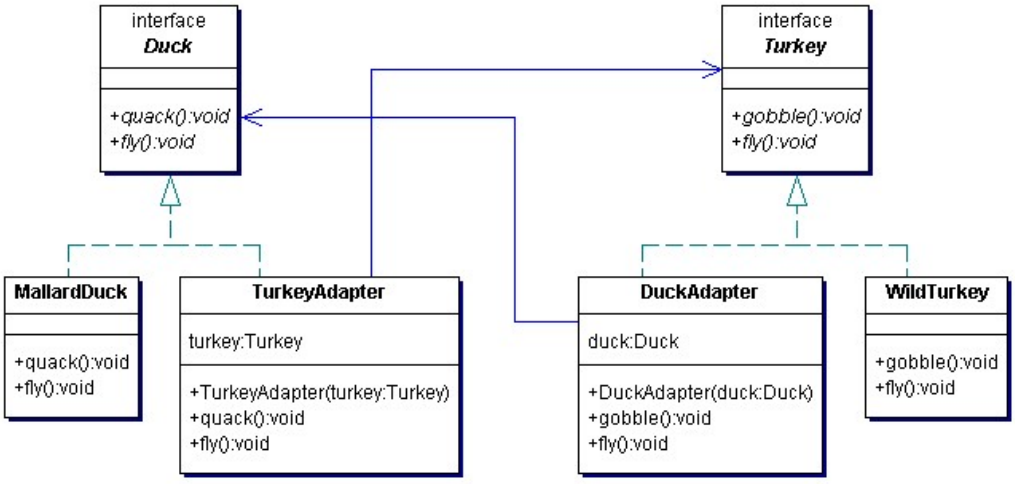
The Adapter Pattern

Lab 1:

Turkey có interface khác với **Duck**. Chúng ta muốn dùng **Turkey** như **Duck** và ngược lại. Chuyển đổi interface của **Duck** và **Turkey** để chúng có thể được dùng qua lại như nhau. Dùng Object Adapter hai chiều như sau:



Cài đặt các lớp cho thiết kế và bổ sung các phần còn thiếu (// TODO)

```
public interface Duck {
   public void quack();
   public void fly();
}
```

```
public class MallardDuck implements Duck {
   public void quack() {
      System.out.println("Quack");
   }

   public void fly() {
      System.out.println("I'm flying");
   }
}
```

```
public interface Turkey {
   public void gobble();
   public void fly();
}
```

```
public class WildTurkey implements Turkey {
   public void gobble() {
      System.out.println("Gobble gobble");
   }

   public void fly() {
      System.out.println("I'm flying a short distance");
   }
}
```

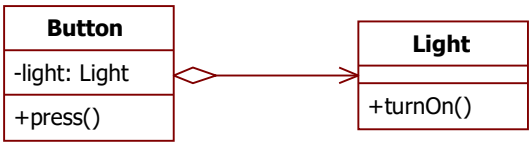
```
public class TurkeyAdapter implements Duck {
   private Turkey turkey;

   public TurkeyAdapter(Turkey turkey) {
      this.turkey = turkey;
   }

   public void quack() {
      // TODO
   }

   public void fly() {
      for (int i = 0; i < 5; i++) {
         // TODO
      }
   }
}
```

```
public class DuckAdapter implements Turkey {
   private Duck duck;
   private Random rand;

   public DuckAdapter(Duck duck) {
      this.duck = duck;
      rand = new Random();
   }

   public void gobble() {
      // TODO
   }

   public void fly() {
      if (rand.nextInt(5) == 0) {
         // TODO
      }
   }
}
```

```
public class DuckTestDrive {
   public static void main(String[] args) {
      MallardDuck duck = new MallardDuck();
      WildTurkey turkey = new WildTurkey();
      Duck turkeyAdapter = new TurkeyAdapter(turkey);

      System.out.println("The Turkey says...");
      turkey.gobble();
      turkey.fly();

      System.out.println("\nThe Duck says...");
      testDuck(duck);

      System.out.println("\nThe TurkeyAdapter says...");
      testDuck(turkeyAdapter);
   }

   static void testDuck(Duck duck) {
      duck.quack();
      duck.fly();
   }
}
```

```
public class TurkeyTestDrive {
   public static void main(String[] args) {
      MallardDuck duck = new MallardDuck();
      Turkey duckAdapter = new DuckAdapter(duck);

      for (int i = 0; i < 10; i++) {
         System.out.println("The DuckAdapter says...");
         duckAdapter.gobble();
         duckAdapter.fly();
      }
   }
}
```

Lab 2: Consider a **`Button`** class that uses a **`Light`** class as follows:

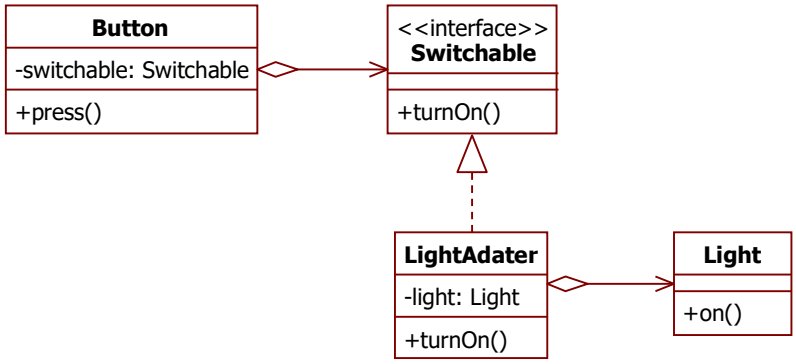



```
public class Button {
   private Light light;

   public Button(Light light) {
      this.light = light;
   }

   public void press() {
      light.turnOn();
   }
}
```

How can be break the dependency between **`Button`** and **`Light`**, and thus conform to the SOLID principles of OOD, if we can't modify the **`Light`** class?

1. The Adapter Pattern

As shown in the diagram below, this pattern adds an *extra object* called **`LightAdapter`**, implements the **`Switchable`** interface and delegates messages received by that interface to the associated **`Light`** object.



The code that implements the **`Adapter`** is trivial.

```
public class Button {
   private Switchable switchable;

   public Button(Switchable switchable) {
      this.switchable = switchable;
   }

   public void press() {
      // TO DO
   }
}
```

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

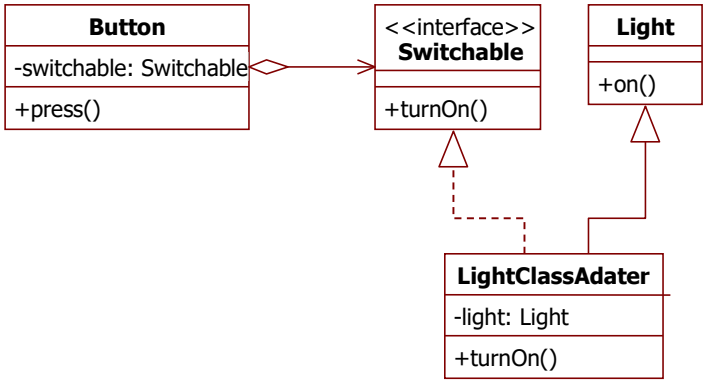
```
public class LightAdapter implements Switchable {
   private Light light;

   public LightAdapter(Light light) {
      this.light = light;
   }

   public void turnOn() {
      // TO DO
   }
}
```

```
public class Light {
   private boolean on;
   public void on() {
      System.out.println("Light turns on");
      on = true;
   }

   public void off() {
      System.out.println("Light turns off");
      on = false;
   }

   public boolean isOn() {
      return on;
   }
}
```

This solves the problem nicely. The **`Button`** class no longer knows about the **`Light`**, and the **`Light`** has not been modified.

```
public class LightAdapterTestDrive extends TestCase {
   public void testButtonControlsLight() throws Exception {
      Light l = new Light();
      Switchable la = new LightAdapter(l);
      Button b = new Button(la);
      b.press();
      assertTrue(l.isOn());
   }
}
```

2. The Class Adapter form




```
public class LightClassAdapter extends Light implements Switchable {
   public void turnOn() {
      on();
   }
}
```

Isn't it wonderful that this class has no body? And yet, it completely fulfills its role as an **`Adapter`**. The unit test below shows how easy this form of the **`Adapter`** is to use.

```
public class LightAdapterTestDrive extends TestCase {
   // ...

   public void testButtonControlsLightThroughClassAdapter() {
      Switchable lca = new LightClassAdapter();
      Button b = new Button(lca);
      b.press();
      assertTrue(lca.isOn());
   }
}
```

3. Adapter Implemented with Anonymous Inner Class

The anonymous inner class feature of Java can be a wonderful way to create *object*-form Adapters. Consider the following code:

```
public class AdapterAnonymousTest extends TestCase {
   private Light l = new Light();

   public void testAnonymousInnerClassAdapter() throws Exception {
      Switchable s = new Switchable() {
         public void turnOn() {
            l.on();
         }
      };
      Button b = new Button(s);
      b.press();
      assertTrue(l.isOn());
   }
}
```

4. Adapt a Sender's Protocol

Another use of the **`Adapter`** pattern is to *adapt* the protocol of a sender to a receiver.
For example, let's say that we have a class that looks like this:

```
public class ThreeWayLight {
   private int brightness = 0;

   public void lo() {
      brightness = 1;
   }

   public void medium() {
      brightness = 2;
   }

   public void high() {
      brightness = 3;
   }

   public void off() {
      brightness = 0;
   }

   public int getBrightness() {
      return brightness;
   }
}
```

The **`Button`** was never designed to control a three-way light. Moreover, it looks as though the **`ThreeWayLight`** class was not designed to take input from a **`Button`**. How can we adapt the **`Button`** class to the **`ThreeWayLight`** ? Consider the following unit test:

```
public class LightAdapterTestDrive extends TestCase {
   // ...
   public void testThreeWayLight() throws Exception {
      ThreeWayLight twl= new ThreeWayLight();
      Switchable twa = new ThreeWayAdapter(twl);
      Button b = new Button(twa);
      assertEquals(0, twl.getBrightness());
      b.press();
      assertEquals(1, twl.getBrightness());
      b.press();
      assertEquals(2, twl.getBrightness());
      b.press();
      assertEquals(3, twl.getBrightness());
      b.press();
      assertEquals(0, twl.getBrightness());
   }
}
```

Clearly our intent is that the **`ThreeWayAdapter`** should ratchet the **`ThreeWayLight`** through its states every time the **`Button`** is pressed. We can easily implement this adapter as follows:

```
public class ThreeWayAdapter implements Switchable {
   private ThreeWayLight twl;

   public ThreeWayAdapter(ThreeWayLight twl) {
      this.twl = twl;
   }

   public void turnOn() {
      switch (twl.getBrightness()) {
      case 0:
         twl.lo();
         break;
      case 1:
         twl.medium();
         break;
      case 2:
         twl.high();
         break;
      case 3:
         twl.off();
         break;
      }
   }
}
```