Wzorce projektowe 3

ADAPTER

1. Kod klasy SquarePeg:

```
package adapter;

public class SquarePeg {
    private final int width;

    public SquarePeg(int width) {
        this.width = width;
    }

    public int getWidth() {
        return this.width;
    }
}
```

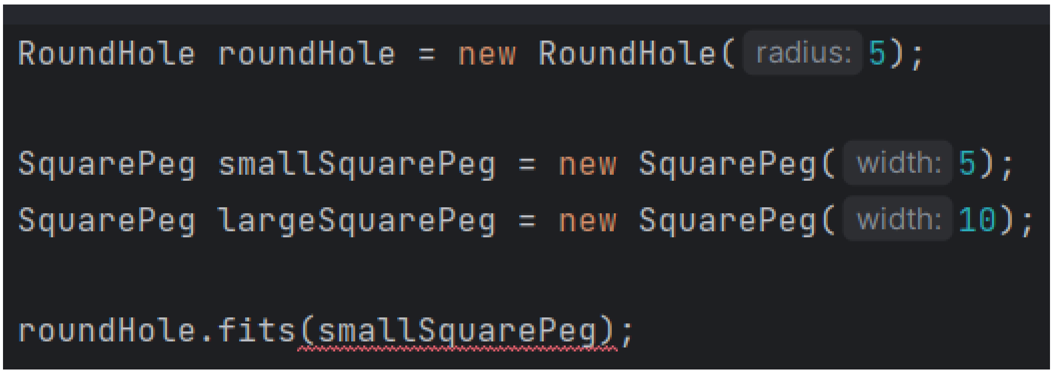
2. Kod klasy RoundPeg:

```
package adapter;

public class RoundPeg {
    private int radius;

    public RoundPeg() {}

    public RoundPeg(int radius) {
        this.radius = radius;
    }

    public int getRadius() {
        return this.radius;
    }
}
```

3. Kod klasy RoundHole:

```
package adapter;

public class RoundHole {
    private final int radius;

    public RoundHole(int radius) {
        this.radius = radius;
    }

    public int getRadius() {
        return this.radius;
    }

    public boolean fits(RoundPeg roundPeg) {
        return roundPeg.getRadius() <= this.getRadius();
    }
}
```

4. Kod klasy `Main`:



```
RoundHole roundHole = new RoundHole( radius: 5);

SquarePeg smallSquarePeg = new SquarePeg( width: 5);
SquarePeg largeSquarePeg = new SquarePeg( width: 10);

roundHole.fits(smallSquarePeg);
```

Widzimy problemy z kompilacją kodu, ponieważ klasy `SquarePeg` oraz `RoundHole` niestety nie są ze sobą kompatybilne. Będziemy potrzebowali adaptera, który zamiast długości boku będzie porównywać odpowiednią długość promienia.

5. Adapter wygląda następująco:

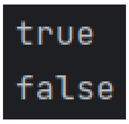
```
package adapter;

public class SquarePegAdapter extends RoundPeg {
    private final SquarePeg squarePeg;

    public SquarePegAdapter(SquarePeg squarePeg) {
        super(squarePeg.getWidth());
        this.squarePeg = squarePeg;
    }

    @Override
    public int getRadius() {
        return (int) (squarePeg.getWidth() * Math.sqrt(2) / 2);
    }
}
```

Najważniejszą metodą w niej jest `getRadius()`, gdzie faktycznie odbywa się mapowanie długości boku na promień okręgu.

6. Teraz kod klasy `Main` wygląda następująco:

```
import adapter.RoundHole;
import adapter.SquarePeg;
import adapter.SquarePegAdapter;

public class Main {
    public static void main(String[] args) {
        RoundHole roundHole = new RoundHole(5);

        SquarePeg smallSquarePeg = new SquarePeg(5);
        SquarePeg largeSquarePeg = new SquarePeg(10);

        SquarePegAdapter smallSquarePegAdapter = new
SquarePegAdapter(smallSquarePeg);
        SquarePegAdapter largeSquarePegAdapter = new
SquarePegAdapter(largeSquarePeg);

        System.out.println(roundHole.fits(smallSquarePegAdapter));
        System.out.println(roundHole.fits(largeSquarePegAdapter));
    }
}
```

7. Na standardowym wyjściu możemy zobaczyć:



```
true
false
```

DECORATOR

1. Dodano interfejs, który będą implementować wszystkie istotne klasy we wzorcu dekorator, a więc konkretne dekoratory, klasa nadrzędna nad konkretnymi dekoratorami oraz sam dekorowany obiekt.

```
package org.example.decorator;

import java.io.IOException;

public interface DataSource {

    void writeData(String data) throws IOException;
    String readData();
}
```

2. Klasa zapisująca i odczytująca z pliku udekorowany napis:

```
package org.example.decorator;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class FileDataSource implements DataSource {

    private final String filePath;

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

    @Override
    public void writeData(String data) {
        Path path = Paths.get(getFilePath());
        try {
            Files.write(path, data.getBytes());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public String readData() {
        try {
            return new
String(Files.readAllBytes(Paths.get(filePath)));
        } catch (IOException e) {
            System.out.println(e.getMessage());
            return null;
        }
    }

    private String getFilePath() {
        return this.filePath;
    }
}
```

3. Klasa szyfrująca i deszyfrująca napis:

```
package org.example.decorator;

import java.io.IOException;
import java.util.Base64;

public class EncryptionDecorator extends DataSourceDecorator {

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

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

    @Override
    public String readData() {
        return decode(super.readData());
    }
    private String encode(String data) {
        String encodedString =
Base64.getEncoder().encodeToString(data.getBytes());
        System.out.println("Zaszyfrowany napis:   " +
encodedString);
        return encodedString;
    }

    private String decode(String data) {
        String decodedString = new
String(Base64.getDecoder().decode(data));
        System.out.println("Odszyfrowany napis:   " +
decodedString);
        return decodedString;
    }
}
```

4. Klasa kompresująca i dekompresująca napis:

```
package org.example.decorator;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.Base64;
import java.util.zip.DataFormatException;
import java.util.zip.Deflater;
import java.util.zip.Inflater;

public class CompressionDecorator extends DataSourceDecorator {

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

    @Override
    public String readData() {
        return decompress(super.readData());
    }
    @Override
    public void writeData(String data) throws IOException {
        super.writeData(compress(data));
    }

    private String decompress(String data) {
        byte[] input = Base64.getDecoder().decode(data);

        Inflater decompressor = new Inflater();
        decompressor.setInput(input);

        ByteArrayOutputStream outputStream = new
ByteArrayOutputStream();
        byte[] buffer = new byte[1024];
        try {
            while (!decompressor.finished()) {
                int bytesRead = decompressor.inflate(buffer);
                outputStream.write(buffer, 0, bytesRead);
            }
        } catch (DataFormatException e) {
            e.printStackTrace();
        } finally {
            decompressor.end();
        }

        String decompressedString =
outputStream.toString(StandardCharsets.UTF_8);
        System.out.println("Zdekompresowany napis:   " +
decompressedString);

        return decompressedString;
    }

    private String compress(String data) {
        byte[] inputData = data.getBytes(StandardCharsets.UTF_8);

        Deflater deflater = new Deflater();
        deflater.setInput(inputData);
        deflater.finish();

        ByteArrayOutputStream outputStream = new
ByteArrayOutputStream(inputData.length);
        byte[] buffer = new byte[1024];
        while (!deflater.finished()) {
            int bytesCompressed = deflater.deflate(buffer);
            outputStream.write(buffer, 0, bytesCompressed);
        }
        deflater.end();

        String compressedString = new
String(Base64.getEncoder().encode(outputStream.toByteArray()));
        System.out.println("Skompresowany napis:   " +
compressedString);

        return compressedString;
    }
}
```

5. I na końcu bazowa klasa dekoratorów:

```
package org.example.decorator;

import java.io.IOException;

public class DataSourceDecorator implements DataSource {

    protected final DataSource wrappee;

    public DataSourceDecorator(DataSource dataSource) {
        this.wrappee = dataSource;
    }

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

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

6. Kod klienta prezentuje się następująco:

```
package org.example;

import org.example.decorator.CompressionDecorator;
import org.example.decorator.DataSource;
import org.example.decorator.EncryptionDecorator;
import org.example.decorator.FileDataSource;

import java.io.IOException;

public class Main {

    public static void main(String[] args) throws IOException {
        String stringToCompressAndEncrypt = "nypel";
        String filePath =
"./src/main/java/org/example/decorator/data.txt";
        DataSource dataSource = new CompressionDecorator(new
EncryptionDecorator(new FileDataSource(filePath)));
        dataSource.writeData(stringToCompressAndEncrypt);
        dataSource.readData();
    }
}
```

Widzimy tutaj w jak zgrabny sposób jesteśmy w stanie istniejącemu obiektowi dodawać coraz to nowe funkcjonalności bez modyfikacji istniejącego kodu.

COMMAND

1. Dodano klasę abstrakcyjną `Command`, którą będą rozszerzać konkretne implementacje poleceń w edytorze tekstu (między innymi kopiowanie i wklejanie tekstu).

```
package org.example.command.commands;

import org.example.command.Application;
import org.example.command.Editor;

```

```java
package org.example.command.commands;

import org.example.command.Application;
import org.example.command.Editor;

public abstract class Command {

    protected Application app;
    protected Editor editor;
    protected String backup;

    public Command(Application app, Editor editor) {
        this.app = app;
        this.editor = editor;
    }

    public void saveBackup() {
        this.backup = editor.getJTextArea().getText();
    }

    public void undo() {
        this.editor.getJTextArea().setText(backup);
    }

    public abstract boolean execute();
}
```

2. Implementacja polecenia kopiowania:

```java
package org.example.command.commands;

import org.example.command.Application;
import org.example.command.Editor;

public class CopyCommand extends Command {

    public CopyCommand(Application app, Editor editor) {
        super(app, editor);
    }

    @Override
    public boolean execute() {
        app.clipboard = editor.getSelection();
        return false;
    }
}
```

3. Implementacja polecenia wycinania:

```
package org.example.command.commands;

import org.example.command.Application;
import org.example.command.Editor;

public class CutCommand extends Command {

    public CutCommand(Application app, Editor editor) {
        super(app, editor);
    }

    @Override
    public boolean execute() {
        saveBackup();
        app.clipboard = editor.getSelection();
        editor.deleteSelection();
        return true;
    }
}
```

4. Implementacja polecenia wklejania:

```
package org.example.command.commands;

import org.example.command.Application;
import org.example.command.Editor;

public class PasteCommand extends Command {

    public PasteCommand(Application app, Editor editor) {
        super(app, editor);
    }

    @Override
    public boolean execute() {
        saveBackup();
        editor.replaceSelection(app.clipboard);
        return true;
    }
}
```

5. Implementacja polecenia cofania zmian:

```
package org.example.command.commands;

import org.example.command.Application;
import org.example.command.Editor;

public class UndoCommand extends Command {

    public UndoCommand(Application app, Editor editor) {
        super(app, editor);
    }

    @Override
    public boolean execute() {
        app.undo();
        return false;
    }
}
```

6. Chcemy móc przechowywać historię wykonanych operacji w edytorze tak, aby w razie potrzeby wrócić do poprzedniej wersji tekstu. Pomaga nam w tym klasa `CommandHistory`.

```
package org.example.command;

import org.example.command.commands.Command;

import java.util.LinkedList;

public class CommandHistory {

    private final LinkedList<Command> history = new
LinkedList<>();

    public void push(Command command) {
        history.push(command);
    }

    public Command pop() {
        return history.pop();
    }

    public boolean isEmpty() { return history.isEmpty(); }
}
```

7. Klasa `Editor` pomaga nam w zarządzaniu tekstową zawartością edytora, czyli tym co jest w nim wyświetlane. Ta klasa jest odbiorcą, czyli zawiera metody, które zostaną wykonane w odpowiedzi na polecenia dostarczane od klasy nadawcy.

```
package org.example.command;

import javax.swing.*;

public class Editor {

    private final JTextArea jTextArea = new JTextArea();

    public String getSelection() {
        return this.jTextArea.getText();
    }

    public void deleteSelection() {
        this.jTextArea.setText("");
    }

    public void replaceSelection(String text) {
        this.jTextArea.setText(text);
    }

    public JTextArea getJTextArea() {
        return this.jTextArea;
    }
}
```

8. Na sam koniec mamy klasę `Application`, która zawiera graficzną implementację edytora tekstu oraz metody wywołujące odpowiednie polecenia do edytora. Ta klasa jest nadawcą, czyli inicjuje wykonanie konkretnego polecenia.

```
package org.example.command;

import org.example.command.commands.*;

import javax.swing.*;
import java.awt.*;

public class Application {

    private final Editor activeEditor = new Editor();
    private final CommandHistory history = new CommandHistory();
    public String clipboard;

    public void createUI() {
        JFrame frame = new JFrame("Text editor (type & use
buttons, Luke!)");
        JPanel content = new JPanel();
        frame.setContentPane(content);

frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        content.setLayout(new BoxLayout(content,
BoxLayout.Y_AXIS));
        JTextArea textField = this.activeEditor.getJTextArea();
        textField.setLineWrap(true);
        content.add(textField);

        JPanel buttons = new JPanel(new
FlowLayout(FlowLayout.CENTER));
        JButton ctrlC = new JButton("Ctrl+C");
        JButton ctrlX = new JButton("Ctrl+X");
        JButton ctrlV = new JButton("Ctrl+V");
        JButton ctrlZ = new JButton("Ctrl+Z");
        Application application = this;
        ctrlC.addActionListener(e -> executeCommand(new
CopyCommand(application, this.activeEditor)));
        ctrlX.addActionListener(e -> executeCommand(new
CutCommand(application, this.activeEditor)));
        ctrlV.addActionListener(e -> executeCommand(new
PasteCommand(application, this.activeEditor)));
        ctrlZ.addActionListener(e -> executeCommand(new
UndoCommand(application, this.activeEditor)));

        buttons.add(ctrlC);
        buttons.add(ctrlX);
        buttons.add(ctrlV);
        buttons.add(ctrlZ);
        content.add(buttons);

        frame.setSize(450, 200);
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

    public void undo() {
        if (history.isEmpty()) return;

        Command command = history.pop();
        if (command != null) {
            command.undo();
        }
    }

    private void executeCommand(Command command) {
        if (command.execute()) history.push(command);
    }
}
```