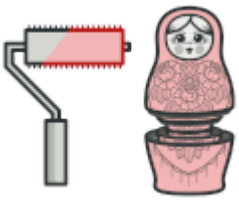

Decorator in Java

Decorator is a structural pattern that allows adding new behaviors to objects dynamically by placing them inside special wrapper objects, called *decorators*.

Using decorators you can wrap objects countless number of times since both target objects and decorators follow the same interface. The resulting object will get a stacking behavior of all wrappers.

Learn more about Decorator →

Navigation

Complexity: ★★☆

Popularity: ★★☆

streams.

Here are some examples of Decorator in core Java libraries:

- All subclasses of `java.io.InputStream`, `OutputStream`, `Reader` and `Writer` have constructors that accept objects of their own type.
- `java.util.Collections`, methods `checkedXXX()`, `synchronizedXXX()` and `unmodifiableXXX()`.
- `javax.servlet.http.HttpServletRequestWrapper` and `HttpServletResponseWrapper`

Identification: Decorator can be recognized by creation methods or constructors that accept objects of the same class or interface as a current class.

Encoding and compression decorators

This example shows how you can adjust the behavior of an object without changing its code.

Initially, the business logic class could only read and write data in plain text. Then we created several small wrapper classes that add new behavior after executing standard operations in a wrapped object.

The first wrapper encrypts and decrypts data, and the second one compresses and extracts data.

You can even combine these wrappers by wrapping one decorator with another.

decorators

decorators/DataSource.java: A common data interface, which defines read and write operations

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

public interface DataSource {
    void writeData(String data);

    String readData();
}
```

```
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;
    }
```

```
    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(compLe
            dos.write(data);
            dos.close();
            bout.close();
```

```
            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());
```