SOLID Principles Case Study – Travel Booking Website

1. Single Responsibility

This principle states that a class should only have one responsibility. The class should only have one reason to change.

Example:

```
public class User {
private String userId;
private String username;
private String emailId;
private String password;
private String mobileNumber;
private String securityQuestion;
private String securityAnswer;
//getters and setters
//constructors
}
```

Here, the primary responsibility of the **User** class appears to be representing user data, likely for authentication or user management purposes. The single responsibility of the **User** class is to represent and manage user data. It adheres to the idea that a class should have only one reason to change, which, in this case, is related to changes in the structure or management of user data.

2. Open-Closed Principle

This principle states that classes should be open for extension but closed for modification. In doing so, we stop ourselves from modifying existing code and causing potential new bugs in an otherwise happy application.

Example:

```
public interface CustomerActions {
public void book();
public void cancel();
}
```

```
public class Customer extends User implements CustomerActions {
private String customerFirstName;
private String customerLastName;
private String customerAddress;
private String customerPaymentDetails;
private ArrayList<Booking> customerBookings;

        //getters and setters
        //constructor
        @Override
public void cancel() {
System.out.println("Customer can cancel booking");
}
@Override
public void book() {
System.out.println("Customer can make booking");
}
}
```

In summary, the Customer class adheres to the Open/Closed Principle by extending the functionality of the User class for specific customer-related properties and actions without modifying the base class. It allows for the introduction of new customer-specific behavior through interfaces and method overrides, promoting extensibility and maintainability.

3. Liskov Substitution Principle

This principle states that subtypes should be substitutable for their base types without changing the correctness of the program. It ensures that objects of a base class can be replaced with objects of a derived class without affecting the program's behavior.

Example:

```
public class User {
private String userId;
private String username;
private String emailId;
private String password;
private String mobileNumber;
private String securityQuestion;
private String securityAnswer;
```

```
        //getters and setters
//constructor
}
public class Customer extends User implements CustomerActions {

private String customerFirstName;

private String customerLastName;

private String customerAddress;

private String customerPaymentDetails;

private ArrayList<Booking> customerBookings;

//getters and setters

//constructor

        @Override

public void cancel() {

System.out.println("Customer can cancel booking");

}

@Override

public void book() {

System.out.println("Customer can make booking");

}

}
```

The code adheres to the Liskov Substitution Principle because instances of the Customer class can be used wherever instances of the User class are expected without affecting the program's correctness. The overridden methods in the Customer class don't violate the expectations of the superclass methods. The code follows the Liskov Substitution Principle by establishing a proper inheritance relationship between the Customer and User classes. Instances of the Customer class can be seamlessly substituted for instances of the User class, ensuring compatibility and preserving the expected behavior.

4. Interface Segregation

This principle states that a class should not be forced to implement interfaces it does not use. It suggests breaking large interfaces into smaller, specific ones so that clients are not required to implement methods they do not need.

Example:

```
public interface CustomerActions {
public void book();
public void cancel();
}

public interface ServiceProviderActions {
public void cancel();
}


public class ServiceProvider extends User implements ServiceProviderActions {
private String serviceProviderName;
private String serviceProviderAddress;
private String serviceProviderContact;
private ArrayList<BookingService> servicesProvided;
//getters and setters
//constructor
@Override
public void cancel() {
System.out.println("Service Provider can cancel booking");
}
}
```

The provided code exemplifies adherence to the Interface Segregation Principle (ISP) of SOLID design. In the `ServiceProviderActions` interface, a single method, `cancel()`, is declared, and the corresponding `ServiceProvider` class implements it with a specific implementation. Similarly, the `CustomerActions` interface declares two methods, `book()` and `cancel()`, both of which are implemented by the `Customer` class with specific functionalities. This design ensures that each class is responsible for and implements only the methods pertinent to its role, avoiding the introduction of unnecessary methods. By adhering to the ISP, the code promotes a modular and maintainable structure, facilitating future extensions or modifications without affecting unrelated components. The interfaces precisely capture the actions

associated with service providers and customers, contributing to a clear and focused design in line with SOLID principles.

5. Dependency Inversion Principle

This principle states that high-level modules should not depend on low-level modules; both should depend on abstractions. It promotes dependency on abstractions rather than concrete implementations, facilitating flexibility and ease of change in a system.

Example

```
public interface UserActions {
public default void register() {
System.out.println("User registered");
}
}
```

```
import com.solid.interfaces.UserActions;
//this user registration service can be changed to reset password or update user etc.
public class UserRegistrationService {
 private UserActions userActions;
 //dependency inversion
 public UserRegistrationService(UserActions userActions) {
 this.userActions = userActions;
 }

 public void registerUser() {
 userActions.register();
 }
}
```

In summary, the code adheres to the Dependency Inversion Principle by relying on abstractions (**UserActions** interface) and allowing the low-level module (**UserRegistrationService**) to depend on these abstractions rather than concrete details. This promotes flexibility, extensibility, and ease of maintenance.