LSP

Phone.java

```java
public class Phone {
    private String name;
    private String manufacturer;
    private int cost;

    public Phone(String name, String manufacturer, int cost) {
        this.name = name;
        this.manufacturer = manufacturer;
        this.cost = cost;
    }

    public void takePhoto() {
        System.out.println("Snaping a Selfie!");
    }
}
```

FlipPhone.java

```java
public class FlipPhone extends Phone {

    public FlipPhone(String name, String manufacturer, int cost) {
        super(name, manufacturer, cost);
    }

    // This method is not applicable to FlipPhone
    @Override
    public void takePhoto() {
        throw new UnsupportedOperationException();
    }

    public static void main(String[] args) {
        Phone phone = new Phone("iPhone", "Apple", 1000);
        Phone flipPhone = new FlipPhone("Not a iPhone", "Not Apple", 1);

        phone.takePhoto();
        flipPhone.takePhoto();
    }
}
```

This example violates LSP because the phone class has a takePhoto function, assuming every phone has a camera. But in reality there is a subclass for a flip phone (which for this example doesn't have a camera even though a lot of them do). If you swap any phone instance that calls takePhoto() with a flipPhone instance, the code will through an error resulting in code breaking behavior.

Control.java

```java
public class Control {
    public static void main(String[] args) {
        LightBulb lightBulb = new LightBulb();
        Fan fan = new Fan();
        ElectricPowerSwitch bulbSwitch = new ElectricPowerSwitch(lightBulb);
        ElectricPowerSwitch fanSwitch = new ElectricPowerSwitch(fan);

        bulbSwitch.press();
        bulbSwitch.press();
        fanSwitch.press();
        fanSwitch.press();
    }
}
```

Switchable.java

```java
public interface Switchable {
    public void turnOn();

    public void turnOff();
}
```

ElectricPowerSwitch.java

```java
public class ElectricPowerSwitch {
    public boolean on;
    public Switchable client;

    public ElectricPowerSwitch(Switchable client) {
        this.on = false;
        this.client = client;
    }

    public boolean isOn() {
        return this.on;
    }

    public void press() {
        boolean checkOn = isOn();
        if (checkOn) {
            client.turnOff();
            this.on = false;
        } else {
            client.turnOn();
            this.on = true;
        }
    }
}
```

LightBulb.java

```java
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...");
    }
}
```

Fan.java

```java
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...");
    }

}
```

Changes: to make the provided code satisfy DIP, I created a switchable interface with a turnOn and turnOff method. Both the LightBulb and new Fan class implement this switchable class and define the functionality of these two classes. The electricPowerSwitch class now takes a switchable argument instead of a lightbulb to allow for any switchable to be used; the member variables and functions were modified to support the new refactoring. Only changes to control were to create new fan and test to make sure the classes function as expected.