

PRACTICA CALIFICADA 3 - DESARROLLO DE SOFTWARE

Integrantes:

Lluen Gallardo, Armando Alberto.

Meza Rodriguez, Fiorella Ivonne.

Paredes López, Maxwell Joel

Link del repositorio: <https://github.com/Well192/Practica-3>

```
Demostracion sin SRP
Nombre del empleado: Abejita,Jessica
Este empleado tiene 7.5 años de experiencia.
El ID del empleado es: J927
Este empleado es un empleado senior

----

Nombre del empleado: Smart,Chalito
Este empleado tiene 3.2 años de experiencia.
El ID del empleado es: C20
Este empleado es un empleado junior
```

1. Como podemos ver, el método showEmpDetail recibe un objeto empleado y lo utiliza para mostrar todos los detalles del empleado, para ello usa los métodos del objeto empleado como el método displayEmpDetail el cual imprime el nombre del empleado, haciendo uso de los atributos lastName y firstName e imprime los años de experiencia, haciendo uso del atributo experienceInYears. Luego imprime el identificador del empleado con el método generateEmpId, el cual genera un número aleatorio de 0 a 999 y este valor es concatenado con la primera inicial del apellido del Empleado y así se obtiene el IdEmp, por último este valor es retornado. Luego muestra si es un empleado junior o senior con el método checkSeniority el cual verifica si el empleado es senior o junior, si la variable experienceInYears es mayor que 5 se considera senior, si no se cumple se considera junior y éste valor es retornado.
2. El problema con este diseño es que no está cumpliendo con el principio de Responsabilidad única por varias razones:
 - a. La primera es que la clase empleado se está usando tanto como capa de presentación (pues se está imprimiendo los datos del empleado), también es capa lógica (pues es la que genera el identificador del empleado y se detecta si es un senior o un junior) y también como capa de persistencia (pues allí tambien se guardan los datos del empleado).
 - b. Otra razón es que el método generateEmpId cada vez que sea llamado puede generar un ID diferente a un mismo empleado, lo cual generaría errores con la identificación de un mismo empleado.
 - c. Otro posible fallo es la falta de encapsulación de los atributos.

3. Se subió el código al repositorio y se explica a continuación:

- a. Primero completamos la clase empleado, encapsulando los atributos y retirando los métodos generateEmpId y checkSeniority para poder cumplir con el principio de responsabilidad única.

```
public class Empleado{
    3 usages
    private final String firstName, lastName;
    2 usages
    private String empId;
    3 usages
    private final double experienceInYears;
    2 usages  ▶ ArmandoLlue
    public Empleado(String firstName, String lastName, double experience) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.experienceInYears = experience;
    }

    1 usage  ▶ ArmandoLlue
    public void displayEmpDetail(){
        System.out.println("Nombre del empleado: "+lastName+", "+firstName);
        System.out.println("Este empleado tiene "+ experienceInYears+" años de experiencia.");
    }

    1 usage  ▶ ArmandoLlue
    public String getFirstName() {
        return firstName;
    }

    1 usage  ▶ ArmandoLlue
    public String getEmpId() {
        return empId;
    }
}
```

- b. En la clase GeneradorIDEmpleado se implementó el método estático generateEmpId para poder obtener un ID aleatorio, este método recibe un dato string que es el nombre del empleado y devuelve el string ID.

```
package Solid.SRP;
import java.util.Random;

1 usage  ▶ ArmandoLlue
public class GeneradorIDEmpleado {
    1 usage  ▶ ArmandoLlue
    public static String generateEmpId(String empFirstName) {
        int random = new Random().nextInt( bound: 1000);
        return empFirstName.substring(0,1)+random;
    }
}
```

- c. Se implementó la clase SeniorityChecker con el método estático checkSeniority para poder verificar si un empleado es senior o junior, este método recibe un dato doble, que representan los años de experiencia y retorna "junior" si es menor a 5 y "senior" si es mayor a 5.

```
package Solid.SRP;

import ArmandoLlueen;

public class SeniorityChecker{
    1 usage ArmandoLlueen
    public static String checkSeniority(double experienceInYears){
        return experienceInYears > 5 ? "senior": "junior";
    }
}
```

- d. Finalmente se implementó la clase Cliente para que presente los datos de los empleados teniendo en cuenta los cambios hechos en la clase empleado.

```
public class Cliente {
    ArmandoLlueen
    public static void main(String[] args) {
        System.out.println("Demostracion de SRP");

        Empleado jessica = new Empleado( firstName: "Jessica", lastName: "Abejita", experience: 7.5);
        showEmpDetail(jessica);

        System.out.println("\n*****\n");

        Empleado chalo = new Empleado ( firstName: "Chalito", lastName: "Smart", experience: 3.2);
        showEmpDetail(chalo);
    }

    2 usages ArmandoLlueen
    private static void showEmpDetail(Empleado emp) {

        // Muestra detalles del empleado
        emp.displayEmpDetail();

        //Genera el ID
        emp.setEmpId(GeneradorIDEmpleado.generateEmpId(emp.getFirstName()));
        System.out.println("El ID del empleado es: " + emp.getEmpId());

        // Verifica el nivel laboral
        System.out.println("Este empleado es un " + " empleado " + SeniorityChecker.checkSeniority(emp.getExperienceInYears()));
    }
}
```

4. Los resultados son los siguientes:

```
Demostracion de SRP
Nombre del empleado: Abejita,Jessica
Este empleado tiene 7.5 años de experiencia.
El ID del empleado es: J754
Este empleado es un empleado senior

*****

Nombre del empleado: Smart,Chalito
Este empleado tiene 3.2 años de experiencia.
El ID del empleado es: C226
Este empleado es un empleado junior
```

Como podemos observar los resultados son muy parecidos que los de al comienzo, la diferencia es que esta vez, la clase empleado no es la que genera el id ni verifica si es un empleado junior, en este caso, primero se usa el método generateEmpId de la clase GeneradorIDEmpleado para setearlo en el empleado. Luego se utiliza el método checkSeniority para verificar si el empleado es junior o senior y se imprimen los resultados.

5. No sería correcto colocar displayResult() y evaluateDistinction() en la misma clase ya que si implementamos distintos departamentos, se tendría que modificar el método evaluateDistinction() dependiendo del puntaje obtenido y del departamento al que pertenezca el estudiante, lo que aumentaría la complejidad y disminuiría la reutilización, otra razón es que está violando el principio de Responsabilidad única pues en una clase se está utilizando como presentación y también como capa de lógica.

```
C:\Users\Usuario\.jdk\corretto-17.0.3\bin\java.exe "-javaagent:C:\Progr
Demostracion sin OCP
Resultados:
Nombre: Irene
Numero Regex: R1
Dept:Ciencia de la Computacion.
Marks:81.5
*****
Nombre: Jessica
Numero Regex: R2
Dept:Fisica
Marks:72.0
*****
Nombre: Chalo
Numero Regex: R3
Dept:Historia
Marks:71.0
*****
Nombre: Claudio
Numero Regex: R4
Dept:Literatura
Marks:66.5
*****
Distinciones:
R1 ha recibido una distincion en ciencias.
R3 ha recibido una distincion en artes.

Process finished with exit code 0
```

6.

En el método enrollStudents() se instancian 4 objetos de tipo Estudiante, irene, jessica, chalo, claudio, éstos se añadieron a una lista llamada estudiantes y ésta es retornada.

```
private static List<Estudiante> enrollStudents() {
    Estudiante irene = new Estudiante("Irene", "R1", 81.5, "Ciencia de la
Computacion.");
    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");

    List<Estudiante> estudiantes = new ArrayList<Estudiante>();
    estudiantes.add(irene);
    estudiantes.add(jessica);
    estudiantes.add(chalo);
    estudiantes.add(claudio);
    return estudiantes;
}
```

El método `evaluateDistinction(Estudiante estudiante)` verifica si el estudiante es elegible para un certificado, dependiendo del departamento al que pertenece ya que a cada departamento le corresponde un score mínimo, y si el estudiante es elegible, imprime el número Regex.

```
public void evaluateDistinction(Estudiante estudiante) {  
  
    if (science.contains(estudiante.department)) {  
        if (estudiante.score > 80) {  
            System.out.println(estudiante.regNumber+" ha recibido una distincion en ciencias.");  
        }  
    }  
  
    if (arts.contains(estudiante.department)) {  
        if (estudiante.score > 70) {  
            System.out.println(estudiante.regNumber+" ha recibido una distincion en artes.");  
        }  
    }  
}
```

El método `toString()`:

Este método es utilizado para retornar un string con los nombres, número regex, departamento y el puntaje de un estudiante.

```
public String toString() {  
    return ("Nombre: " + name + "\nNumero Regex: " + regNumber + "\nDept:" + department + "\nMarks:" + score + "\n*****");  
}
```

7. El principal problema de este diseño es debido al bajo índice de usabilidad que posee, lo cual termina violando el principio de abierto/ cerrado, poniendo el supuesto en el que se aumente un departamento en la universidad, se tendrá que modificar el código original y volver a comprobar que todos los componentes modificados funcionen correctamente, lo cual a un programa más grande conlleva a un mayor esfuerzo. La razón de este problema es que al haber varios tipos de estudiantes y que según su departamento hay que tener tratos especiales va a generar diferentes casos especiales por departamento que a la larga generará que el código de la clase estudiante aumente y se vea saturado de casos especiales. Esto se puede solucionar si se hace uso del polimorfismo.

8. Implementado en el repositorio
9. Implementado en el repositorio
10. En la clase Estudiante se definió el constructor con los atributos, name, regNumber, double score y también se implementó el método toString() para que muestre la información del estudiante, haciendo uso de los atributos definidos en el constructor.

```
package Solid.OCP;

import ArmandoLLuen;

abstract class Estudiante {
    2 usages
    String name;
    4 usages
    String regNumber;
    4 usages
    double score;
    5 usages
    String department;

    2 usages ArmandoLLuen
    public Estudiante(String name, String regNumber, double score) {
        this.name = name;
        this.regNumber = regNumber;
        this.score = score;
    }

    2 overrides ArmandoLLuen
    public String toString() {
        return ("Nombre: " + name + "\nNumero Reg: " + regNumber + "\nDept:" + department + "\nMarks:"
            + score + "\n");
    }
}
```

En la clase ArteEstudiante que hereda de Estudiante, se implementó el constructor que hace referencia al constructor de la superclase y se agregó el atributo department. Además se sobrescribe el método toString() haciendo referencia a la superclase.

```
import fiorellamr +1 *

public class ArteEstudiante extends Estudiante{
    2 usages fiorellamr
    public ArteEstudiante(String name, String regNumber, double score, String department) {
        super(name, regNumber, score);
        this.department = department;
    }

    fiorellamr +1
    @Override
    public String toString() { return super.toString(); }
}
```

En la clase CienciaEstudiante, vemos que en el constructor se hizo referencia al constructor de la superclase y se añadió el atributo department. Así como, se sobrescribe el método toString() haciendo referencia al método de la superclase.

```

fiorellamr +1
public class CienciaEstudiante extends Estudiante{
    2 usages  fiorellamr +1
    public CienciaEstudiante(String name, String regNumber, double score,String departament) {
        super(name, regNumber, score);
        this.department = departament;
    }

    fiorellamr +1
    @Override
    public String toString() { return super.toString(); }
}

```

En la interface DistinctionDecider se definió el método evaluateDistinction() para que sea implementado según el departamento, ya que cada uno de éstos considera un puntaje distinto para definir si un estudiante es elegible para un certificado de distinción.

```

fiorellamr
interface DistinctionDecider{
    2 usages  2 implementations  fiorellamr
    public void evaluateDistinction(Estudiante estudiante);
}

```

La clase ArtsDistinctionDecider implementó la interfaz DistinctionDecider, por lo que sobrescribió el método evaluateDistinction(Estudiante estudiante) que determina si un estudiante es elegible para un certificado de distinción según el score del estudiante, considerando que para el departamento de arte el score del estudiante debe ser mayor a 70.

```

1 usage  fiorellamr +1
public class ArtsDistinctionDecider implements DistinctionDecider{
    1 usage
    List<String> arts= Arrays.asList("Historia","Literatura");
    2 usages  fiorellamr +1
    @Override
    public void evaluateDistinction(Estudiante estudiante) {
        if (arts.contains(estudiante.department)) {
            if (estudiante.score > 70) {
                System.out.println(estudiante.regNumber+" ha recibido una distincion en artes.");
            }
        }
    }
}

```


La clase ScienceDistinctionDecider implementa la interface DistinctionDecider, se sobrescribe el método de evaluateDistinction que determina si el estudiante es elegible para un certificado de distinción, considerando ya que el puntaje de los estudiantes que pertenecen al departamento de ciencias debe ser mayor a 80 para considerar a un estudiante como elegible al certificado.

```
import Solid.OCP.Estudiante;

import java.util.Arrays;
import java.util.List;

1 usage 1 fiorellamr +1
public class ScienceDistinctionDecider implements DistinctionDecider {
    1 usage
    List<String> science= Arrays.asList("Ciencia de la Computacion.", "Fisica");
    2 usages 1 fiorellamr
    public void evaluateDistinction(Estudiante estudiante) {
        if (science.contains(estudiante.departament)) {
            if (estudiante.score > 80) {
                System.out.println(estudiante.regNumber+" ha recibido una distincion en ciencias.");
            }
        }
    }
}
```

Cliente.java:

En la clase Cliente, se implementaron 2 métodos, el primero llamado enrollScienceStudents() y el segundo llamado enrollArtsStudents().

En el primer método se instancian 2 objetos, de clase CienciaEstudiante, luego se añaden a una lista y ésta es retornada.

En el segundo método se instancian 2 objetos, de clase ArteEstudiante, luego se añaden a la lista y ésta es retornada.

```
1 usage 1 ArmandoLLuen
private static List<Estudiante> enrollScienceStudents() {
    Estudiante Irene = new CienciaEstudiante( name: "Irene", regNumber: "R1", score: 81.5, departament: "Ciencia de la computacion.");
    Estudiante jessica = new CienciaEstudiante( name: "Jessica", regNumber: "R2", score: 72, departament: "Fisica");
    List<Estudiante> CienciasEstudiantes = new ArrayList<>();
    CienciasEstudiantes.add(Irene);
    CienciasEstudiantes.add(jessica);
    return CienciasEstudiantes;
}

1 usage 1 ArmandoLLuen
private static List<Estudiante> enrollArtsStudents() {
    Estudiante chalo = new ArteEstudiante( name: "Chalo", regNumber: "R3", score: 71, departmant: "Historia");
    Estudiante claudio = new ArteEstudiante( name: "Claudio", regNumber: "R4", score: 66.5, departmant: "Literatura");
    List<Estudiante> ArtesEstudiantes = new ArrayList<>();
    ArtesEstudiantes.add(chalo);
    ArtesEstudiantes.add(claudio);
    return ArtesEstudiantes;
}
```

11. La ventaja es que al convertir DistinctionDecider() en una interfaz, por la abstracción nos permite definir los métodos de acuerdo a las nuevas clases que se quieran implementar, lo que evita que tengamos que modificar el método al agregar un departamento ya que sería un método extenso, lo cual dificultaría trabajar con ellos y se estaría duplicando el código, lo que lo convertiría en un código menos eficiente.

12.

El resultado obtenido con los métodos dados en la imagen anterior, muestra que tanto Abejita, Chalito no muestran detalles en los últimos detalles de pagos y actuales solicitud de pagos, esto se debe a que la clase GuestUserPayment la cual implementa la interfaz Payment aún no está implementada.

```
Demostracion sin LSP

Recuperando de Abejita, ultimos detalles de pagos.
-----
Recuperando de Chalito, ultimos detalles de pagos.
-----+
-----
Procesando de Abejita, la actual solicitud de pagos .
-----
Procesando de Chalito, la actual solicitud de pagos .
-----
```

13. public 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.");
    }
}
```

Se está procesando la solicitud de pago actual de un usuario invitado.

14. El tipo de excepción con el que nos encontramos es `UnsupportedOperationException` y la solución a este problema es cambiar la línea en dónde se lanza este tipo de excepción por una impresión de un mensaje "Recuperando del usuario invitado, últimos detalles de pagos".
15. 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.

```
Demostracion LSP.

Recuperando de Irene, ultimos detalles de pagos.
-----
Recuperando de Claudio, ultimos detalles de pagos.
-----
Procesando de Irene, la actual solicitud de pagos .
-----
Procesando de Claudio, la actual solicitud de pagos .
-----
Procesando de guest pago actual.
-----
```

16. Resultado obtenido luego de la refactorización:

Los cambios realizados se explicarán en el siguiente punto.

17. Interfaz `PreviousPayment` :

contiene el método `previousPaymentInfo()`; el cual es utilizado para la clase `registeredUserPayment` y en la clase `PaymentHelper`.

Interfaz `NewPayment` :

Contiene el método `newPayment`; el cual es utilizado para la clase `registeredUserPayment` y en la clase `PaymentHelper`.

Clase `PaymentHelper` :

En esta clase se han instanciado dos listas, el primero llamado `newPayments`, cuya principal función es almacenar a los nuevos usuarios que hacen referencia al proceso de un nuevo pago impreso en el método `processNewPayments` mientras que en el segundo llamado `previousPayments`, cuya función es almacenar a los usuarios que cuentan con un historial de pagos y se imprime dicha información en el método `showPreviousPayments`.

“El principal cambio que se hizo fue separar las interfaces PreviousPayment y NewPayment de la interfaz Payment y con ello crear 2 listas en clase PaymentHelper para gestionar a los historiales de pagos pasados para un usuario específico en caso exista, o procesar nuevos pagos.

Clase RegisteredPaymentHelper :

Esta clase implementa las dos interfaces hablado anteriormente (NewPayment y PreviousPayment), es utilizado para registrar a los usuarios con sus nombres e imprime un mensaje de recuperando últimos detalles de pago en el caso de que el usuario haya tenido pagos realizados con anterioridad o sino procesando la solicitud actual de dicho usuario.

Clase GuestUserPayment :

Esta clase gestiona el pago de un usuario tipo invitado por esta razón implementa la interfaz NewPayment, se almacena el nombre de este usuario tipo invitado y utiliza un método para informar que esta solicitud de pago se está procesando.

Clase Cliente :

En esta clase se han Instanciando dos usuarios registrados, luego se ha Instanciando el pago de un usuario invitado, se consolida la información del pago anterior al helper, también se consolida nuevas solicitudes de pago al helper, por último se recupera todos los pagos anteriores de los usuarios registrados y se muestra el mensaje de que se procesa todas las solicitudes de pago nuevos de todos los usuarios.

18. Se pudo refactorizar fácilmente el código del cliente usando algún método estático, por ejemplo al realizar una modificación donde se utiliza un método estático para mostrar todas las solicitudes de pago y utilizar este método siempre y cuando se le necesite.
19. Al querer mostrar el tipo de fax que se está utilizando, la interfaz o clase que ya habíamos implementado no puede responder a los nuevos requerimientos del programa, por lo cual se hace necesario modificar parte de esta interfaz con el fin de satisfacer las nuevas necesidades del cliente.

```
C:\Users\clever\.jdk\openjdk-18.0.1.1\bin\java.exe "-javaagent:C:\Program Files\
Demostracion sin ISP
La impresora avanzada imprime un documento.
La impresora avanzada envía un fax.
La impresora basica imprime un documento.
Exception in thread "main" java.lang.UnsupportedOperationException Create breakpoint
    at NoSolid.ISP.ImpresoraBasica.sendFax(ImpresoraBasica.java:11)
    at NoSolid.ISP.Cliente.main(Cliente.java:16)

Process finished with exit code 1
```

20. El problema que surge es que el método `sendFax` se está implementando en la clase `Impresora Básica`, lo cual al llamar al método, lanza una excepción pues una impresora básica sólo puede imprimir y no soporta la operación de enviar fax.
21. La interfaz `impresora`, al tratar de dar soporte a todas las clases de impreso está generando un error de segregación, pues algunas clase implementarán métodos que no van a utilizar y que en su defecto lanzan una excepción de `UnsupportedOperationException()`. Lo que debemos hacer es generar nuevas interfaces para segregar los métodos y así poder implementarlos solo en las clases que los van a utilizar.
22. No es conveniente, debido a que si usamos ese segmento de código haremos que la impresora básica implemente el método `sendFax()` a pesar que no se va a utilizar, en consecuencia obligaremos a la clase a poner una excepción cuando se llame a este método.
23. Dado que la impresora avanzada necesita la implementación de métodos que la impresora básica no requiere, podemos agrupar los métodos comunes entre ambos tipos de impresora en una interfaz y hacer otras interfaces específicas para cada tipo de impresora. Otra manera es hacer que interfaz específica para cada impresora herede de la interfaz que tiene métodos comunes o generales de cada impresora.
24. Los resultados son los siguientes:

```
class Cliente {  
    @ ArmandoLlue *  
    public static void main(String[] args) {  
        EFax eFax = new EFax();  
        System.out.println("Demostracion sin ISP");  
  
        Impresora impresora = new ImpresoraAvanzada();  
        impresora.printDocument();  
        impresora.sendFax(eFax);  
  
        impresora = new ImpresoraBasica();  
        impresora.printDocument();  
        impresora.sendFax(eFax); // Lanza un error  
    }  
}
```

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

Demostracion sin ISP

La impresora avanzada imprime un documento.

La impresora avanzada env a un fax.

La impresora basica imprime un documento.

```
Exception in thread "main" java.lang.UnsupportedOperationException C
    at NoSolid.ISP.ImpresoraBasica.sendFax(ImpresoraBasica.java:11)
    at NoSolid.ISP.Cliente.main(Cliente.java:17)
```

Demostraci n sin ISP

La impresora avanzada imprime un documento.

La impresora avanzada env a un fax.

La impresora basica imprime un documento.

```
Exception in thread "main" java.lang.UnsupportedOperationException
    at NoSolid.ISP.ImpresoraBasica.sendFax(ImpresoraBasica.java:11)
    at NoSolid.ISP.Cliente.main(Cliente.java:25)
```

25. La expresi n lambda es:

```
impresoras.forEach((dispositivo)->{dispositivo.printDocument();
    dispositivo.sendFax(eFax);});
```

y el resultado es:

Demostracion sin ISP

La impresora avanzada imprime un documento.

La impresora avanzada env a un fax.

La impresora basica imprime un documento.

```
Exception in thread "main" java.lang.UnsupportedOperationException
    at NoSolid.ISP.ImpresoraBasica.sendFax(ImpresoraBasica.java:11)
    at NoSolid.ISP.Cliente.lambda$main$0(Cliente.java:30)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511)
    at NoSolid.ISP.Cliente.main(Cliente.java:29)
```

26. En la interfaz impresora se crean los métodos printDocument() y sendFax que serán implementados en las clases impresora básica e impresora avanzada.

```
interface Impresora {  
    2 usages 2 implementations ArmandoLlue  
    void printDocument();  
  
    2 usages 2 implementations ArmandoLlue  
    void sendFax(Fax faxType);  
}
```

La clase impresora avanzada implementa la interfaz impresora con los métodos printDocument y sendFax.

```
public class ImpresoraAvanzada implements Impresora {  
    2 usages ArmandoLlue  
    @Override  
    public void printDocument() {  
        System.out.println("La impresora avanzada imprime un documento.");  
    }  
  
    2 usages ArmandoLlue  
    @Override  
    public void sendFax(Fax sendType) {  
        System.out.println("La impresora avanzada envía un fax.");  
    }  
}
```

La clase impresora básica también implementa la interfaz impresora con todo sus métodos, notar que para el método sendFax() se lanza una excepción pues este tipo de impresora no puede enviar faxes.

```
class ImpresoraBasica implements Impresora {  
    2 usages ArmandoLlue  
    @Override  
    public void printDocument() {  
        System.out.println("La impresora basica imprime un documento.");  
    }  
  
    2 usages ArmandoLlue  
    @Override  
    public void sendFax(Fax faxType) {  
        throw new UnsupportedOperationException();  
    }  
}
```

La clase JerarquíaFax crea una interfaz Fax y además despliega 2 clases que implementan esta interfaz para diferenciar los tipos de comunicación que está empleando al enviar un documento por Fax.

```

interface Fax {
    2 implementations  ArmandoLlue
    void sendFax(Fax faxType);
}

ArmandoLlue
class LanFax implements Fax {
    ArmandoLlue
    @Override
    public void sendFax(Fax faxType) {

}

}

2 usages  ArmandoLlue
class EFax implements Fax {
    ArmandoLlue
    @Override
    public void sendFax(Fax faxType) {

```

La clase cliente instancia un objeto eFax que se usará para pasar como parametro a los sendFax() de las impresoras, luego se instancia un objeto impresoraAvanzada y se manda a imprimir un documento y enviar fax, por último se hace lo mismo con una instancia de impresoraBasica.

```

class Cliente {
    ArmandoLlue *
    public static void main(String[] args) {
        EFax eFax = new EFax();
        System.out.println("Demostracion sin ISP");

        Impresora impresora = new ImpresoraAvanzada();
        impresora.printDocument();
        impresora.sendFax(eFax);

        impresora = new ImpresoraBasica();
        impresora.printDocument();
        impresora.sendFax(eFax); // Lanza un error
    }
}

```

El resultado obtenido es:

```

La impresora avanzada imprime un documento.
La impresora avanzada envía un fax.
La impresora basica imprime un documento.
Exception in thread "main" java.lang.UnsupportedOperationException Cr
    at NoSolid.ISP.ImpresoraBasica.sendFax(ImpresoraBasica.java:11)|
    at NoSolid.ISP.Cliente.main(Cliente.java:17)

```

En este caso vemos que la impresora avanzada imprime el documento y envía el fax, pero en el caso de impresora básica solo imprime el documento más no envía el

fax, en lugar de eso lanza una excepción pues en la clase impresora se implementó que cuando se llame al método sendFax() lance una excepción.

27. En la interfaz DispositivoFax se definió el método sendFax() de tipo void:

```
ArmandoLlue +1
interface DispositivoFax {
    1 usage 1 implementation fiorellamr
    void sendFax();
}
```

En la interfaz Impresora se definieron los métodos printDocument() y fotocopiado():

```
fiorellamr +1
interface Impresora {
    2 usages 2 implementations fiorellamr
    void printDocument();

    2 implementations fiorellamr
    void fotocopiado();
}
```

La clase ImpresoraAvanzada implementó las interfaces Impresora y DispositivoFax, por lo que se tuvo que implementar el método printDocument() que indica que la impresora está imprimiendo un documento, el método sendFax() que indica que está enviando a fax y el método fotocopiado, que indica que la impresora está fotocopiando.

```
fiorellamr +1
public class ImpresoraAvanzada implements Impresora, DispositivoFax{
    2 usages fiorellamr
    public void printDocument(){
        System.out.println("La impresora avanzada imprime un documento.");
    }
    1 usage fiorellamr
    public void sendFax(){
        System.out.println("Enviando a FAX.");
    }
    fiorellamr
    public void fotocopiado() { System.out.println("La impresora avanzada está fotocopiando."); }
}
```

La clase ImpresoraBasica implementa la interfaz Impresora, por lo que implementó el método printDocument() que indica que imprimió un documento y el método fotocopiando() que indica que la impresora básica está fotocopiando.

```
package Solid.ISP;

@fiorellamr +1
public class ImpresoraBasica implements Impresora{
    2 usages @fiorellamr
    public void printDocument(){
        System.out.println("La impresora basica imprime un documento.");
    }

    @fiorellamr
    public void fotocopiando(){
        System.out.println("La impresora basica está fotocopiando.");
    }
}
```

En la clase cliente se tiene el objeto impresora que se instanció con el constructor de ImpresoraBasica(), luego se llama a la función printDocument(), que muestra que la impresora básica imprime un documento, después la impresora se instanció con el constructor ImpresoraAvanzada() y se llamó al método printDocument() que muestra que la impresora avanzada imprime un documento.

Se instanció un objeto de clase DispositivoFax con el constructor de ImpresoraAvanzada llamado fax y se llamó al método sendFax(), el cual imprime que la impresora avanzada envía Fax.

```
ArmandoLlueen +1
public class Cliente {
    ArmandoLlueen +1
    public static void main(String[] args) {
        System.out.println("Demostracion ISP");

        Impresora impresora = new ImpresoraBasica();
        impresora.printDocument();

        impresora = new ImpresoraAvanzada();
        impresora.printDocument();

        DispositivoFax fax = new ImpresoraAvanzada();
        fax.sendFax();
    }
}

C:\Users\Usuario\.jdk\corretto-17.0.3\bin\java.exe "-javaagent:C:\
Demostracion ISP
La impresora basica imprime un documento.
La impresora avanzada imprime un documento.
La impresora avanzada envía FAX

Process finished with exit code 0
```

28. Si se agregara un método determinado en la interfaz, no se necesitaría definir el método en la clase que implementa, sino que haría referencia al método predeterminado. Además se tendría la opción de sobrescribirlo en caso se desee modificar la operación de realiza.

29. Si proporcionamos un método de fax predeterminado, éste no se ejecutará debido a que la excepción `UnsupportedOperationException` termina la ejecución el programa.

Método predeterminado en la interface Fax:

```
ArmandoLlueen *
interface Fax {
    1 usage new *
    default void printFax(){
        System.out.println("Imprimiendo desde fax");
    }
}
ArmandoLlueen *
```

Con el objeto impresora de clase ImpresoraAvanzada() hacemos referencia al método printFax(), luego se define como una instancia de ImpresoraBasica() y hacemos referencia al método printFax():

```
public static void main(String[] args) {  
    EFax eFax = new EFax();  
    System.out.println("Demostracion sin ISP");  
  
    Impresora impresora = new ImpresoraAvanzada();  
    impresora.printDocument();  
    impresora.sendFax(eFax);  
    impresora.printFax();  
  
    impresora = new ImpresoraBasica();  
    impresora.printDocument();  
    impresora.sendFax(eFax);  
    impresora.printFax();  
}
```

Salida del programa:

```
Demostracion sin ISP  
La impresora avanzada imprime un documento.  
La impresora avanzada envía un fax.  
Imprimiendo desde fax  
La impresora basica imprime un documento.  
Exception in thread "main" java.lang.UnsupportedOperationException Create breakpoint  
    at NoSolid.ISP.ImpresoraBasica.sendFax(ImpresoraBasica.java:11)  
    at NoSolid.ISP.Cliente.main(Cliente.java:18)  
  
Process finished with exit code 1
```

30. Si usamos un método vacío en vez de usar una excepción, se no se se terminaría la ejecución del problema como en el caso indicado anteriormente, sino que la ejecución del programa continuaría y la instancia de ImpresoraBasica(), llamada impresora, hace referencia al método printFax() y éste es mostrado en la salida.

```

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

    2 usages  Armandolluen *
    @Override
    public void sendFax(Fax faxType) {
    }
}

```

Salida:

```

Demostracion sin ISP
La impresora avanzada imprime un documento.
La impresora avanzada envía un fax.
Imprimiendo desde fax
La impresora basica imprime un documento.
Imprimiendo desde fax

Process finished with exit code 0

```

Los métodos vacíos, al no tener una funcionalidad no siguen el principio ICP y dado que no nos brindan información, no ayuda en el proceso de refactorización.

31.

```

A demo without DIP.
El id: E001 es guardado en la base de datos Oracle.

Process finished with exit code 0

```

InterfazUsuario:

En esta clase se declara un atributo oracleDatabase del tipo OracleDatabase el cual es instanciado en el constructor InterfazUsuario y además posee el método saveEmployeeId con parámetro empID del tipo string, cuya funcionalidad de este método es guardar el id en la base de datos de Oracle.

Cliente:

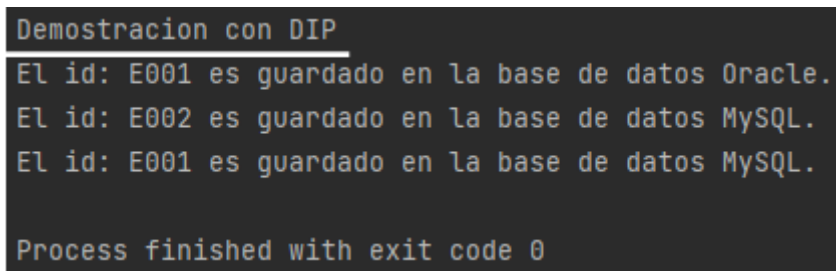
Se instala el objeto usuario de la clase InterfazUsuario y se le guarda el ID de empleado.

OracleDatabase:

Esta clase tiene un método saveEmplIdInDatabase con parámetro emplId del tipo String, este método tiene la función de mostrar un mensaje de que el ID del empleado se ha guardado en la base de datos de Oracle.

32. El problema de este programa básico es que la clase InterfazUsuario depende de la clase OracleDatabase y viceversa, por esta razón la solución sería que haya dependencia de abstracciones.

33.



```
Demostracion con DIP
El id: E001 es guardado en la base de datos Oracle.
El id: E002 es guardado en la base de datos MySQL.
El id: E001 es guardado en la base de datos MySQL.

Process finished with exit code 0
```

Como resultado se quitó la dependencia que se tenía entre las clases InterfazUsuario y OracleDatabase permitiendo que se puedan implementar otras base de datos como MySQL u otras más; como por ejemplo la imagen de arriba muestra que el ID 'E001' ha sido inicialmente guardado en ORACLE y luego es guardado en MYSQLe.

34.

Interfaz BaseDatos:

Se implementó un método saveEmplIdInDatabase(String emplId) cuya finalidad es sobrescribir este método e implementarlo de acuerdo a las clases que implementan esta interfaz.

Clase Cliente:

Se instanciaron dos usuarios para almacenar sus respectivos ID de empleado en las dos diferentes bases de datos (Oracle y MySQL), luego el objetivo base de datos de un usuario fue cambiado, es decir; al principio el ID de empleado del usuario 1 fue almacenado en Oracle y luego fue almacenado en MySQL.

Clase InterfazUsuario:

se crearon dos tipos de base de datos, baseDatosOracle y baseDatosMysql; se implementaron dos constructores en el primero se pasa como atributo oracleDatabase de la clase OracleDatabase y en el segundo mySQL Database de la clase MySQLDatabase en donde los atributos baseDatosOracle y baseDatosMysql son instanciados. Luego se crearon dos métodos saveEmployeeIdInMySQL y saveEmployeeIdInOracle cuya función es guardar el ID del empleado.

Clase MySQLDatabase:

Esta clase implementa la interfaz BaseDatos y sobre escribe su método y es usado para informar que el ID de empleado fue guardado en la base de datos MySQL.

Clase OracleDatabase:

Esta clase implementa la interfaz BaseDatos y sobre escribe su método y es usado para informar que el ID de empleado fue guardado en la base de datos Oracle.

35. Podría ser cuando las clase de alto nivel necesitan instanciar un objeto de la clase base para hacer operaciones con ello debido a la dependencia de una clase más grande.
36. El beneficio es que al momento de cambiar el objetivo de la base de datos para un usuario ya no sería necesario instanciarlo hacia el nuevo objetivo de la base datos sino que solo llamaría al método setDatabase, para especificar a qué nueva base de datos se desea almacenar el identificador de este usuario.