Liskov Substitution Principle (LSP)

In the example below, the animal is a base class called Animal, where its move function prints a message indicating that it is moving. The move method is overridden by the Animal subtype's Bird class to output a message indicating that the bird is flying. The move method is overridden by the Penguin class, which is a subtype of Animal, to output a message indicating that the penguin is waddling.

Then, an AnimalTrainer class is created, and its trainAnimal method invokes the move method of an Animal object. It creates instances of the Bird and Penguin, then provide those instances to the trainAnimal function.

This violates the Liskov Substitution Principle since the Bird class cannot be replaced by the Penguin class without changing the program's correctness. A bird object in this scenario is able to fly, while a penguin object is not. However, because Penguin is an Animal subtype, the trainAnimal method accepts it as a valid argument.

The move method in the Penguin class outputs that the penguin is waddling, but the trainAnimal method anticipates it to fly, so when the code ran, it receives an inaccurate message for the penguin object. This demonstrates how the Liskov Substitution Principle can be broken if our class hierarchy isn't designed correctly and subtypes can't be substituted for their base types without causing the behavior to break.

}public class Animal {

public void move() {

System.***out***.println("The animal is moving.");

}

}

public class Bird extends Animal{

*@Override*

public void move() {

System.***out***.println("The bird is flying.");

}

}

public class Penguin extends Animal{

*@Override*

public void move() {

System.***out***.println("The penguin is waddling.");

}

}

public class AnimalTrainer {

public void trainAnimal(Animal animal) {

animal.move();

}

}

public class Main {

public static void main(String[] args) {

AnimalTrainer trainer = new AnimalTrainer();

Animal bird = new Bird();

Animal penguin = new Penguin();

trainer.trainAnimal(bird);

trainer.trainAnimal(penguin);

}

}

Dependency Inversion Principle

In order to implement the Dependency Inversion Principle I got rid of the dependencies that are hardcoded into the ElectricPowerSwitch class. To accomplish this, I first developed a new interface called Switchable that has two methods turnOn() and turnoff(). The LightBulb class was then changed to implement the Switchable interface. The ElectricPowerSwitch class was then updated to use the Switchable interface and all references to the LightBulb were eliminated.

To show how the ElectricPowerSwitch class is independent of specific implementations, I developed a new Fan class that adheres to the Switchable interface.

public class Control {

public static void main(String[] args) {

Switchable lightBulb = new LightBulb();

Switchable fan = new Fan();

ElectricPowerSwitch bulbSwitch = new ElectricPowerSwitch(lightBulb);

ElectricPowerSwitch fanSwitch = new ElectricPowerSwitch(fan);

bulbSwitch.press();

bulbSwitch.press();

fanSwitch.press();

fanSwitch.press();

}

}

public class ElectricPowerSwitch {

public Switchable switchable;

public boolean on;

public ElectricPowerSwitch(Switchable switchable) {

this.switchable = switchable;

this.on = false;

}

public boolean isOn() {

return this.on;

}

public void press() {

boolean checkOn = isOn();

if (checkOn) {

switchable.turnOff();

this.on = false;

} else {

switchable.turnOn();

this.on = true;

}

}

}

public class Fan implements Switchable{

public void turnOn() {

System.***out***.println("Fan: Fan turned on...");

}

public void turnOff() {

System.***out***.println("Fan: Fan turned off...");

}

}

public class LightBulb implements Switchable {

public void turnOn() {

System.***out***.println("LightBulb: Bulb turned on...");

}

public void turnOff() {

System.***out***.println("LightBulb: Bulb turned off...");

}

}

public interface Switchable {

void turnOn();

void turnOff();

}

First, a new interface called Switchable was created with two methods. Next the LightBulb class was modified to implement the Switchable interface. Then, the ElectricalPowerSwitch class was modified to use the Switchable interface and remove all references to LightBulb. The new Fan class implements Switchable to demonstrate how the ElectricalPowerSwitch class is not dependent on specific implementations. With these changes, the code has been made more extensible by allowing the ElectricPowerSwitch to work with any class that implements the Switchable interface.