7 Command-Muster

Das Command-Muster dient zur Entkoppelung von auslösender und ausführender Klasse.

Im folgenden Beispiel soll das Command-Muster für Dokumente umgesetzt werden und die Funktionen „Öffnen“ und „Speichern“ ermöglichen. Sie können die Funktionen durch eine Ausgabe auf der Console simulieren, oder eine konkrete Implementierung zum Speichern von Text in einer Datei verwenden.

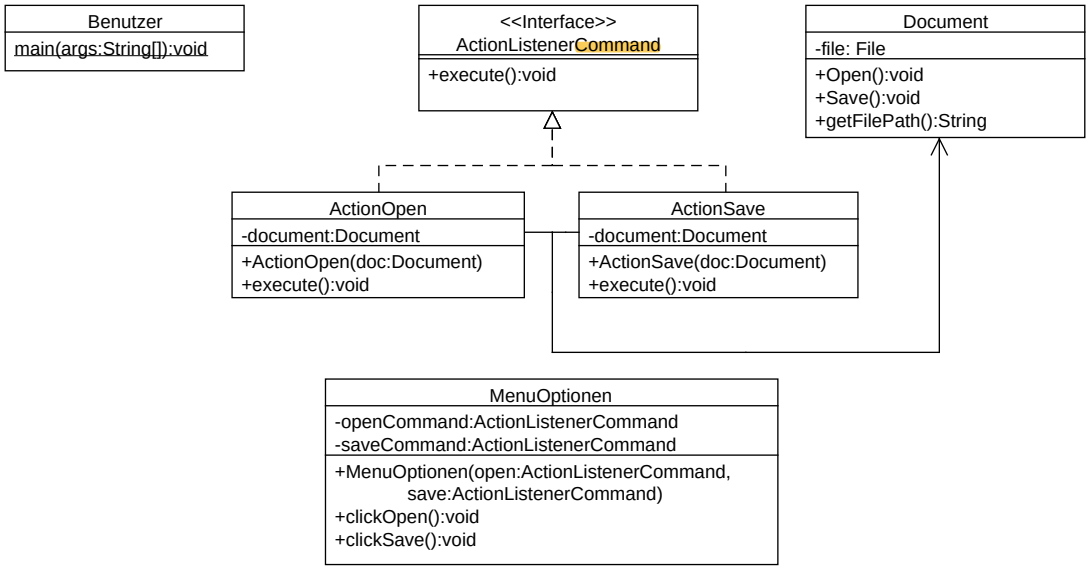


Abbildung 1: Command-Muster

Aufgaben

1. Setzen Sie die Methoden „Save“ und „Open“ der Klasse `Document` um. Verwenden Sie entweder eine Konsolenausgabe oder speichern Sie einen beliebigen Textinhalt in einer Datei.

2. Implementieren Sie den Konstruktor der Klassen `ActionOpen` und `ActionSave` und überschreiben Sie die `execute`-Methode beider Klassen, welche die Operationen auf dem `Document` ausführt. Geben Sie zusätzlich in die Konsole einen Hinweis über die durchgeführte Operation aus.

3. Implementieren Sie die auslösende Klasse `MenuOptionen`. Im Konstruktor werden die konkreten Command-Implementierungen zugewiesen. Die beiden übrigen Methoden dienen zur „Simulation“ eines Mausklicks auf ein fiktives Menü und führen die `execute`-Methode der entsprechenden Commands aus (Normalerweise würden die Commands aus einem Menü heraus ausgelöst werden.)

4. Simulieren Sie in der `main`-Methode der Klasse Benutzer die Benutzeraktionen. Zunächst wird ein neues Dokument und die `ActionListenerCommand`s erzeugt. Danach instanzieren Sie eine `MenuOptionen`-Klasse und führen die Aktionen für das Öffnen und Speichern aus.

Lösung:

Document.java

```
package command.dokumente;

import java.io.File;

public class Document {

    private File file;

    public Document(String path) {
        file = new File(path);
    }

    public Document(File file) {
        this.file = file;
    }

    public void open() {
        System.out.println("File opened");
    }

    public void save() {
        System.out.println("File saved");
    }

    public String getFilePath() {
        return file.toString();
    }

}
```

ActionListenerCommand.java

```
package command.dokumente;

public interface ActionListenerCommand {

    public void execute();

}
```

ActionOpen.java

```
package command.dokumente;
```

```
public class ActionOpen implements ActionListenerCommand {

    private Document document;

    public ActionOpen(Document doc) {
        document = doc;
    }

    @Override
    public void execute() {
        System.out.println("Dokument " + document.getFilePath() + "
        ↪  ge ffnet");
        document.open();
    }

}
```

ActionSave.java

```
package command.dokumente;

public class ActionSave implements ActionListenerCommand {

    private Document document;

    public ActionSave(Document doc) {
        document = doc;
    }

    @Override
    public void execute() {
        System.out.println("Dokument " + document.getFilePath() + "
        ↪  gespeichert");
        document.save();
    }

}
```

MenuOptionen.java

```
package command.dokumente;

public class MenuOptionen {

    private ActionListenerCommand openCommand;
    private ActionListenerCommand saveCommand;
```

```
    public MenuOptionen(ActionListenerCommand openCommand,
            ActionListenerCommand saveCommand) {
        this.openCommand = openCommand;
        this.saveCommand = saveCommand;
    }

    public void clickSave() {
        saveCommand.execute();
    }

    public void clickOpen() {
        openCommand.execute();
    }

}
```

Benutzer.java

```
package command.dokumente;

public class Benutzer {

    public static void main(String[] args) {
        Document doc = new Document("foo.txt");
        ActionListenerCommand commmandOpen = new ActionOpen(doc);
        ActionListenerCommand commmandSave = new ActionSave(doc);
        MenuOptionen options = new MenuOptionen(commmandOpen,
        ↪  commmandSave);
        options.clickOpen();
        options.clickSave();
    }

}
```

7.1 Command-Stack

Das Command-Muster wird häufig für die Aktionen „Rückgängig“ und „Wiederholen“ in Editoren und Programmen genutzt. Dabei werden die Commands in chronologischer Reihenfolge auf einen `Stack` (gleichnamige Java-Klasse) geschoben.

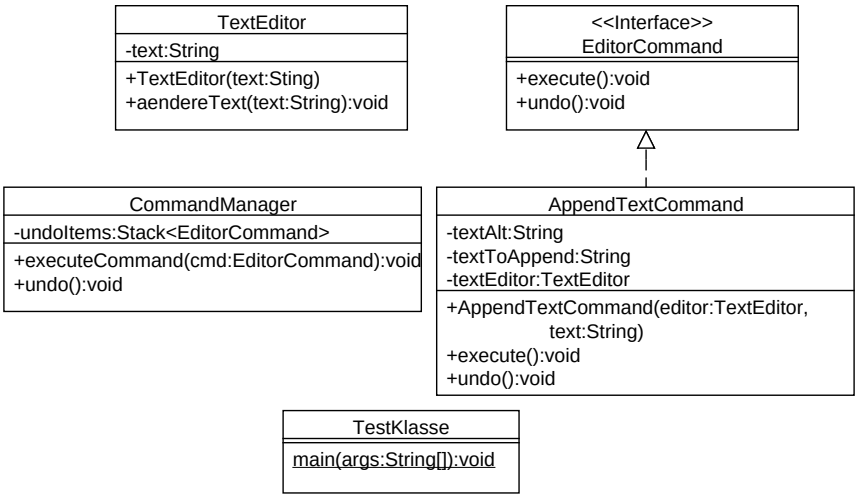


Abbildung 2: Command-Stack

Aufgaben

1. Implementieren Sie die Klasse TextEditor, inklusive eines Getters für den Text. Beim Ausführen der Methode `aendereText` soll der Inhalt des Attributes geändert und der neue Text auf der Console ausgegeben werden.

2. Implementieren Sie die Klassen `EditorCommand` und `AppendTextCommand`. Im Konstruktor des konkreten Commands wird der alte Text des Editors in das Attribut `alterText` gespeichert (und der anzuhängende Text im Attribut `textToAppend`). Beim Ausführen der `execute`-Methode wird der Text im Editor um `textToAppend` erweitert. Beim Ausführen der `Undo`-Methode, wird der Text des Editors auf `alterText` gesetzt.

3. Implementieren Sie die `CommandManager`-Klasse, welche sich um das Verwalten der Commands kümmert. Beim Ausführen der Commands wird das Command auf den Stack gepusht (Java-Methode `Stack.push()`). Wird die Methode `undo` aufgerufen, so wird das oberste Command auf dem Stack mittels `Stack.pop()` zurückgeholt und auf ihm die Undo-Operation ausgeführt.

4. Erstellen Sie eine Test-Klasse, die alle notwendigen Klassen instanziert und den Text des Editors mittels der Commands erweitert und die Änderungen wieder rückgängig macht.

Lösung:

CommandManager.java

```
package command.texteditor;

import java.util.Stack;
```

```
public class CommandManager {

    private Stack<EditorCommand> undoItems = new
    ↪  Stack<EditorCommand>();

    public void executeCommand(EditorCommand cmd) {
        undoItems.push(cmd);
        cmd.execute();
    }

    public void undo() {
        EditorCommand cmd = undoItems.pop();
        cmd.undo();
    }

}
```

AppendTextCommand.java

```
package command.texteditor;

public class AppendTextCommand implements EditorCommand {

    private String textAlt;
    private String textToAppend;
    private TextEditor textEditor;

    public AppendTextCommand(TextEditor editor, String text) {
        textEditor = editor;
        textAlt = editor.getText();
        textToAppend = text;
    }

    @Override
    public void execute() {
        textAlt = textEditor.getText();
        textEditor.aendereText(textEditor.getText() + textToAppend);
    }

    @Override
    public void undo() {
        textEditor.aendereText(textAlt);
    }

}
```

EditorCommand.java

```java
package command.texteditor;

public interface EditorCommand {

    public void execute();
    public void undo();

}
```

TextEditor.java

```java
package command.texteditor;

public class TextEditor {

    private String text;

    public TextEditor(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }

    public void aendereText(String text) {
        this.text = text;
        System.out.println(this.text);
    }

}
```

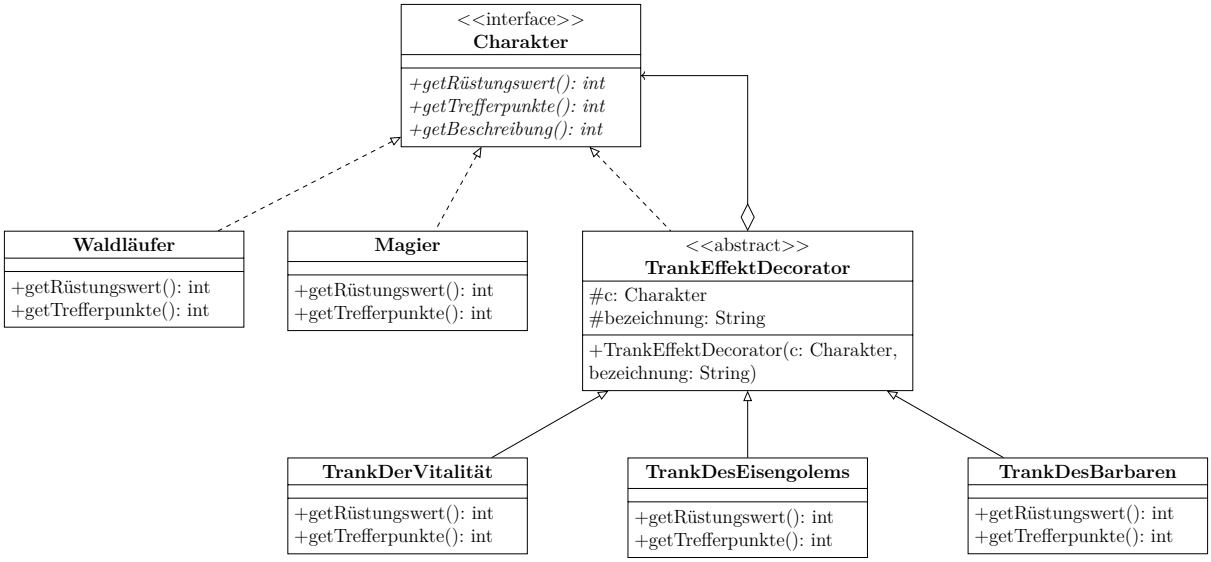
TestKlasse.java

```java
package command.texteditor;

public class TestKlasse {

    public static void main(String[] args) {
        TextEditor editor = new TextEditor("Some text");
        AppendTextCommand cmd1 = new AppendTextCommand(editor, " and
        ↪  some more text.");
        AppendTextCommand cmd2 = new AppendTextCommand(editor, " And
        ↪  even more text.");
```

```
        CommandManager manager = new CommandManager();
        manager.executeCommand(cmd1);
        manager.executeCommand(cmd2);
        manager.undo();
    }

}
```

Decorator-Muster

Für Ihr erstes selbstprogrammiertes RPG (Roleplaying Game) „MusterHelden“ soll der Spieler aus verschiedenen Klassen wählen können. Die Charaktere haben einen Wert für Trefferpunkte und für Rüstung. Während des Spiels kann der Charakter verschiedene Tränke zu sich nehmen die seine Werte zu verändern. Sie verwenden dafür das Decorator-Muster mit dem folgenden UML-Diagramm:



Charaktere der Klasse „Waldläufer“ haben anfangs 10 Rüstung und 20 Trefferpunkte, „Magier“ hingegen nur 3 Rüstung und 8 Trefferpunkte.

Die Tränke haben folgende Effekte:

Trank der Vitalität	Trefferpunkte +10
Trank des Eisengolems	Rüstung x2
Trank des Barbaren	Trefferpunkte +5, Rüstung -5

Aufgaben

1. Setzen Sie das UML-Diagramm in Programmcode um.

Lösung:

Charakter.java

```
public interface Charakter {
    int getRuestungswert();
    int getTrefferpunkte();

    String getBezeichnung();
}
```

Waldlaeufer.java

```
public class Waldlaeufer implements Charakter {
    static final int DEFAULT_TP = 20;
    static final int DEFAULT_RW = 10;

    @Override
    public int getRuestungswert() {
        return DEFAULT_RW;
    }

    @Override
    public int getTrefferpunkte() {
        return DEFAULT_TP;
    }

    @Override
    public String getBezeichnung() {
        return "Waldläufer";
    }
}
```

Magier.java

```
public class Magier implements Charakter{
    static final int DEFAULT_TP = 8;
    static final int DEFAULT_RW = 3;

    @Override
    public int getRuestungswert() {
        return DEFAULT_RW;
    }

    @Override
    public int getTrefferpunkte() {
        return DEFAULT_TP;
```

```
    }

    @Override
    public String getBezeichnung() {
        return "Magier";
    }
}
```

TrankEffektDekorator.java

```
public abstract class TrankEffektDecorator implements Charakter {
    protected Charakter c;
    protected String bezeichnung;

    public TrankEffektDecorator(Charakter c, String bezeichnung) {
        this.c = c;
        this.bezeichnung = bezeichnung;
    }

    public String getBezeichnung() {
        return c.getBezeichnung() + "; " + bezeichnung;
    }
}
```

TrankDerVitalitaet.java

```
public class TrankDerVitalitaet extends TrankEffektDecorator {
    private static final String BEZEICHNUNG = "Trank der
    ↪ Vitalität";

    public TrankDerVitalitaet(Charakter c) {
        super(c, BEZEICHNUNG);
    }

    @Override
    public int getRuestungswert() {
        return c.getRuestungswert();
    }

    @Override
    public int getTrefferpunkte() {
        return c.getTrefferpunkte() + 10;
    }
}
```

TrankDesEisengolems.java

```java
public class TrankDesEisengolems extends TrankEffektDecorator{
    private static final String BEZEICHNUNG = "Trank des
    ↪ Eisengolems";

    public TrankDesEisengolems(Charakter c) {
        super(c, BEZEICHNUNG);
    }

    @Override
    public int getRuestungswert() {
        return c.getRuestungswert() * 2;
    }

    @Override
    public int getTrefferpunkte() {
        return c.getTrefferpunkte();
    }
}
```

TrankDesBarbaren.java

```java
public class TrankDesBarbaren extends TrankEffektDecorator{
    private static final String BEZEICHNUNG = "Trank des
    ↪ Barbaren";

    public TrankDesBarbaren(Charakter c) {
        super(c, BEZEICHNUNG);
    }

    @Override
    public int getRuestungswert() {
        return c.getRuestungswert() - 5;
    }

    @Override
    public int getTrefferpunkte() {
        return c.getTrefferpunkte() + 5;
    }
}
```

2. Testen Sie die Berechnung mit einer Testklasse, die verschiedene Kombinationsmöglichkeiten testet, indem sie mehrere Charaktere mit verschiedenen Trankeffekten erzeugen.

Tipp

Ein Magier der einen Trank der Vitalität getrunken hat wird über `Character mag = new TrankDerVitalität(new Magier());` erzeugt.

Lösung:

Main.java

```
public class Main {
    public static void main(String[] args) {
        Charakter drizztdourden = new Waldlaeufer();

        drizztdourden = new TrankDerVitalitaet(drizztdourden);
        drizztdourden = new TrankDesEisengolems(drizztdourden);
        drizztdourden = new TrankDesEisengolems(drizztdourden);
        drizztdourden = new TrankDesBarbaren(drizztdourden);
        drizztdourden = new TrankDesEisengolems(drizztdourden);

        showCharacter(drizztdourden);
    }

    private static void showCharacter(Charakter charakter) {
        System.out.printf(charakter.getBezeichnung() +
                " (TP: " + charakter.getTrefferpunkte() +
                ", RW: " + charakter.getRuestungswert() + ")");
    }
}
```