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.

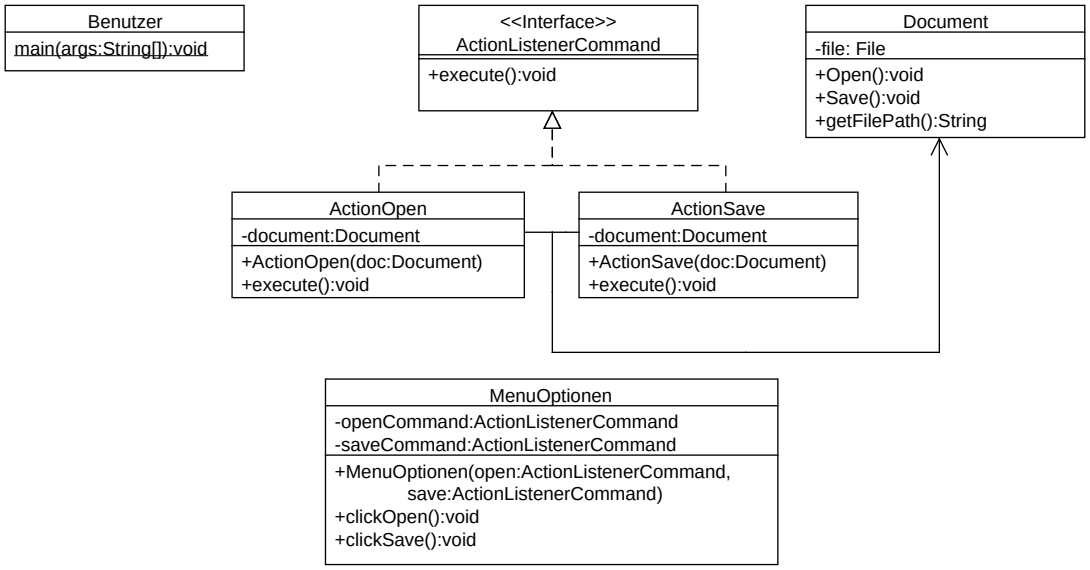


Abbildung 1: Command-Muster

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.

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.

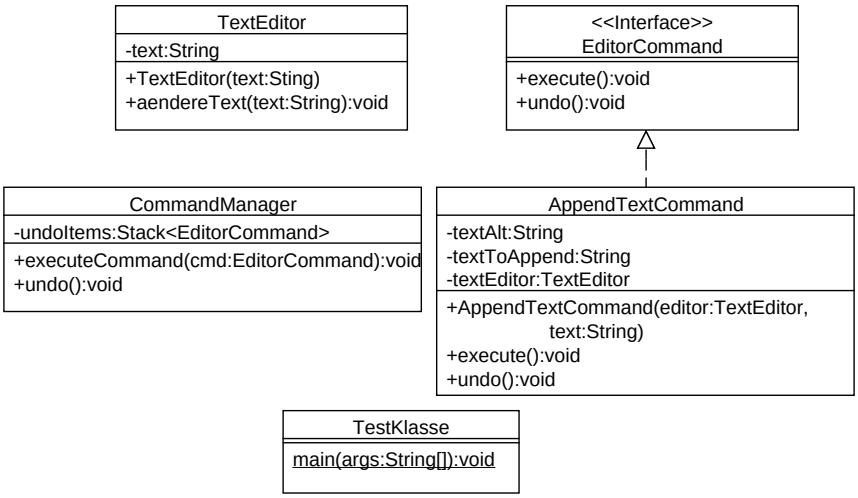


Abbildung 2: Command-Stack

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.

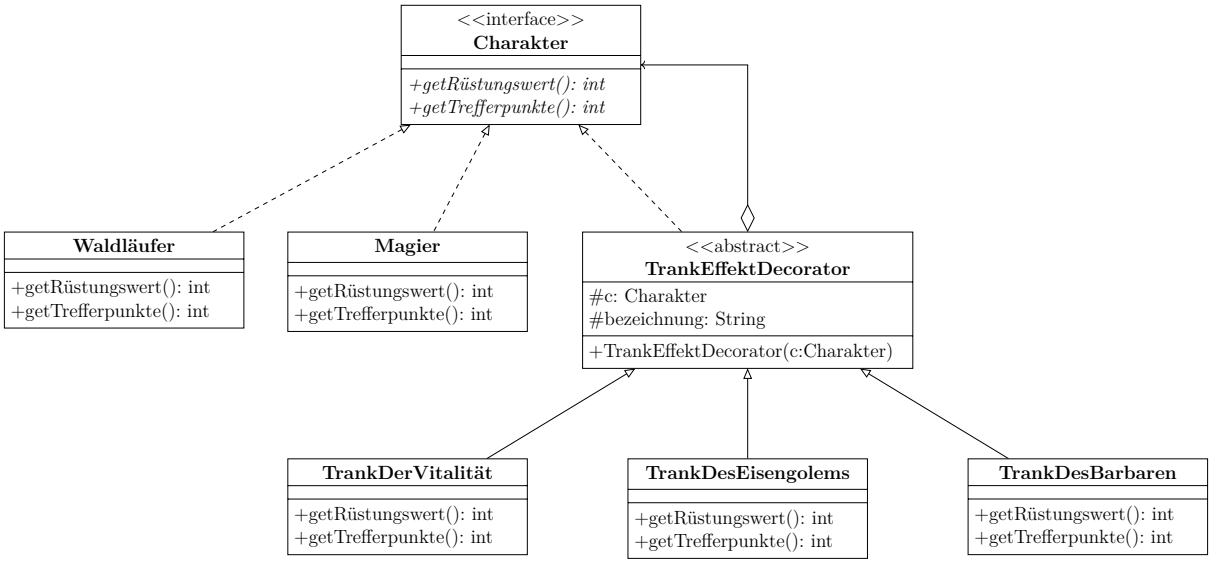
2. Implementieren Sie die Klassen `EditorCommand` und `AppendTextCommand`. Im Konstruktor des konkreten Commands wird der alte Text des Editors in das Attribut `alterText` gespeichert (und der anzuhängende Text im Attribut `textToAppend`). Beim Ausführen der `execute`-Methode wird der Text im Editor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

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.
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.