throw new UnsupportedOperationException();
    }
}
```

Pregunta 11: ¿Cuál es el problema con este código?

Pregunta 12: ¿Puedes escribir código polimórfico como el siguiente ?. Explica tu respuesta.

```
Impresora  impresora = new ImpresoraAvanzada();
impresora.printDocument();
impresora.sendFax();
impresora = new ImpresoraBasica();
impresora.printDocument();
// impresora.sendFax();
```

Pregunta 13: ¿Qué sucede si escribimos algo así en el código dado?

```
List<Impresora> impresoras = new ArrayList<Impresoras>();
impresoras.add(new ImpresoraAvanzada());
impresoras.add(new ImpresoraBasica());
for (Impresora device : impresoras) {
 device.printDocument();
 // device.sendFax();
}
```

Demostración 7

Impresora.java
ImpresoraBasica.java
ImpresoraAvanzada.java
Cliente.java

Pregunta 14: Realiza una salida de muestra.

Para evitar las excepciones de tiempo de ejecución, necesitaba comentar una línea de código. Guardé este código muerto para esta discusión. Sabes que debes evitar este tipo de código comentado porque puede causar problemas potenciales a largo plazo. Dado que nadie toca el código comentado, existe la posibilidad de que en el futuro ocurran muchos cambios en el código base y luego este código se vuelva irrelevante. Entonces, cuando un nuevo desarrollador lea el código, no tendrá ni idea al respecto.

Lo más importante, como se dijo antes, en este diseño es que si cambias la firma del método sendFax() en ImpresoraAvanzada, debes ajustar el cambio en Impresora, lo que hace que ImpresoraBasica cambie y vuelva a compilar.

Piensa en el problema desde otro ángulo. Supongamos que necesitas admitir otra impresora que pueda imprimir, enviar faxes y fotocopiar. En este caso, si agregas un método de fotocopiado en la interfaz Impresora, los dos clientes existentes, ImpresoraBasica y Impresora Avanzada, deben adaptarse al cambio.

Mejor programa

Busquemos una mejor solución. Entiendes que hay dos actividades diferentes: una es imprimir unos documentos y la otra es enviar un fax. Entonces, en el siguiente ejemplo, crea dos interfaces llamada Impresora y DispositivoFax. Impresora contiene el método printDocument() y FaxDevice contiene el método SendFax(). La idea es sencilla:

- La clase que desea la función de impresión implementa la interfaz Impresora y la clase que desea la función de fax implementa la interfaz DispositivoFax.
- Si una clase quiere ambas funcionalidades, implementa ambas interfaces.

No debes asumir que el ISP dice que una interfaz debe tener solo un método. En este ejemplo, hay dos métodos en la interfaz de Impresora y la clase ImpresoraBasica necesita solo uno de ellos. Es por eso que ves las interfaces segregadas con un solo método.

Demostración 8

Impresora.java
DispositivoFax.java

ImpresoraBasica.java
ImpresoraAvanzada.java

Cliente.java

Pregunta 15: Realiza una salida de muestra.

Pregunta 16: ¿Qué sucede si usa un método predeterminado dentro de la interfaz? Por ejemplo, si proporcionas un método de fax predeterminado en una interfaz (o una clase abstracta), ImpresoraBasica debe sobreescribirlo y decir algo similar a lo siguiente:

```
@Override
public void sendFax() {
  throw new UnsupportedOperationException();
}
```

¿viste el problema potencial con esto! . Pero, ¿qué sucede si usa un método vacío, en lugar de lanzar la excepción?

Principio de inversión de dependencia

El DIP cubre dos cosas importantes:

- Una clase concreta de alto nivel no debe depender de una clase concreta de bajo nivel. En cambio, ambos deberían depender de abstracciones.
- Las abstracciones no deben depender de los detalles. En cambio, los detalles deberían depender de abstracciones.

Examinaremos ambos puntos. La razón del primer punto es simple. Si la clase de bajo nivel cambia, la clase de alto nivel necesita adaptarse al cambio; de lo contrario, la aplicación se rompe. ¿Qué significa esto? Significa que debes evitar crear una clase concreta de bajo nivel dentro de una clase de alto nivel. En su lugar, debes utilizar clases o interfaces abstractas. Como resultado, elimina el acoplamiento estrecho entre las clases.

El segundo punto también es fácil de entender cuando analizas el caso de estudio discutido en la sección ISP. Viste que si una interfaz necesita cambiar para admitir uno de sus clientes, otros clientes pueden verse afectados por el cambio. A ningún cliente le gusta ver una aplicación así. Entonces, en tu aplicación, si tus módulos de alto nivel son independientes de los módulos de bajo nivel, puedes reutilizarlos fácilmente. Esta idea también te ayuda a diseñar marcos agradables.

Robert C. Martin explica que un modelo de desarrollo de software tradicional en esos días (como el análisis y el diseño estructurados) tiende a crear software en el que los módulos de alto nivel solían depender de módulos de bajo nivel. Pero en OOP, un programa bien diseñado se opone a esta idea. Aquí se invierte la estructura de dependencia que a menudo resulta de un método de procedimiento tradicional. Esta es la razón por la que usó la palabra "inversión" en este principio.

Programa Inicial

Supongamos que tienes una aplicación de dos capas. Usando esta aplicación, un usuario puede guardar una identificación de empleado en una base de datos. Para demostrar esto, usamos una aplicación de consola en lugar de una aplicación GUI. Tiene dos clases, InterfazUsuario y OracleDatabase. Según su nombre, InterfazUsuario representa una interfaz de usuario como un formulario donde un usuario puede escribir una identificación de empleado y hacer clic en el botón Guardar para guardar la identificación en una base de datos. OracleDatabase se utiliza para imitar una base de datos Oracle.

Nuevamente, para simplificar, no hay una base de datos real en esta aplicación y no hay un código para validar una identificación de empleado. Aquí su atención se centra únicamente en el DIP, por lo que esas discusiones no son importantes.

Al usar el método saveEmployeeId() de UserInterface, puedes guardar una identificación de empleado en una base de datos. Observa el código para la clase InterfazUsuario:

```
public class InterfazUsuario {
 private OracleDatabase oracleDatabase;

  public InterfazUsuario() {
    this.oracleDatabase = new OracleDatabase();
   }

  public void saveEmployeeId(String empId) {
```

```
    oracleDatabase.saveEmpIdInDatabase(empId);
        }
}
```

Esto crea una instancia de un objeto OracleDatabase dentro del constructor InterfazUsuario. Más tarde, utiliza este objeto para invocar el método saveEmpIdInDatabase(), que realiza el guardado real dentro de la base de datos de Oracle. Este estilo de codificación es muy común. Pero hay algunos problemas. Por ahora, vea el programa completo, que no sigue el DIP.

Demostración 9

InterfaceUsuario.java
OracleDataBase.java
Cliente.java

Pregunta 17: Realiza una salida de muestra. ¿Cuáles son los problemas que adolece el código?

Mejor programa

En este programa, verás la siguiente jerarquía:

BaseDatos.java
OracleDatabase.java

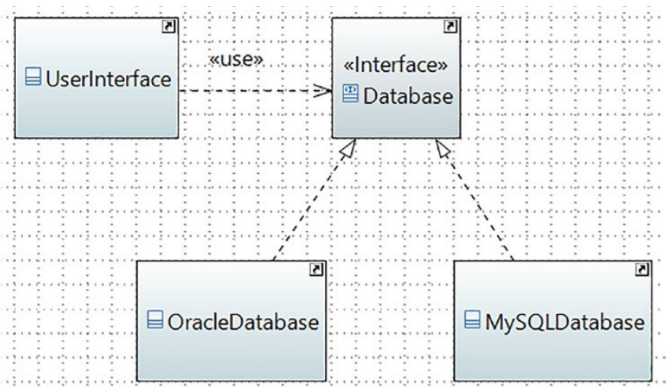
La primera parte del DIP sugiere que nos centramos en la abstracción. Esto hace que el programa sea eficiente. Entonces, esta vez la clase InterfazUsuario tiene como objetivo la base de datos de abstracción, en lugar de una implementación concreta como OracleDatabase. Aquí está la nueva construcción de InterfazUsuario:

```
class InterfazUsuario {
   BaseDatos basedatos;

    public InterfazUsuario(BaseDatos basedatos) {
      this.basedatos = basedatos;
        }


    public void saveEmployeeId(String empId) {
        basedatos.saveEmpIdInDatabase(empId);
        }
}
```

Esto brinda la flexibilidad de considerar una nueva base de datos, como MYSQLDatabase también. La figura describe el escenario.



La interfaz de usuario de clase de alto nivel depende de la base de datos de abstracción. Las clases concretas OracleDatabase y MySQLDatabase también dependen de Database.

La segunda parte del DIP sugiere hacer la interfaz de la base de datos considerando la necesidad de la clase UserInterface. Es importante porque si una interfaz necesita cambiar para admitir a uno de sus clientes, otros clientes pueden verse afectados por el cambio.

Demostración 10

InterfazUsuario.java
BaseDatos.java
OracleDataBase.java
MySQLDatabase.java
Cliente.java

Pregunta 18: Realiza una salida de muestra. ¿Esto resuelve todos los problemas que adolece el código?

En resumen, en OOP, sugiero seguir la cita de Robert C. Martin:

Los módulos de alto nivel simplemente no deberían depender de los módulos de bajo nivel de ninguna manera.

Entonces, cuando tienes una clase base y una clase derivada, tu clase base no debe conocer ninguna de sus clases derivadas. Pero hay pocas excepciones a esta sugerencia. Por ejemplo, considera el caso en el que tu clase base necesita restringir el recuento de las instancias de la clase derivada en un punto determinado.

Un último punto.

Puede ver que en la demostración, el constructor de la clase InterfazUsuario acepta un parámetro de base de datos.
Puedes proporcionar una instalación adicional a un usuario cuando utiliza tanto el constructor como el método setter (setDatabase) dentro de esta clase. Aquí hay un código de muestra. Observa el código adicional en negrita.

```
class InterfazUsuario {
    BaseDatos basedatos;
```

```
    public InterfazUsuario(BaseDatos basedatos) {
      this.basedatos = basedatos;
            }

     public void setDatabase(BaseDatos basedatos) {
            this.basedatos = basedatos;
            }

    public void saveEmployeeId(String empId) {
            basedatos.saveEmpIdInDatabase(empId);
            }
}
```

¿Cuál es el beneficio? Ahora puedes crear una instancia de una base de datos mientras creas una instancia de la clase InterfazUsuario y cambiar la base de datos objetivo más adelante utilizando el método setter. Aquí hay un código de muestra.

Puedes agregar la última parte de este segmento al final de main() para ver el nuevo resultado.

```
// Usando Mysql
   basedatos = new MySQLDatabase();
   usuario = new InterfazUsuario(basedatos);
   usuario.saveEmployeeId("E002");

  // Cambiando la base de datos objetivo
   //usuario = new InterfazUsuario(new OracleDatabase());
   usuario.setDatabase(new OracleDatabase());
   usuario.saveEmployeeId("E002");
```

Pregunta 19: Verifica los resultados.