참고 자료

- https://refactoring.guru/design-patterns/decorator

DecoratorPattern이란?

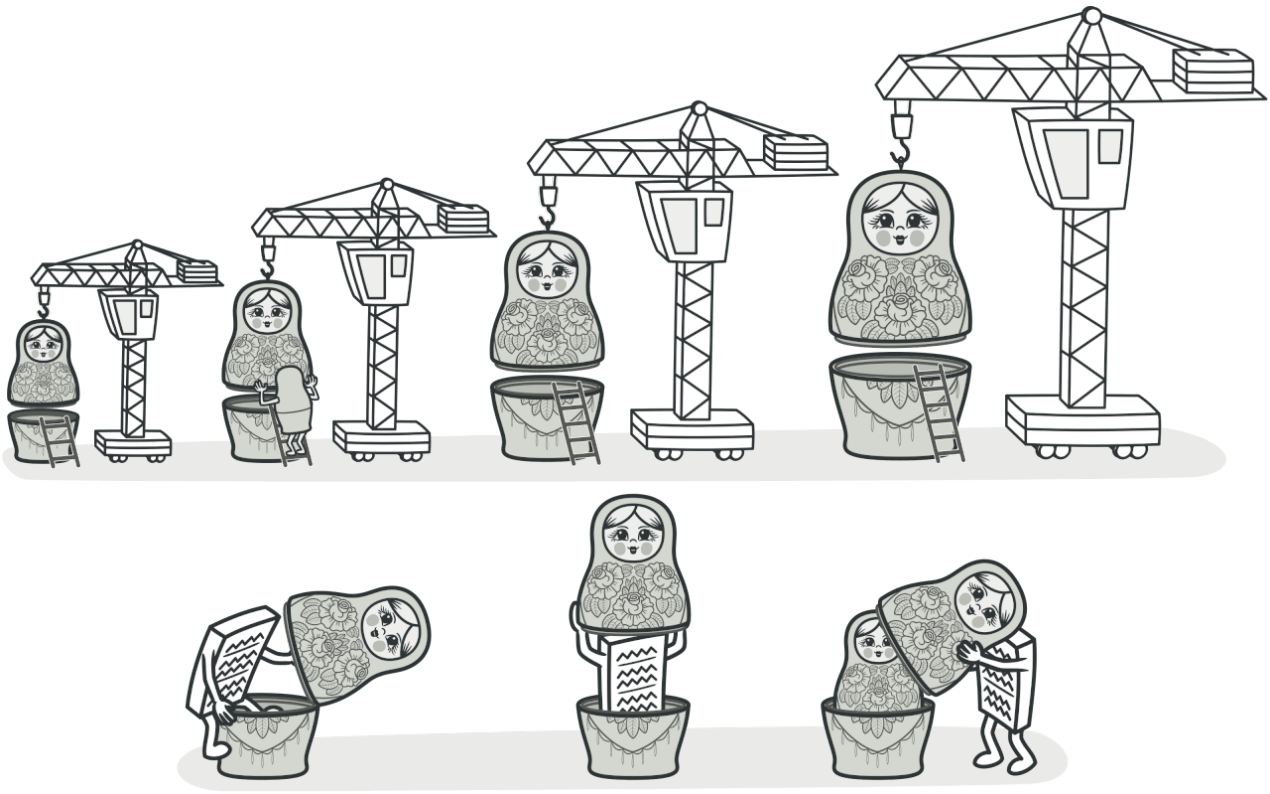
- structural design pattern
- 기존 객체에 새로운 기능을 추가하고 싶을 때, 새로운 기능을 가진 wrapper 객체 내부에 기존 객체를 넣음으로써 기능을 추가하는 패턴입니다.

  wiki

  decorator pattern allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class

  guru

  decorator pattern lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors

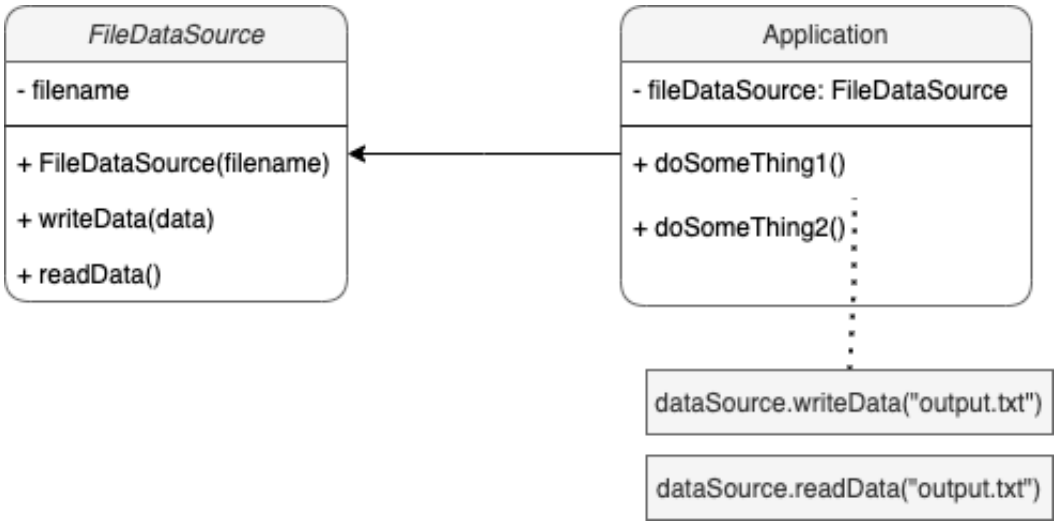


상황

이 상황은 기능이 확장되면서 발생하는 문제를 다룹니다.

초기

- 파일을 읽고 쓰는 기능을 가진 라이브러리를 생각해봅시다.
- 이 라이브러리는 파일을 입력받아 해당 파일을 읽고 쓰게 해줍니다.



구조

1. FileDataSource

   읽고 쓰는 기능을 가진 라이브러리입니다. 파일 이름을 받아 읽거나 쓰는 작업을 수행합니다.

2. Application

   FileDataSource를 가지고 있고, FileDataSource를 통해 비즈니스 로직에서 파일을 읽거나 쓰는 작업을 수행합니다.

   doSomeThing은 유저를 관리하는 어플리케이션이라면 개인정보를 읽거나 저장하는 메서드명(예를 들면 saveUserInfo 정도)이 되겠지요.

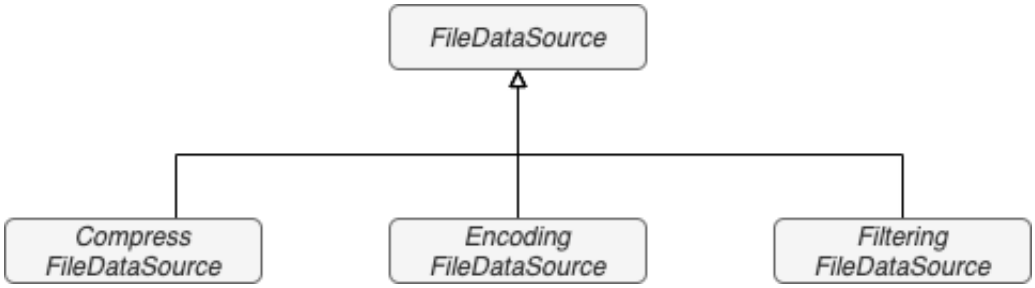
확장 1

요구사항

압축하거나, 인코딩하거나, 필터링을 통해 파일을 읽고 쓰고 싶다고 피드백을 받았습니다.

대응

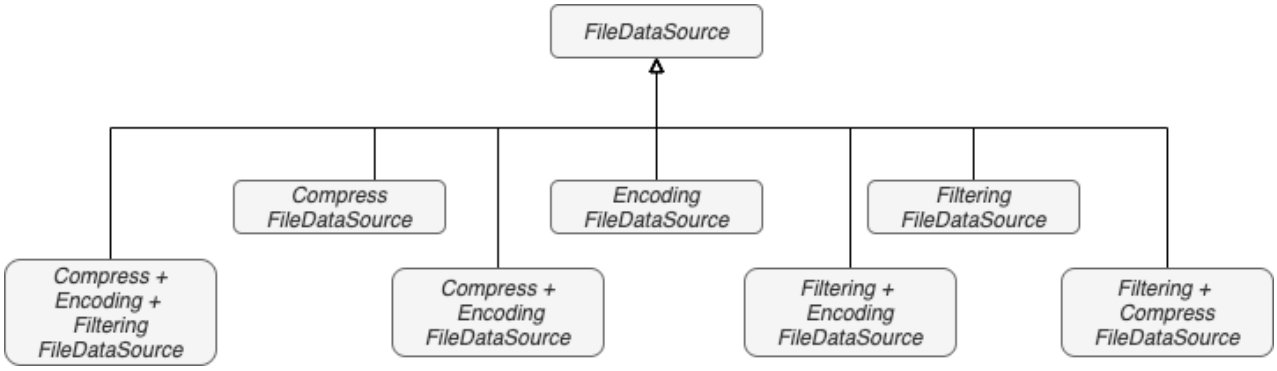
FileDataSource를 상속받아 readData, writeData를 구현함으로써 해결하였습니다.



확장 2

요구사항

또 다시 피드백을 받았습니다. 읽고 쓰는 여러 방식 중 하나만 보낼 수 있는 것이 아니라, 동시에 여러 방식을 적용해 달라고 피드백을 받았습니다.



대응

하나의 클래스가 여러 알림 기능이 결합된 메서드를 만들도록 합니다.

문제점

1. 모든 경우에 대응하기 위한 클래스의 수가 많아집니다.

이항 정리에 따라 2^n-1개 만큼의 클래스를 만들어야만 합니다 (nC1 + nC2 + nC3 + ... + nCn)

> 한편으로, 이렇게 조합으로 인해 복잡도가 급격히 증가하는 문제를 combinatorial explosion라고 부릅니다.

2. 런타임 중에 동작을 바꿀 수 없습니다.

상속은 컴파일 타임에 정해지기 때문이지요. 다시 말해서 재컴파일해야 합니다.

3. 대다수 프로그램이 다중 상속을 지원하지 않습니다.

따라서 여러 기능을 상속받을 수 없습니다.

해결책

상속보다는 Aggregation 또는 Composition으로 기능을 추가하라!

1. 하나의 객체(Whole)는 다른 객체(Part)의 reference를 가지고 있습니다.

   이전에 설명드린 객체의 관계에서, Whole은 part를 감싸고 있는 형태이므로 **Wrapper**라고도 불린다 설명드렸습니다.
2. 그리고 하나의 객체(Whole)는 다른 객체(Part)에게 작업을 위임할 수 있습니다.

이처럼 구성하면 재컴파일 하지 않고도 기능을 정의할 수 있으며, 여러 클래스들을 기능의 위임을 통해 사용할 수 있습니다.

Aggregation과 Composition은 디자인 패턴의 핵심요소이며, 이번에 설명드릴 **Decorator**에서 또한 핵심요소입니다.

구조

구현시 신경써야할 사항은 다음과 같습니다.

1. 이미 기존에 무언가가 있고, 그것을 꾸미는 형태이다.

   > 정의: 기존 객체에 새로운 기능을 추가하고 싶을 때, 새로운 기능을 가진 wrapper 객체 내부에 기존 객체를 넣음으로써 기능을 추가하는 패턴입니다.
2. FileDataSource가 어떻게 꾸며졌든, 동일한 타입이여야 합니다.

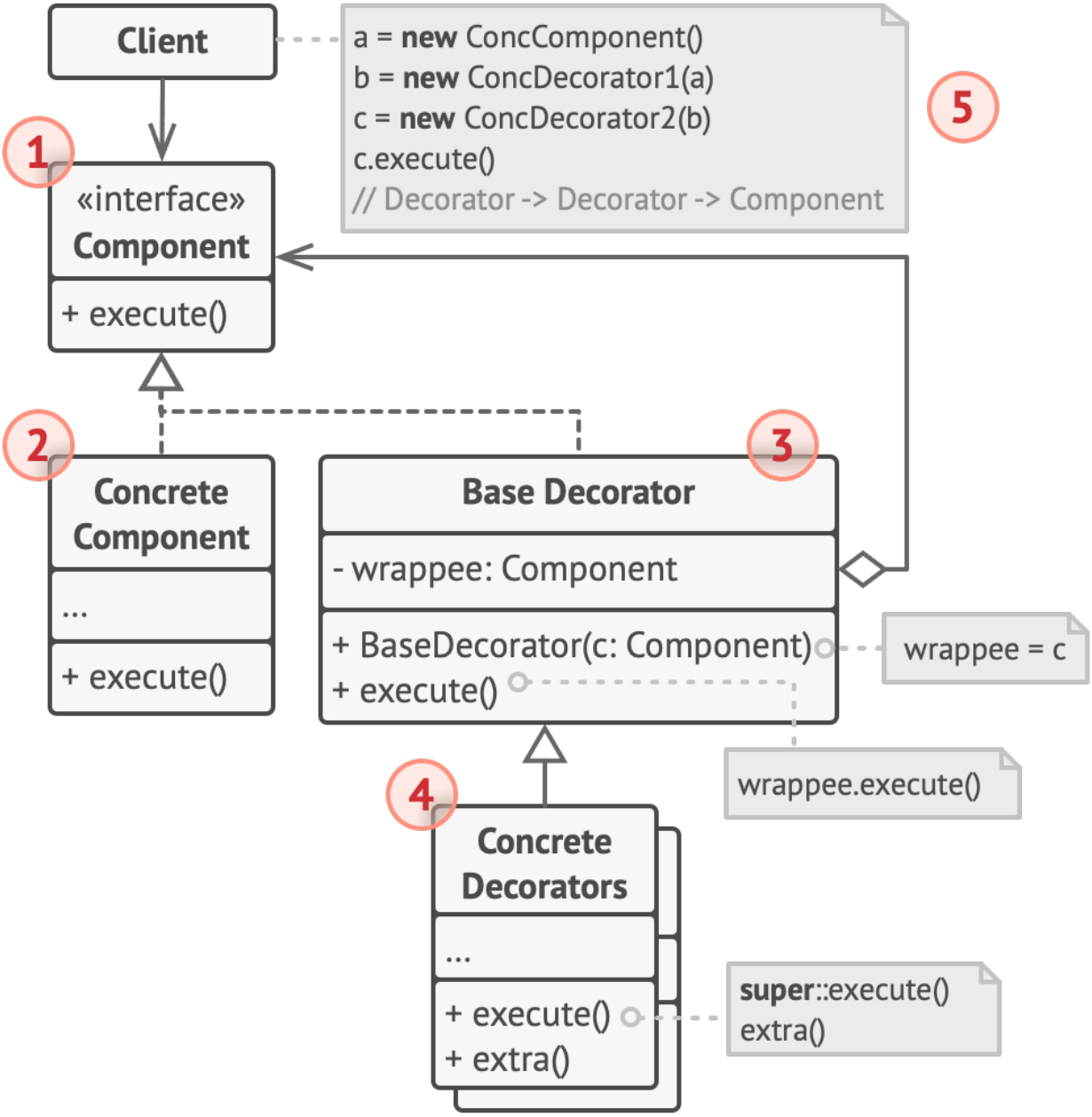
   클라이언트에서 FileDataSource의 의존성을 줄이기 위해서 동일한 타입이여야하죠. 이렇게 안하면 모든 케이스에 대해 조건문으로 구분해야겠지요?
3. 메서드를 위임받아 실행하므로 wrapee와 wrapper는 동일한 타입이여야 합니다.

   위임받아 실행하는데 다른 타입이 되면 wrapper type일땐 어떻게 실행하고 wrappee 타입일 땐 이렇게 실행하라라고 분기처리해야겠지요.

   한편으로, wrapee와 wrapper에 대한 연산으로 작업이 수행된다고 볼 수도 있을 것 같습니다.

   2 x 4 x 6 => (2 x 4) x 6 => 8 x 6 => 24

   2 x 4 x 6 => (2 x 4) x 6 => 꽁치 x 6 => ?



1. Component

- wrapper와 wrapee의 공통의 인터페이스를 정의합니다.

2.Concrete Component

- 데코레이터에 의해 감싸여질 기본 기능입니다.

데코레이터에 의해 기능이 변경될 수 있습니다.

3.Base Decorator

- wrapped object를 가질 수 있는 필드를 선언합니다.
- 이 필드는 반드시 Component(interface)타입을 가지도록 합니다.

  이와 같이 함으로써 concrete Component와 decorator를 wrapped object로 가질 수 있게 됩니다.
- 이 Base Decorator는 모든 연산을 wrapped object에게 위임합니다.

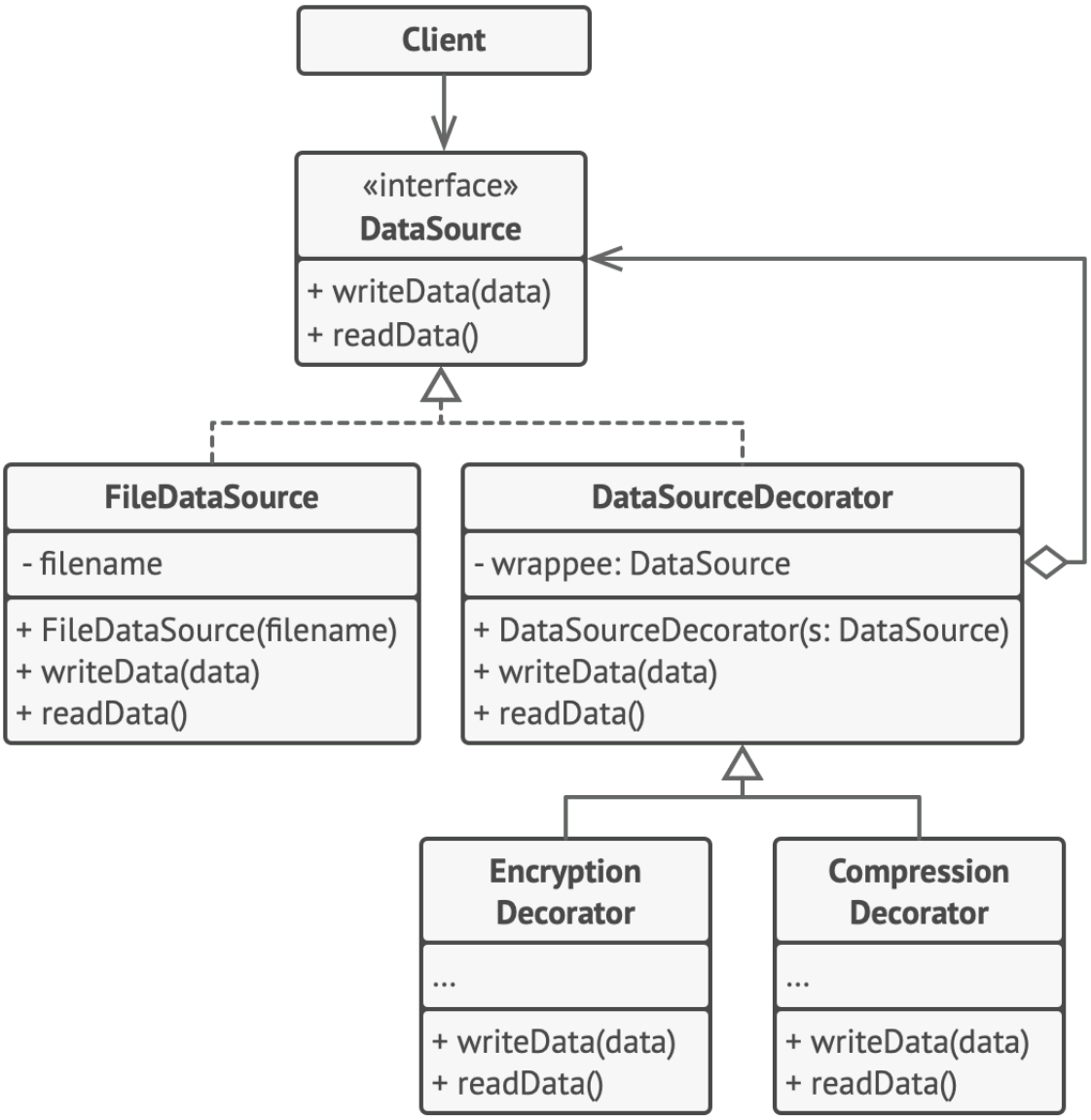
4.Concrete Decorators

- component에 추가될 기능을 정의합니다.
- base decorator의 기능을 오버라이딩하며, 이 base decorator 메서드가 실행되기 전 또는 후에 자신의 기능을 실행합니다.

5.Client

- component interface로 여러 개로 감싸여진 component로 작업을 합니다.

코드 구현



#####

decorators/DataSource.java:

```
package refactoring_guru.decorator.example.decorators;

public interface DataSource {
    void writeData(String data);

    String readData();
}
```

decorators/FileDataSource.java: Simple data reader-writer

```
package refactoring_guru.decorator.example.decorators;

import java.io.*;

public class FileDataSource implements DataSource {
    private String name;

    public FileDataSource(String name) {
        this.name = name;
    }

    @Override
    public void writeData(String data) {
        File file = new File(name);
        try (OutputStream fos = new FileOutputStream(file)) {
            fos.write(data.getBytes(), 0, data.length());
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }

    @Override
    public String readData() {
        char[] buffer = null;
        File file = new File(name);
        try (FileReader reader = new FileReader(file)) {
            buffer = new char[(int) file.length()];
            reader.read(buffer);
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
        return new String(buffer);
    }
}
```

decorators/DataSourceDecorator.java: Abstract base decorator

```
package refactoring_guru.decorator.example.decorators;

public class DataSourceDecorator implements DataSource {
    private DataSource wrappee;

    DataSourceDecorator(DataSource source) {
```

```
        this.wrappee = source;
    }

    @Override
    public void writeData(String data) {
        wrappee.writeData(data);
    }

    @Override
    public String readData() {
        return wrappee.readData();
    }
}
```

decorators/EncryptionDecorator.java: Encryption decorator

```
package refactoring_guru.decorator.example.decorators;

import java.util.Base64;

public class EncryptionDecorator extends DataSourceDecorator {

    public EncryptionDecorator(DataSource source) {
        super(source);
    }

    @Override
    public void writeData(String data) {
        super.writeData(encode(data));
    }

    @Override
    public String readData() {
        return decode(super.readData());
    }

    private String encode(String data) {
        byte[] result = data.getBytes();
        for (int i = 0; i < result.length; i++) {
            result[i] += (byte) 1;
        }
        return Base64.getEncoder().encodeToString(result);
    }

    private String decode(String data) {
        byte[] result = Base64.getDecoder().decode(data);
        for (int i = 0; i < result.length; i++) {
```

```
            result[i] -= (byte) 1;
        }
        return new String(result);
    }
}
```

decorators/CompressionDecorator.java: Compression decorator

```
package refactoring_guru.decorator.example.decorators;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.Base64;
import java.util.zip.Deflater;
import java.util.zip.DeflaterOutputStream;
import java.util.zip.InflaterInputStream;

public class CompressionDecorator extends DataSourceDecorator {
    private int compLevel = 6;

    public CompressionDecorator(DataSource source) {
        super(source);
    }

    public int getCompressionLevel() {
        return compLevel;
    }

    public void setCompressionLevel(int value) {
        compLevel = value;
    }

    @Override
    public void writeData(String data) {
        super.writeData(compress(data));
    }

    @Override
    public String readData() {
        return decompress(super.readData());
    }

    private String compress(String stringData) {
```

```
        byte[] data = stringData.getBytes();
        try {
            ByteArrayOutputStream bout = new ByteArrayOutputStream(512);
            DeflaterOutputStream dos = new DeflaterOutputStream(bout, new
Deflater(compLevel));
            dos.write(data);
            dos.close();
            bout.close();
            return Base64.getEncoder().encodeToString(bout.toByteArray());
        } catch (IOException ex) {
            return null;
        }
    }

    private String decompress(String stringData) {
        byte[] data = Base64.getDecoder().decode(stringData);
        try {
            InputStream in = new ByteArrayInputStream(data);
            InflaterInputStream iin = new InflaterInputStream(in);
            ByteArrayOutputStream bout = new ByteArrayOutputStream(512);
            int b;
            while ((b = iin.read()) != -1) {
                bout.write(b);
            }
            in.close();
            iin.close();
            bout.close();
            return new String(bout.toByteArray());
        } catch (IOException ex) {
            return null;
        }
    }
}
```

Demo.java: Client code

```
package refactoring_guru.decorator.example;

import refactoring_guru.decorator.example.decorators.*;

public class Demo {
    public static void main(String[] args) {
        String salaryRecords = "Name,Salary\nJohn Smith,100000\nSteven
Jobs,912000";
        DataSourceDecorator encoded = new CompressionDecorator(
                                        new EncryptionDecorator(
```

```
                                                new
FileDataSource("out/OutputDemo.txt")));
        encoded.writeData(salaryRecords);
        DataSource plain = new FileDataSource("out/OutputDemo.txt");

        System.out.println("- Input ----------------");
        System.out.println(salaryRecords);
        System.out.println("- Encoded --------------");
        System.out.println(plain.readData());
        System.out.println("- Decoded --------------");
        System.out.println(encoded.readData());
    }
}
```

Output.txt

```
- Input ----------------
Name,Salary
John Smith,100000
Steven Jobs,912000
- Encoded --------------
Zkt7e1Q5eU8yUm1Qe0ZsdHJ2VXp6dDBKVnhrUHtUe0sxRUYxQkJIdjVLTVZ0dVI5Q2IwOXFISmVUMU5
rcENCQmdxRlByaD4+
- Decoded --------------
Name,Salary
John Smith,100000
Steven Jobs,912000
```