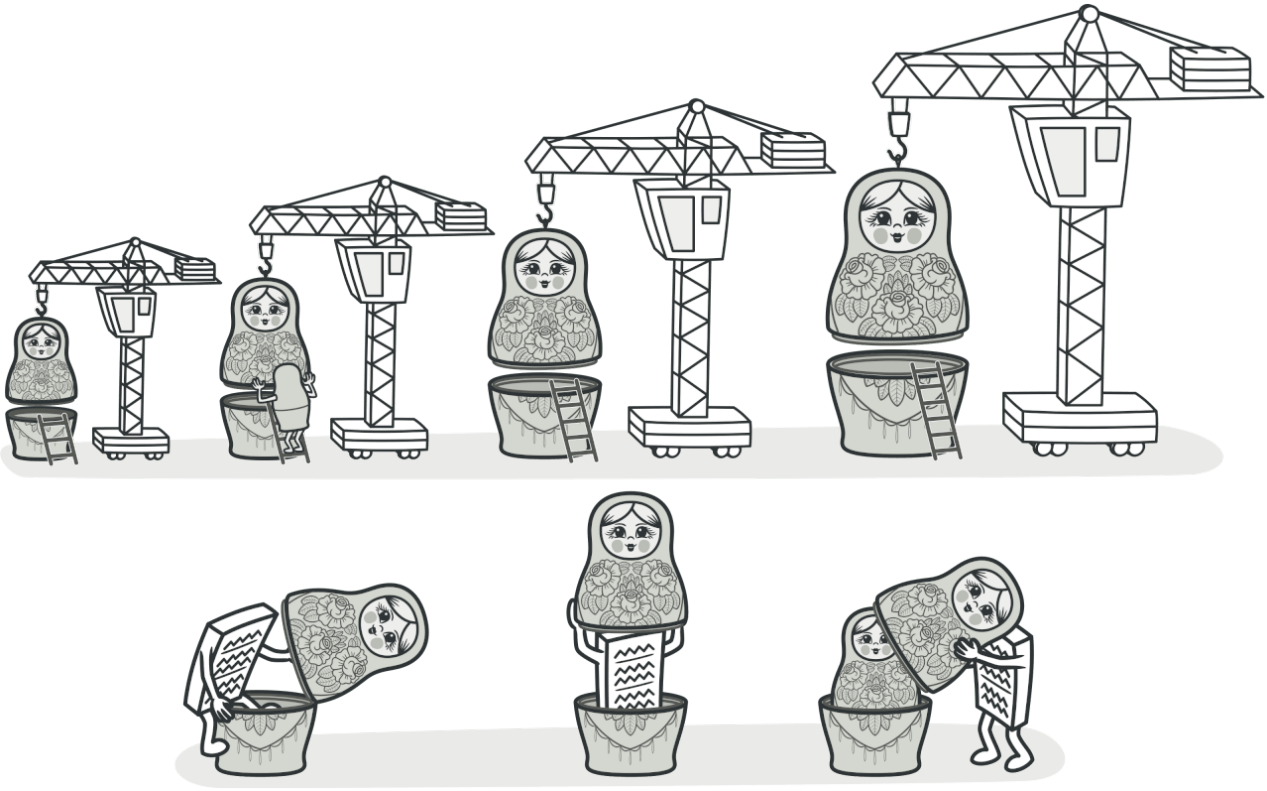
참고 자료

- 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.

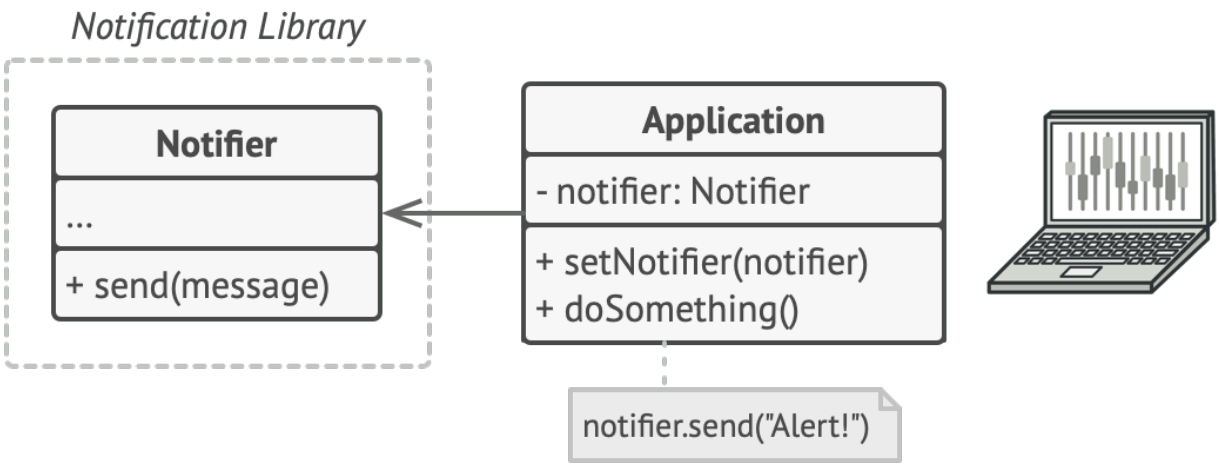


상황

이 상황은 기능이 확장되면서 발생하는 문제를 다룹니다.

초기

- 알림(notification)기능을 가진 라이브러리를 생각해봅시다.
- 이 라이브러리는 메세지를 입력받아 해당 메세지를 메일로 보내줍니다.



구조

1. Notifier

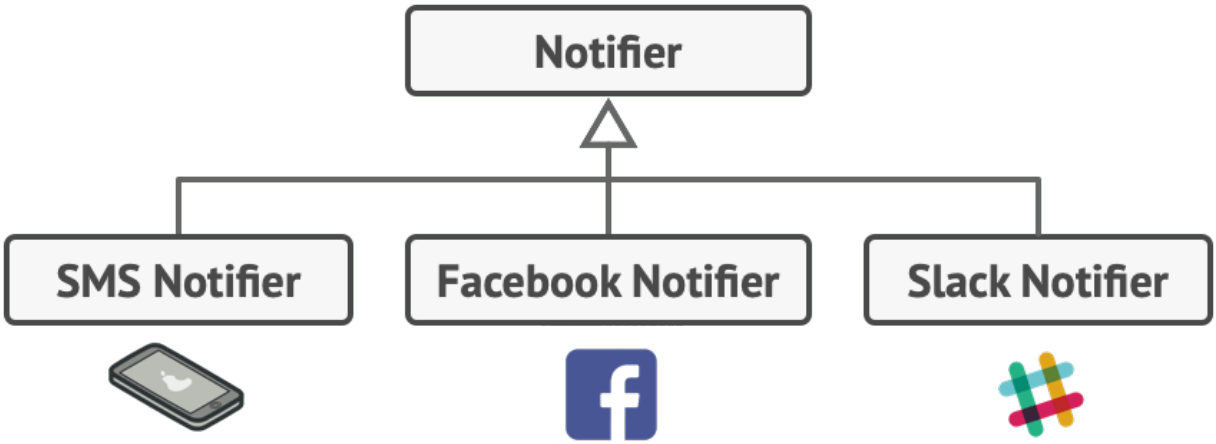
   알림기능을 가진 라이브러리입니다. 메세지를 받아 send메서드를 통해 이메일을 보냅니다

2. Application

   Notifier을 가지고 있고, 그 Notifier를 통해 메세지를 보내는 메서드를 가지고 있습니다.

확장 1

- 사용자의 피드백을 받아 e-mail로만 보내는 것이 아닌, SMS문자나, Facebook, Slack을 통해 알림을 전송해주는 기능을 만들고 싶습니다.
- 그래서 이 문제는 SMS, Facebook, Slack등이 Notifier를 상속받아 `send()`를 구현함으로써 해결하였습니다.



확장 2

- 또 다시 사용자의 피드백을 받았습니다. 여러 매체 중 하나만 보낼 수 있는 것이 아니라, **여러 매체를 동시**에 보낼 수 있도록 해달라 요청받았습니다.

문제되는 방법 1

하나의 클래스가 여러 알림 기능이 결합된 메서드를 만들도록 합니다. (Application이 하나의 Notifier를 받아서 send기능을 사용합니다.)

만약 이렇게 되면 결국 모든 경우에 대응하기 위해 2^n-1개 만큼의 클래스를 만들어야만 합니다. (이항정리)

문제되는 방법 2

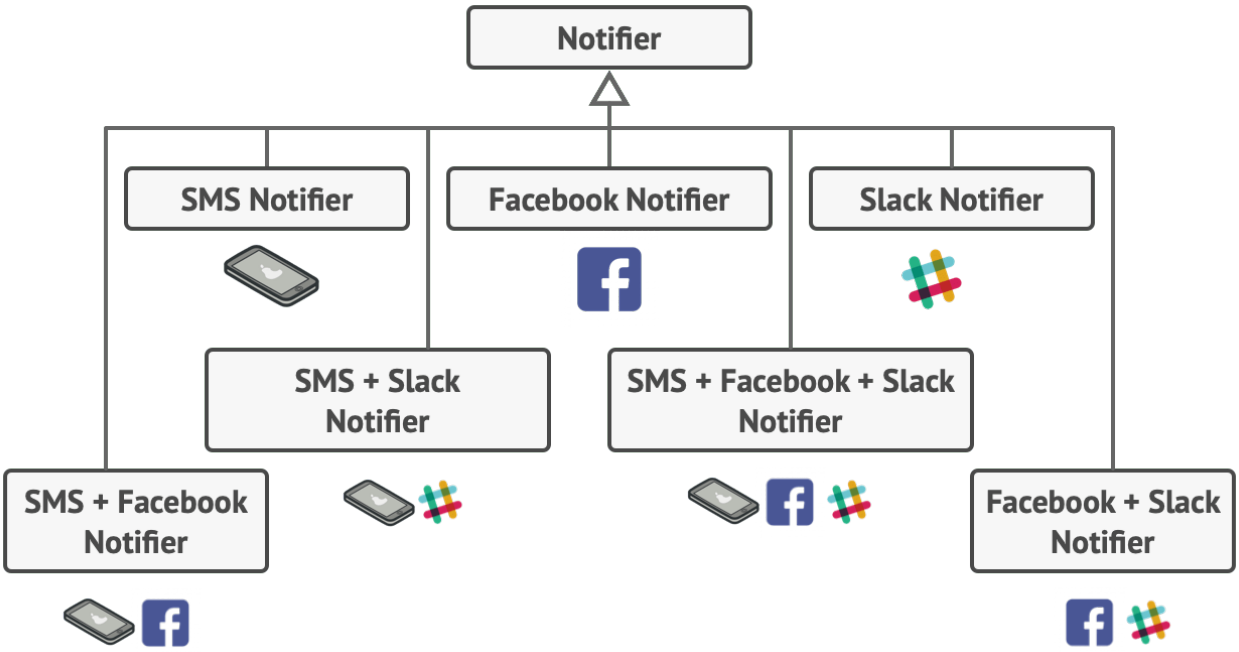
기능을 확장시키는 것이므로 단순히 상속하면 되지 않을까? 라는 생각을 가질 수 있습니다. 하지만 이것은 2가지 문제점을 가집니다.

1. 런타임 중에 동작을 바꿀 수 없습니다.

   상속은 컴파일 타임에 정해지기 때문이지요.

2. 대다수 프로그램이 다중 상속을 지원하지 않습니다.

   따라서 여러 기능을 상속받을 수 없습니다.



해결책

상속보다는 Aggregation 또는 Composition을 이용하라

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

   이전에 설명드린 객체의 관계에서, Whole은 part를 감싸고 있는 형태이므로 **Wrapper**라고도 불린다 설명드렸습니다.
2. 그리고 하나의 객체는 다른 객체들에게 작업을 위임할 수 있습니다.

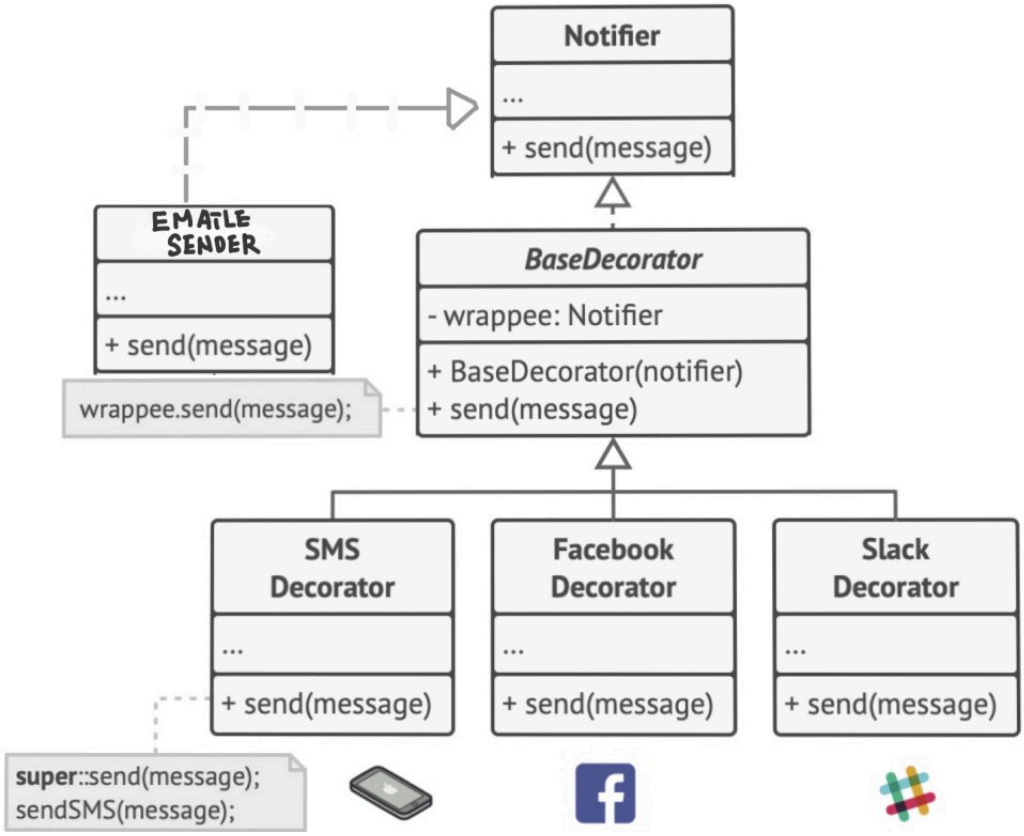
이처럼 구성하면 컨테이너의 행동을 런타임시에 결정지을 수 있으며, 여러 클래스들의 기능을 위임을 통해 사용할 수 있습니다.

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

그래서, Decorator 패턴를 이용하라! (또 다른 표현 Wrapper)

위 문제에서 Decorator를 적용하면 다음과 같은 구조가 됩니다.

(*guru에서는 EmailSender라는 클래스가 없고 Notifier또한 인터페이스가 아닌 클래스로 구성되어있습니다. 아마도 OCP를 유지한 채 확장시켜나가는 모습을 보여주기 위함이겠지요. 하지만 데코레이터 패턴의 구조 일관성을 위해 아래처럼 수정하였습니다.*)



Notifier

메세지를 보내는 기본적인 기능을 Notifier로 인터페이스를 만듭니다.

E-MAIL Sender

리프노드와 같은 성격입니다. 이메일을 보내는 클래스를 만듭니다.

그리고 다른 기능을 가진 클래스들은 Decorator로 변경시킵니다.

BaseDecorator

데코레이터들이 만들어질 기본 클래스입니다.

Notifier타입의 part을 가질 수 있는 field가 있으며, 동일하게 메시지를 보내는 기능을 지닙니다.

Facebook, Slack sender등은 이 데코레이터를 상속받아 자신의 기능을 구현합니다.

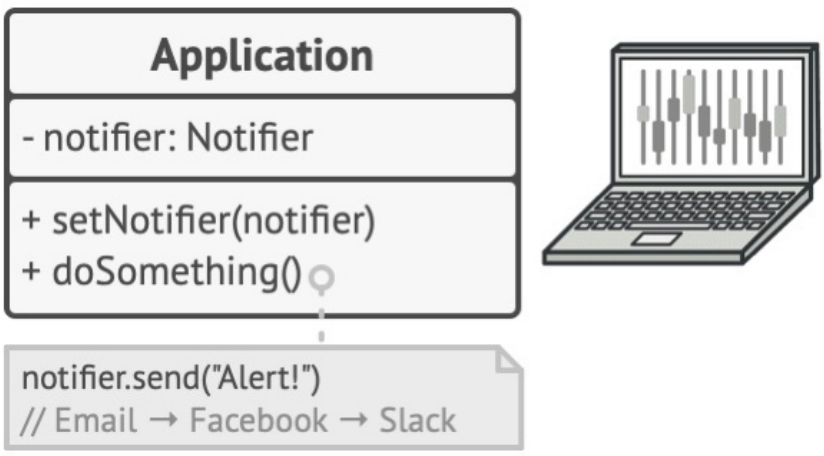
SMS,Facebook,Slack Decorator

super:send(message)를 통해 감싸여 있는 객체의 메세지 보내기 기능을 호출합니다.

그리고 그 호출이 완료되면, 자신만의 메시지 보내기 기능을 수행합니다.

application에서 이 notifier를 사용하는 코드

```
//notifier가 stack형태로 기능이 추가됩니다.
Notifier stack = new EmailSender();
Notifier stack = new FacebookDecorator(stack);
Notifier stack = new SlackDecorator(stack);
app.setNotifier(stack)
```



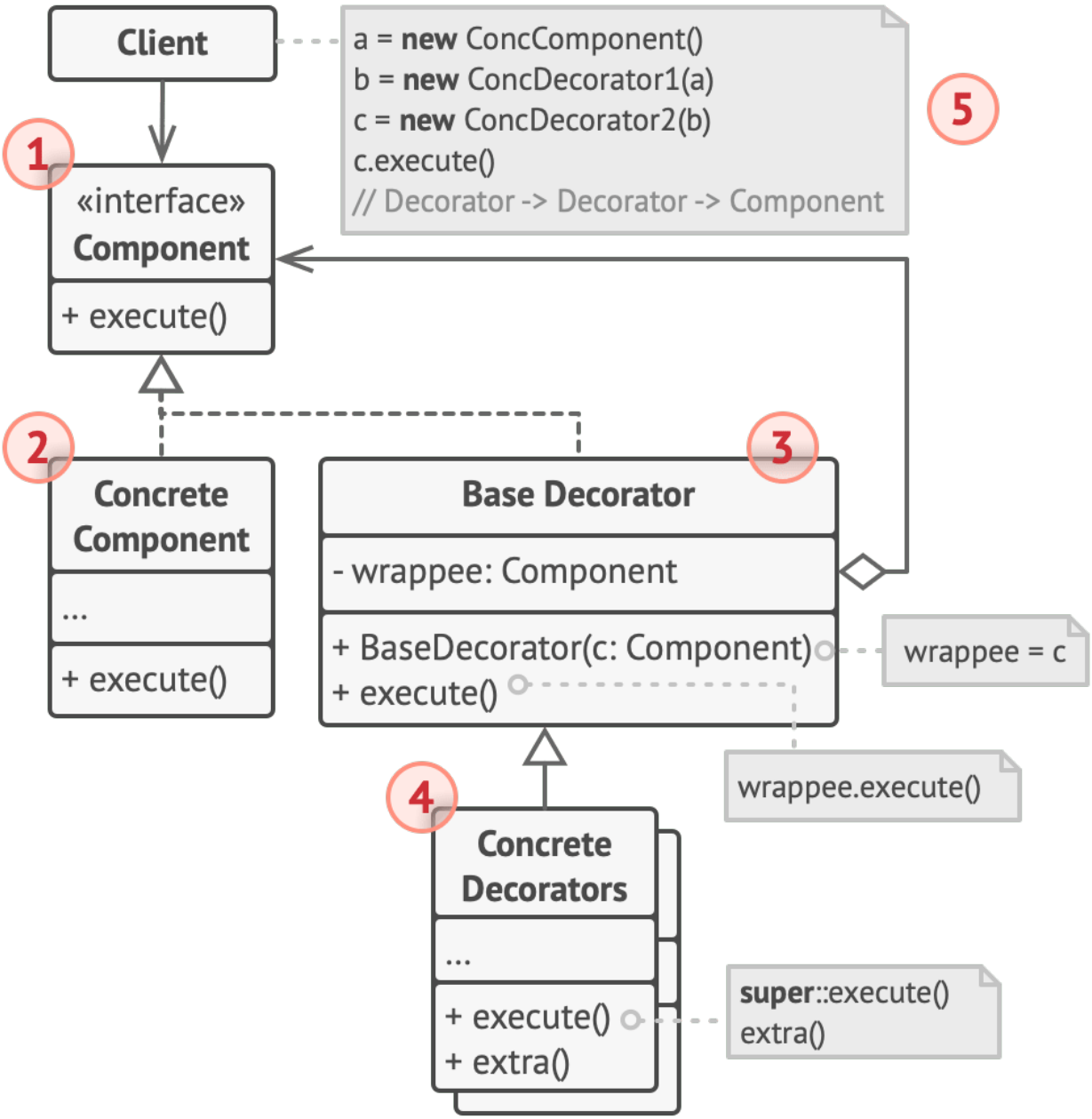
```
app.doSomething(message);
```

EmailSender 따로 뺀 이유

위처럼 재귀적으로 메서드가 실행된다면 base case를 두어야 재귀의 호출이 종료되겠죠. 이에 따라 emailsender가 그 역할을 해주는 것입니다.

만약에 email sender또한 BaseDecorator를 상속받도록 했다면, wrapee가 없으므로 super.send()에서 에러가 발생하겠지요.

구조



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.

구조를 보며... 떠오른 생각

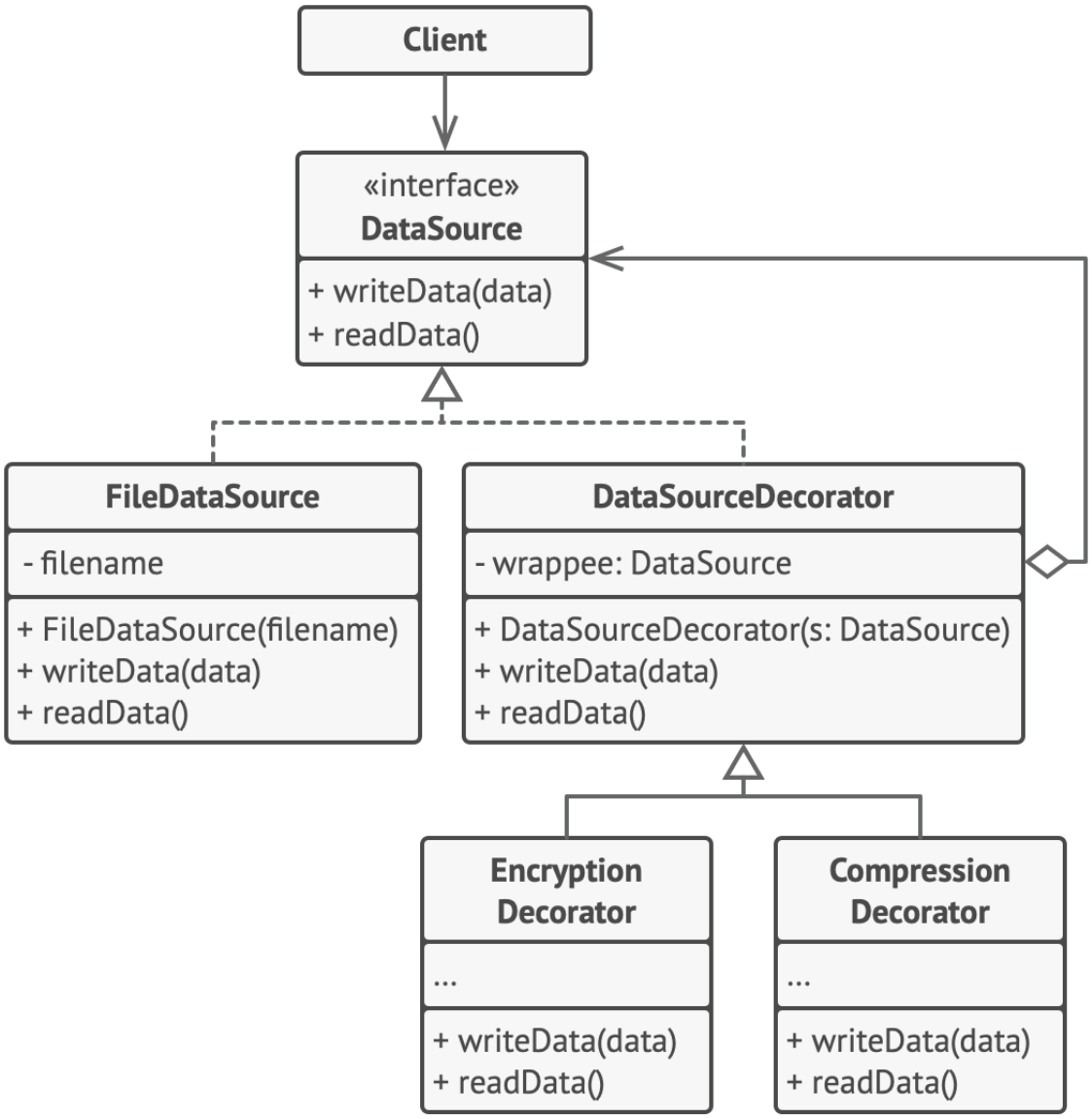
- BaseDecorator를 보면 인터페이스를 그대로 따르지 않음. LSP위반이라 볼 수 있다?

하지만... 클라이언트 관점에서 LSP위반이라 말할 수 있을까?!!

놉! 로버트마틴이 말하길, 클라이언트 관점에서 행동이 동일한 지로 판단해야 함.

따라서 아님.

또 다른 예 - 코드 구현



소개한 이유

이전 예제

1. 위 예제에서 EmailSender는 기초라고 보기엔 SmsDecorator, FacebookDecorator...와 동일한 계층을 띄고 있습니다.
2. 그래서 페이스북만 알림을 보내고 싶다면 페이스북 component 또한 만들어주는 것이 맞는 것 같습니다.

소개할 예제

1. 이 예제는 원본 파일을 읽고 쓸 수 있지만,(concreteComponent)
2. 데코레이터에 의해 원본 파일의 읽고 쓰는 기능에 추가하여 암호화되서, 그리고 압축되어 쓸 수 있습니다.(Decorator)
3. 따라서 약간의 차이가 있는 예제이기 때문에 소개드립니다.

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
```