Task1

(i) 单例模式是为了确保一个类只有一个实例对象

(ii) 创建一个私有的构造函数，使得不能再其他地方实例化这个类。

(iii) 首先初始化一个实例对象，和统计次数。将构造器私有化，使得其他类里面不能调用这个实例方法。通过 public static getInstance 方法来得到 ExampleSingleton 的实例。在方法里面进行判断实例对象是否为空，如果为空，就调用构造函数，如果不为空，就直接返回。这样可以确保只有一个实例对象。

ExampleSingleton 的实现如下

```
public class ExampleSingleton {
    private static ExampleSingleton singletonInstance;
    private static int accessCount = 0;

    private ExampleSingleton(){
        System.out.println( "I, the ExampleSingleton, am being
created");
    }

    public static ExampleSingleton getInstance(){
        accessCount++;
        if(singletonInstance == null) {
            singletonInstance = new ExampleSingleton();
        }
        System.out.println("The sole instance of
ExampleSingleton is being retrieved");
        return singletonInstance;
    }

    public int accessCount(){
        return accessCount;
    }
}
```

Task2

(i) 适配器相当于转接口。将一个类型转换成另外一个特定的类型，使得不同类型的两个类能够一起工作。

(ii) 实现一个 FahrenheitLogger.java. 如下：

```
public class FahrenheitLogger {
    private double theTemperature;

    public double getTheTemperature() {
        return theTemperature;
    }
```

```
    public void setTheTemperature(double theTemperature) {
        this.theTemperature = theTemperature;
    }
}
```

(iii) CelsiusToFahrenheitLoggerAdapter 类的实现如下，首先继承 CelsiusLogger，然后引用 FahrenheitLogger，在 setTemperature 方法里面转换成 FahrenheitLogger 对应的值。代码如下：

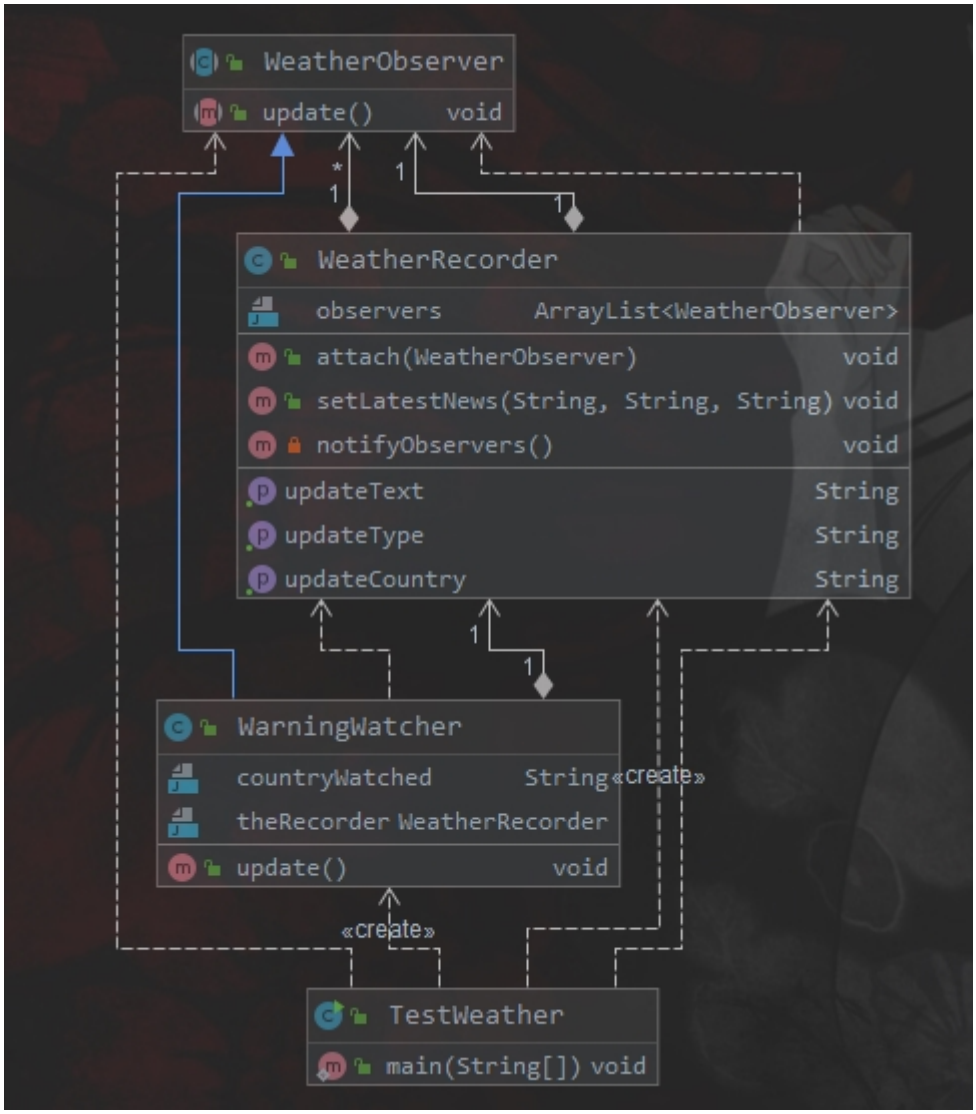
```
public class CelsiusToFahrenheitLoggerAdapter extends
CelsiusLogger {
    FahrenheitLogger fahrenheitLogger = new FahrenheitLogger();

    @Override
    public void setTemperature(double aCelsiusTemp) {
        fahrenheitLogger.setTheTemperature(aCelsiusTemp * 9/5 +
32);
    }

    @Override
    public double getTemperature() {
        double fahreneitTemperature =
fahrenheitLogger.getTheTemperature();
        return ((fahreneitTemperature - 32) * 5/9);
    }
}
```

(Iv) CelsiusToFahrenheitLoggerAdapter 类首先继承 CelsiusLogger，并且引用 FahrenheitLogger，重写 CelsiusLogger 的 setTemperature 方法和 getTemperature 方法时，调用目标类的 getter/setter 方法，并按特定的规则转换成目标列 FahrenheitLogger 对应的值。

Task3

(i) 观察者设计模式中，对象维护了一个依赖列表，以便可以使用观察者定义的任何方法通知所有观察者所发生的变化.

(ii)



(iii) 修改后的代码如下 The WeatherRecorder attach() method

```
public void attach(WeatherObserver o) {
// Complete this method so that it adds the observer to
the list of observers
  observers.add(o);
}
```

the WeatherRecorder notifyObservers() method

```
  private void notifyObservers() {
// Complete this method to go through each observer in
turn,
// sending it a message to notify that an update has
occurred
    for(WeatherObserver o : observers) {
      o.update();
    }
  }
```

the WarningWatcher update() method

```
public void update() {
// Modify this so that it only prints out the update text
if the update is a "Warning" for the country being
watched
  if(this.theRecorder.getUpdateType().equals("Warning")
&&
countryWatched.equals(this.theRecorder.getUpdateCountry()
))
  System.out.println("The WarningWatcher watching for
Warnings for " +
                     countryWatched +
                     "\nhas noticed a new warning:\n\"" +
                     theRecorder.getUpdateText() +
"\"\n");
}
```

(iv) WarningWatcher 继承了 WeatherObserver.实现了 update 方法，并且定义了自己的城市，引用了 WeatherRecorder.这样，在 update 的方法中就可以判断当前观察者与当前记录的城市是否一致。从而判断是否报警。在 WetherRecorder 中，保存了每一个观察者，当调用 setLatestNews 方法时，更新完当前记录后，就通知每一个观察者天气更新了。观察者在自己的 update 的方法里面做出反应。