SOLID Principles

Single Responsibility Principle (SRP):

- A class should have only one reason to change, meaning that a class should have only one responsibility.
- This principle encourages you to design classes that have a clear and specific purpose.

WRONG CODE:

```
public class Train {

    private int trainId;
    private String sourceStation;
    private String destinationStation;
    private Date date;
    private String seatClass;

    // Constructor
    public Train(int trainId, String sourceStation, String destinationStation, Date date, String
seatClass) {
        this.trainId = trainId;
        this.sourceStation = sourceStation;
        this.destinationStation = destinationStation;
        this.date = date;
        this.seatClass = seatClass;
    }

    // Violation of SRP: Booking functionality added to the Train class
    public void bookTicket() {
        // Additional logic for booking a ticket
        System.out.println("Ticket booked for Train ID: " + trainId);
    }
```

In this example, the **`bookTicket()`** method is added to the **`Train`** class, which violates the Single Responsibility Principle. Booking tickets is a separate responsibility, and it should ideally be handled by a class specifically designed for that purpose. A better design would involve creating a separate **`BookingService`** class or a similar construct to handle booking functionality, keeping the **`Train`** class focused on its primary responsibility of representing train information.

Now let us look at the **right code**:

```
public class Train {
 private int trainId;
 private String sourceStation;
 private String destinationStation;
 private Date date;
 private String seatClass;

 // Constructor
 public Train(int trainId, String sourceStation, String destinationStation, Date date,
String seatClass) {
 this.trainId = trainId;
 this.sourceStation = sourceStation;
 this.destinationStation = destinationStation;
 this.date = date;
 this.seatClass = seatClass;
 }
```

Storing Train Information:

- **`trainId`**, **`sourceStation`**, **`destinationStation`**, **`date`**, and **`seatClass`** are all attributes related to storing information about a train.

Open/Closed Principle (OCP):

- Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.
- This implies that you should be able to extend a system's behavior without modifying its existing code.

WRONG CODE:

```
public abstract class BookingService {

 public abstract void NormalbookTicket();

 public abstract void TatkalbookTicket();

}
```

Here if there is another type of booking to come in the future, then we would have to modify the class BookingService which is not preffered.

RIGHT CODE:

```
public abstract class BookingService {


 public abstract void bookTicket();

}

public class NormalBooking extends BookingService{


@Override
public void bookTicket() {
// TODO Auto-generated method stub
System.out.println("Normal Ticket Booked");
}
}

public class TatkalBooking extends BookingService {

@Override
public void bookTicket() {
// TODO Auto-generated method stub
System.out.println("Tatkal Ticket Booked");
}
}
```

- **Abstract Class `BookingService`:**
  - This class is designed as an abstract class, indicating that it is meant to be extended.
  - The **`bookTicket`** method is declared as an abstract method, meaning that its implementation is deferred to its subclasses.
- **Concrete Classes `NormalBooking` and `TatkalBooking`:**
  - These classes extend the **`BookingService`** abstract class.
  - They provide concrete implementations for the **`bookTicket`** method, fulfilling the contract defined in the **`BookingService`** abstract class.

By using this design, you can easily introduce new types of booking services without modifying the existing code. For example, if you wanted to add a new type of booking service, say, "VIPBooking," you can create a new class that extends **`BookingService`** and

provides its own implementation of the **`bookTicket`** method. The existing code that uses **`BookingService`** will not be affected, thus adhering to the Open/Closed Principle.

Liskov Substitution Principle (LSP):

- Subtypes must be substitutable for their base types without altering the correctness of the program.
- This principle ensures that objects of a derived class should be able to replace objects of the base class without affecting the program's functionality.

WRONG CODE:

```
public abstract class BookingService {

 public abstract void bookTicket();
 public abstract void isBookingValid();
```

Here the normalBooking does not have to check for the validation as normal ticket can be booked any time, but the TatkalBooking must be checked so that the date can be validated. Here the child class cannot be perfectly substituted for the parent class.

RIGHT CODE:

```
public abstract class BookingService {

 public abstract void bookTicket();

}

public class NormalBooking extends BookingService{


@Override
public void bookTicket() {
// TODO Auto-generated method stub
System.out.println("Normal Ticket Booked");
}
}

public class TatkalBooking extends BookingService implements TatkalBookingDate {


@Override
public void bookTicket() {
```

```
if (isBookingValid()) {
System.out.println("Tatkal date checked");
System.out.println("Ticket booked through TatkalBooking");
} else {
System.out.println("Booking is not valid. Cannot book the ticket.");
}
}

@Override
public boolean isBookingValid() {
return true;
}

public interface TatkalBookingDate {
boolean isBookingValid();
}
```

Liskov Substitution Principle (LSP) is one of the SOLID principles and it states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In your provided code, you have a class hierarchy with an abstract class **`BookingService`** and a subclass **`TatkalBooking`** implementing an interface **`TatkalBookingDate`**. The Liskov Substitution Principle is demonstrated as follows:

- **`BookingService` as a Superclass:**
  - **`BookingService`** is an abstract class with an abstract method **`bookTicket()`**.
  - The **`TatkalBooking`** class extends **`BookingService`**, inheriting the **`bookTicket()`** method.
- **`TatkalBooking` as a Subclass:**
  - **`TatkalBooking`** extends **`BookingService`** and implements the **`TatkalBookingDate`** interface.
  - It overrides the **`bookTicket()`** method with additional logic related to Tatkal booking.
- **Interface `TatkalBookingDate`:**
  - The **`TatkalBookingDate`** interface declares a method **`isBookingValid()`**.
  - The **`TatkalBooking`** class implements this interface and provides its own implementation of **`isBookingValid()`**.

The Liskov Substitution Principle is upheld in this design because you can use an instance of **`TatkalBooking`** wherever a **`BookingService`** is expected, and the behavior remains consistent. Additionally, the **`TatkalBooking`** class adheres to the contract defined by the **`TatkalBookingDate`** interface.

Interface Segregation Principle (ISP):

- A client should not be forced to implement interfaces it does not use.
- This suggests that a class should not be forced to implement interfaces with methods it does not need.

WRONG CODE:

```
public interface Ticket {
public void bookTicket();
public void cancelTicket();
}
```

Since there is both bookTicket and cancelTicket in Ticket class, when we want to book a ticket, the cancel ticket remains unimplemented and similarly in the case of cancelling a ticket. This is not preffered as there is unimplemented methods in a class. Each Interface should have a specific method.

RIGHT CODE:

```
public interface BookTicket {
public void bookTicket();

}

public interface CancelTicket {
public void cancelTicket();
}

public class NormalBooking implements BookTicket {

@Override
public void bookTicket() {
// TODO Auto-generated method stub
System.out.println("Normal Ticket Booked");
}
}

public class NormalTicketCancellation implements CancelTicket{

@Override
public void cancelTicket() {
// TODO Auto-generated method stub
System.out.println("Your ticket has been cancelled.");

}
}
```

```
public class TatkalBooking implements BookTicket {

@Override
public void bookTicket() {
// TODO Auto-generated method stub
System.out.println("Tatkal Ticket Booked");
}
}
```

The Interface Segregation Principle (ISP) states that a class should not be forced to implement interfaces it does not use. In your provided code, it seems that you are adhering to the ISP because each class implements only the interfaces that are relevant to its behavior. Let's break down how the code follows the Interface Segregation Principle:

- **`BookTicket` Interface:**
  - Declares a method **`bookTicket()`**.
- **`CancelTicket` Interface:**
  - Declares a method **`cancelTicket()`**.
- **`NormalBooking` Class:**
  - Implements the **`BookTicket`** interface.
  - Provides a concrete implementation for the **`bookTicket()`** method, as it is a class responsible for booking normal tickets.
- **`NormalTicketCancellation` Class:**
  - Implements the **`CancelTicket`** interface.
  - Provides a concrete implementation for the **`cancelTicket()`** method, as it is a class responsible for canceling normal tickets.
- **`TatkalBooking` Class:**
  - Implements the **`BookTicket`** interface.
  - Provides a concrete implementation for the **`bookTicket()`** method, as it is a class responsible for booking Tatkal tickets.

In this design, each class implements only the interface(s) that are relevant to its functionality. This adheres to the Interface Segregation Principle because no class is forced to implement methods that it does not use. Each interface represents a specific aspect of ticket-related functionality, and classes implementing these interfaces are only concerned with the methods relevant to their purpose.

Dependency Inversion Principle (DIP):

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

- This principle encourages the use of dependency injection and programming to interfaces.

WRONG CODE:

If the method was defined in the DatabaseRepository, then we there is a dependency on the interface, and when we want to change the database we have to depend on this interface and modify it according to our need, that is not preffered

```
public interface DatabaseRepository {
void saveToDatabase(){
System.out.println("Saved to mongoDb database");

	}

}
```

RIGHT CODE:

```
public interface DatabaseRepository {
void saveToDatabase();
}

public class MongoDb implements DatabaseRepository {
public void saveToDatabase(){
System.out.println("Saved to mongoDb database");
}
}

public class MySqlDb implements DatabaseRepository {
public void saveToDatabase(){
System.out.println("Saved to MySQL database");
}
}
```

the Dependency Inversion Principle is implemented as follows:

- **`DatabaseRepository` Interface:**
  - Declares a method **`saveToDatabase()`**, providing an abstraction for saving to a database.
- **`MongoDb` Class and `MySqlDb` Class:**
  - Implement the **`DatabaseRepository`** interface.

- Provide concrete implementations for the `saveToDatabase()` method specific to MongoDB and MySQL databases.

By adhering to the Dependency Inversion Principle, you've created an abstraction (`DatabaseRepository` interface) that both high-level modules (code that uses the repository) and low-level modules (concrete database implementations) depend on. This allows for flexibility and easy substitution of different database implementations without affecting the higher-level code.