

Taller 3 – SOLID

Integrantes:



- ✓ **Alex Peñafiel**
- ✓ **Juan Urgiles**
- ✓ **Alex Benites**

Contenido

Sección A:3

Sección B:12

Repositorio en Git:16

Sección A:

Primer Principio: Single responsibility principle

```
package solid;

class UserLogin {

    private final DataBase db;

    UserLogin(DataBase db) {
        this.db = db;
    }

    void login(String userName, String password) {
        User user = db.findUserByUserName(userName);
        if (user == null) {
            // do something
        }
        // login process..
    }

    void sendEmail(User user, String msg) {
        // sendEmail email to user
    }

}
```

Observaciones: bueno en este código se observa que la clase UserLogin tiene como responsabilidad realizar el proceso de “login” pero además también se le esta asignando otra tarea o responsabilidad como la de enviar mensajes al usuario “sendEmail”. Con lo cual no cumple con el primer principio ya que esta clase tiene o está tratando de realizar dos tareas (responsabilidades).

Solución: lo que se podría hacer es crear una clase por ejemplo EmailEnviados la cual se encargue del método “sendEmail” dejando la clase principal solo con una responsabilidad que del método “login”.

```
package solid;

class EmailSender {

    void sendEmail(User user, String msg) {
        // send email to user
    }

}
```

Segundo Principio: Open/closed principle

```
package solid;

class Car {

    void accelerate() {
        // accelerate car
    }

    void stop() {
        // stop car
    }

}
```

Observaciones: A primera vista pareciera que la clase no tendría ningún problema, pero al momento agregar una nueva funcionalidad o tarea para un objeto diferente como un vehículo de carreras *RaceCar* podríamos pensar en tal vez colocar un método `injectExtraGas`

```
package solid;

class Car {

    void accelerate(boolean isCarRace) {
        if (isCarRace) {
            injectExtraGas();
        }
        // accelerate car
    }

    void stop() {
        // stop car
    }

    private void injectExtraGas() {
        // do..
    }

}
```

pero eso estaría mal ya que estuviéramos violando la segunda ley y al momento de crear un nuevo vehículo que no sea de carrera este va a tener que implementar por obligación el método “`injectExtraGas`” pero estaría mal ya que ese método solo está pensado para un tipo de objeto en particular como lo son los vehículos de carreras y no para un vehículo normal.

Solución: en vez de tratar de poner ese método “`injectExtraGas`” en la clase `Car` podríamos extender nuestra clase creando otro tipo de clase derivada de la principal. Podríamos crear una clase *RaceCar* que va a extender de *Car* y allí agregamos el método “`injectExtraGas`”.

Al extender la clase *Car* estamos asegurando que quien este usando previamente *Car* no se vea afectado por el cambio.

```
package solid;

public class RaceCar extends Car {

    @Override
    void accelerate() {
        injectExtraGas();
        super.accelerate();
    }

    private void injectExtraGas() {
        // do..
    }

}
```

Tercer principio: Liskov substitution principle

Observación: Bueno usando la clase anterior “Car” creamos una interfaz y dos implementaciones.

La primera para un auto común en la clase *Car* y la segunda para un auto eléctrico en la clase *ElectricCar*.

```
package solid.l;

interface ICar {

    void accelerate();

    void stop();

}
```

```
package solid.l;

public class Car implements ICar {

    @Override
    public void accelerate() {
        System.out.println("accelerating the car");
    }

    @Override
    public void stop() {
        System.out.println("stopping the car");
    }

}
```

Bueno ahora se observa o se identifica que el auto eléctrico necesita verificar el estado de la batería cosa que hacemos en el método *hasBattery*.

```

package solid.1;

public class ElectricCar implements ICar {

    private int battery;

    @Override
    public void accelerate() {
        System.out.println("accelerating the car");
    }

    @Override
    public void stop() {
        System.out.println("accelerating the car");
    }

    public boolean hasBattery() {
        System.out.println("checking battery");
        if (battery < 95) {
            System.out.println("the battery is very low :(");
            return false;
        } else {
            System.out.println("battery OK :)");
            return true;
        }
    }
}

```

Con estos cambios se puede ver que en el caso del auto eléctrico necesitamos invocar el método *hasBattery()* para luego poder acelerar. Pero con este diseño de clases se estaría rompiendo el principio de sustitución porque necesitamos explícitamente conocer el tipo de vehículo y no podemos reemplazar la clase *ElectricCar* con la interfaz *ICar*

```

package solid.1;

public class CarDrive {
    public static void main(String[] args) {

        String cardType = args[0];
        if ("car" == cardType) {
            Car car = new Car();
            car.accelerate();
        } else if ("electric" == cardType) {
            ElectricCarBad electricCar = new ElectricCarBad();
            if ((electricCar.hasBattery())) {
                electricCar.accelerate();
            }
        } else {
            throw new RuntimeException("Invalid car");
        }
    }
}

```

Solución:

```
package solid.1;

public class ElectricCar implements ICar {

    private int battery;

    @Override
    public void accelerate() {
        if (hasBattery()) {
            System.out.println("accelerating the car");
        } else {
            System.out.println("I can not accelerate the car");
        }
    }

    @Override
    public void stop() {
        System.out.println("accelerating the car");
    }

    private boolean hasBattery() {
        System.out.println("checking battery");
        if (battery < 95) {
            System.out.println("the battery is very low :(");
            return false;
        } else {
            System.out.println("battery OK :)");
            return true;
        }
    }
}
```

ahora se puede ver que ya podemos usar el principio de sustitución ya que para acelerar el auto no se necesita conocer el tipo de clase.

```
package solid.1;

public class CarDrive {
    public static void main(String[] args) {
        ICar car;
        String cardType = args[0];
        if ("car" == cardType) {
            car = new Car();
        } else if ("electric" == cardType) {
            car = new ElectricCar();
        } else {
            throw new RuntimeException("Invalid car");
        }
        car.accelerate();
    }
}
```

Cuarto Principio: Interface segregation principle

Para el ejemplo de este principio se tiene el siguiente ejemplo:

Se crea una interfaz *IProduct* de la cual luego crearemos sus clases que la implementan.

```
package solid.i;  
public interface IProduct {  
    String getType();  
}
```

Ahora se Implementa un producto *Shoes* desde la interfaz *IProduct* de la siguiente manera

```
package solid.i;  
class Shoes implements IProduct {  
    @Override  
    public String getType() {  
        return "shoes";  
    }  
}
```

Después se Implementa otro producto *Games* desde la interfaz *IProduct*

```
package solid.i;  
class Games implements IProduct {  
    @Override  
    public String getType() {  
        return "game";  
    }  
}
```

Observación: Ahora se necesita que la clase *Games* también implemente un método *getAge()* para conocer para que edad son los juegos. Una forma simple sería agregar a la interfaz *IProduct* el método de este modo, pero estaría incorrecto. Por lo que nos obligaría a implementar el método *getAge()* también en todas las clases.

```
package solid.i;  
public interface IProduct {  
    String getType();  
    int getAge();  
}
```

Se lo implementa en la clase *Games* , en dónde si aplica este método.


```
package solid.i;
class Games implements IProduct {
    private int age;
    @Override
    public String getType() {
        return "game";
    }
    @Override
    public int getAge() {
        return age;
    }
    // get and set..
}
```

Pero también se ve obligados a usarlo en esta clase *Shoes* que NO lo necesita.

```
package solid.i;
class Shoes implements IProduct {
    @Override
    public String getType() {
        return "shoes";
    }
    @Override
    public int getAge() {
        throw new UnsupportedOperationException();
    }
}
```

Solución:

Lo que se podría hacer para solucionar esta violación es crear otra interfaz para el caso de productos que requieran la edad *getAge()*, así solo las clases que necesiten la restricción por edad lo implementarán. De este modo no se obliga a ninguna clase a implementar el método que no utilizará.

```
package solid.i;

public interface IProduct {

    String getType();

}
```

...

```
package solid.i;

public interface IRestrictedProduct {

    int getAge();

}
```

La clase *Shoes* solo implementa la interfaz *IProduct* y no necesita implementar *IRestrictedProduct*

```
package solid.i;

class Shoes implements IProduct {

    @Override
    public String getType() {
        return "shoes";
    }

}
```

La clase *Games* ahora implementa las dos interfaces.

```
package solid.i;

class Games implements IProduct, IRestrictedProduct {

    private int age;

    @Override
    public String getType() {
        return "game";
    }

    @Override
    public int getAge() {
        return age;
    }

    // get and set..

}
```

Quinto principio: Dependency inversion principle.

Bueno para el siguiente ejemplo se observa que se tiene una clase Cash que recibe un producto y un método de pago luego dentro de ella se instancia la base de datos para persistir el producto con su respectivo método de pago.

Observación: en la imagen muestra cómo se está haciendo uso directo de Database directamente de su implementación con lo cual se estaría utilizando un código de clases de bajo nivel MySQLDatabase. Por lo que un posible inconveniente que podría ocurrir es que en un determinado tiempo si yo quisiera cambiarme a otras bases de datos debería modificar el código también por lo que la manera en que MySQL implementa ese método es diferente a como lo implementaría por ejemplo una base de datos como PostgreSQL o Oracle.

También vemos que estamos combinando diferentes niveles de clases lo cual viola el principio solid de dependencia

```
package solid.d;

class Cash {

    public void pay(Product product, PaymentType paymentType) {

        MySQLDatabaseBad persistence = new MySQLDatabaseBad();
        persistence.save(product, paymentType);

    }

}
```

```
package solid.d;

class MySQLDatabase{

    void save(Product product, PaymentType paymentType) {
        System.out.println("Save product " + product + " paymentType " + paymentType);
        // save into MySQLDatabase...
    }

}
```

Solución:

Implementando el principio de inversion de dependencias lo que podriamos hacer es crear una interfaz que desacople de la persistencia.

Implementamos la siguiente:

Se crea la Interfaz de persistencia y la implementamos en mysqlatabase. Agregando esa nueva interfaz la clase Cash ahora recibirá la interfaz Persistence en el constructor y no depende sola mente de los objetos por lo que mi clase no necesitaria saber cómo se implementa la persistencia.

```
package solid.d;

interface Persistence {

    void save(Product product, PaymentType paymentType);

}
```

```
package solid.d;

class MySQLDatabase implements Persistence {

    public void save(Product product, PaymentType paymentType) {
        System.out.println("Save product " + product + " paymentType " + paymentType);
        // save into MySQLDatabase...
    }

}
```

```

package solid.d;

class Cash {

    Persistence persistence;

    public Cash(Persistence persistence) {
        this.persistence = persistence;
    }

    public void pay(Product product, PaymentType paymentType) {

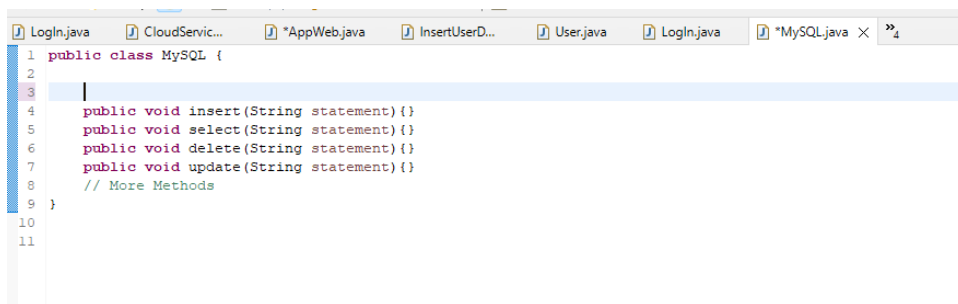
        persistence.save(product, paymentType);

    }

}

```

Sección B:



```

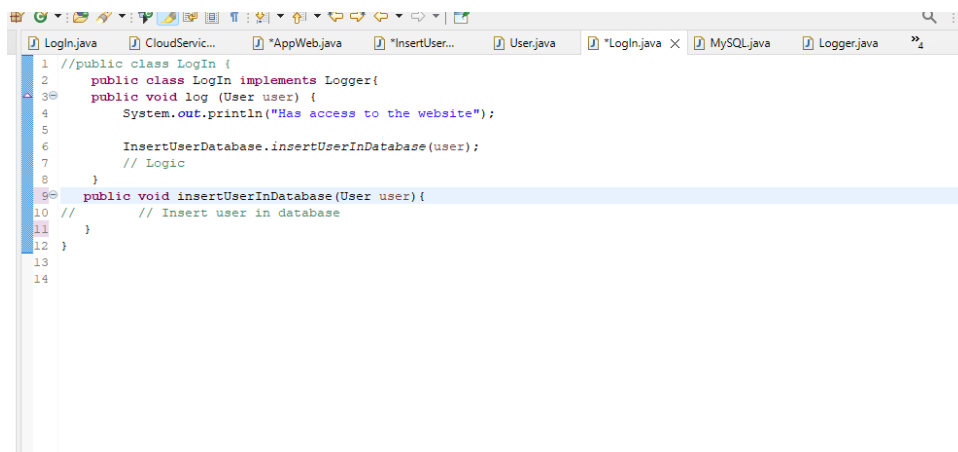
1 public class MySQL {
2
3
4     public void insert(String statement) {}
5     public void select(String statement) {}
6     public void delete(String statement) {}
7     public void update(String statement) {}
8     // More Methods
9 }
10
11

```

*

principio SOLID :

Principio S: (single responsibility principle)



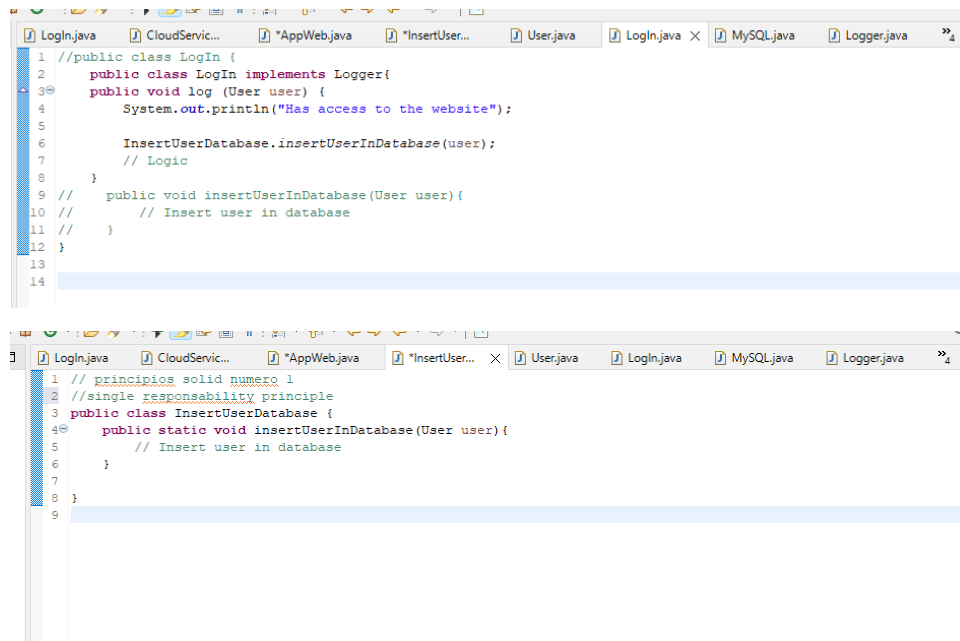
```

1 //public class LogIn {
2     public class LogIn implements Logger{
3     public void log (User user) {
4         System.out.println("Has access to the website");
5
6         InsertUserDatabase.insertUserInDatabase(user);
7         // Logic
8     }
9     public void insertUserInDatabase(User user){
10    //    // Insert user in database
11    }
12 }
13
14

```

La clase tiene mas de un proposito lo cual viola al primer principio solid

La solución para el problema sería crear una clase aparte que se encargue de ejecutar la función “insertUserInDatabase” para corregir el error

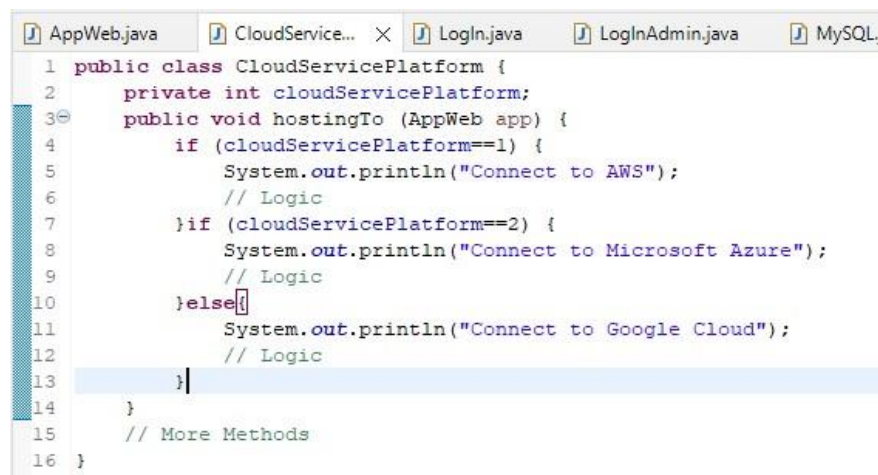


```
1 //public class LogIn {
2     public class LogIn implements Logger{
3         public void log (User user) {
4             System.out.println("Has access to the website");
5
6             InsertUserDatabase.insertUserInDatabase(user);
7             // Logic
8         }
9     }
10    // public void insertUserInDatabase(User user){
11    //    // Insert user in database
12    // }
13 }
14
```

```
1 // principios solid numero 1
2 //single responsibility principle
3 public class InsertUserDatabase {
4     public static void insertUserInDatabase(User user){
5         // Insert user in database
6     }
7 }
8
9
```

Principio O: (Open-Close Principle)

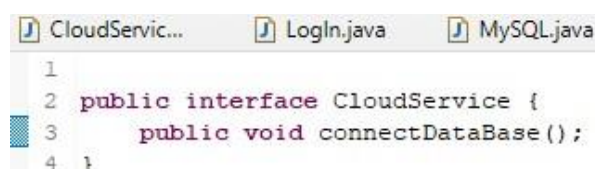
La clase cloudServicePlataform viola el principio de open-close ya que si se intenta agregar otra plataforma se tendría que modificar el método hostingTo por lo que el método no esta cerrado a modificación.



```
1 public class CloudServicePlatform {
2     private int cloudServicePlatform;
3     public void hostingTo (AppWeb app) {
4         if (cloudServicePlatform==1) {
5             System.out.println("Connect to AWS");
6             // Logic
7         }if (cloudServicePlatform==2) {
8             System.out.println("Connect to Microsoft Azure");
9             // Logic
10        }else{
11            System.out.println("Connect to Google Cloud");
12            // Logic
13        }
14    }
15    // More Methods
16 }
```

Solución:

Para solucionarlo primero se crea una interfaz llamada CloudService y se agrega el método connectDataBase.



```
1
2 public interface CloudService {
3     public void connectDataBase();
4 }
```

Se implementan diferentes clases que representan a cada proveedor y les implementamos el método connect a cada uno.

```
public class AWSService implements CloudService{

    @Override
    public void connectDataBase() {
        System.out.println("Connect to AWS");
    }

}

public class MAzureService implements CloudService{

    @Override
    public void connectDataBase() {
        System.out.println("Connect to Microsoft Azure");
    }

}

public class GCloudService implements CloudService{

    @Override
    public void connectDataBase() {
        System.out.println("Connect to Google Cloud");
    }

}
```

En la clase CloudServicePrataform agregamos un atributo de tipo cloudService y en el metodo hostingTo hacemos que la app se conecte a su respectiva base de datos.

```
public class CloudServicePlatform {
    private CloudService cloudService;
    public void hostingTo (AppWeb app) {
        cloudService.connectDataBase();
    }
    // More Methods
}
```

De esta manera si se llega a agregar otra base de datos basta con agregar otra clase para poder conectarse a ella, de esta manera el programa está abierto a extensión.

Principio L: (Liskov substitution principle)

Vemos que la clase hija trata de comportarse como clase padre y eso es un problema

```
*LogInAdmin...  *LogInAdmin... x  MySQLjava  User.java  VerificarUs...  AppV
1 public class LogInAdmin extends LogIn {
2 //public class LogInAdmin extends LogIn implements Logger{
3     private boolean userIsAdmin;
4     @Override
5
6
7     public void log (User user) {
8         this.userIsAdmin = VerificarUser.verifyTheUserIsAdmin(user);
9         if(!userIsAdmin){
10             return;
11         }
12         System.out.println("Has access to the website in admin mode");
13         // Logic
14     }
15 private boolean verifyIfTheUserIsAdmin(User user){
16 //         // Do something
17     return true;
18 }
19
20
21
```

```
LogInAdmin... x  MySQLjava  User.java  VerificarUs...  AppV
1 //public class LogInAdmin extends LogIn {
2 public class LogInAdmin extends LogIn implements Logger{
3     private boolean userIsAdmin;
4     @Override
5
6
7     public void log (User user) {
8         this.userIsAdmin = VerificarUser.verifyTheUserIsAdmin(user);
9         if(!userIsAdmin){
10             return;
11         }
12         System.out.println("Has access to the website in admin mode");
13         // Logic
14     }
15 // private boolean verifyIfTheUserIsAdmin(User user){
16 //         // Do something
17 //     return true;
18 }
19
20
21
```

Solución:

Podemos ver que usando el usuario como admin trata de a la clase hija como una clase padre y así resolvemos el problema

Principio D: (Dependency inversion principle)

Vemos que existe una violación al principio de dependencia lo cual los objetos login , loginadmin , mysql dependen de módulos de bajo nivel

```
public class AppWeb {
    LogIn logIn;
    LogInAdmin logInAdmin;
    MySQL mySQL;
    public AppWeb (LogIn logIn, MySQL mySQL) {
        // Logic
    }
    public AppWeb (LogInAdmin logInAdmin, MySQL mySQL) {
        // Logic
    }
    public void connectToDatabase (MySQL mySQL) {
        // Logic
    }
}
```

Como solución a este problema:

Crear interfaces no dependientes de los objetos previamente vistos así poder tener un código mas optimo y mas sencillo.

```
public interface Authentication {
}

public interface DatabaseConnection {
}
```

Repositorio en Git:

https://github.com/alex3952/taller3_Solid.git