Modellierung zustandsorientierter Systeme in Java: Das Zustandsmuster, Varianten und Alternativen

Julian Fietkau, Janina Nemec

28. August 2009

Seminararbeit im Seminar

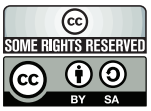
„Konzepte objektorientierter Programmiersprachen“

im Sommersemester 2009

Dozenten: Axel Schmolitzky, Christian Späh

Seminar 64-162 im Sommersemester 2009
Arbeitsbereich Softwaretechnik (SWT)
Department Informatik
Universität Hamburg

Inhaltsverzeichnis



Zusammenfassung

Dieses Paper befasst sich mit der Umsetzung von zustandsbasiertem Verhalten in der Programmiersprache Java. Das Entwurfsmuster „Zustand“ wird erklärt und anhand eines Beispiels in mehreren Varianten in Java umgesetzt. Weitere Möglichkeiten der Umsetzung werden diskutiert, insbesondere die Verwendung moderner Java-Sprachmittel in Abweichung vom strukturellen Entwurf des ursprünglichen Musters.

1. Einführung

Zustandsbasiertes Verhalten ist eines der zentralen Paradigmen, die in der Beschreibung von Hard- und Softwaresystemen zum Einsatz kommen. Hierbei werden Entitäten (z.B. Geräte mit Mikroprozessoren, ganze Softwareprogramme oder auch Objekte im OOP-Sinne) so programmiert, dass sie eine bestimmte Menge an Zuständen einnehmen können, zwischen denen sie auf einen Impuls hin wechseln. Abhängig davon, in welchem Zustand sich eine Entität befindet, kann sie ein spezielles Verhalten zeigen.

Dieses zustandsbasierte Verhalten ist in der Softwareentwicklung immer wieder relevant. Deshalb gehen wir zum Einstieg auf einige typische Einsatzzwecke ein. Im Anschluss grenzen wir den Bereich des Themas ab, mit dem wir uns in diesem Paper eingehend beschäftigen.

1.1. Motivation

In der theoretischen Informatik gibt es eine ganze Hierarchie von zustandsbasierten Modellen, die unterschiedlich mächtig sind. Die Turing-Maschine ist etwa ein Modell, das neben einem einem Speicherband über eine endliche Zustandsmenge verfügt, zwischen denen Wechsel stattfinden. Der einfache endliche Zustandsautomat liest dagegen sequenziell seine Eingabedaten und wechselt dabei zwischen seinen Zuständen. Modelle wie diese sind in der Systemtheorie gängige Mittel der Spezifikation. Ebenso ist zustandsabhängiges Verhalten ein intrinsisches Merkmal des Von-Neumann-Modells, auf dem fast alle heute verbreiteten Rechnerarchitekturen aufbauen.

Als gängiges Beispiel gilt das Netzwerkprotokoll TCP. Dessen Spezifikation gibt das erforderliche Verhalten der Kommunikationspartner teilweise als endliche Automaten an.

Aufgrund der Verbreitung des Paradigmas stellt sich die Frage, wie zustandsbasiertes Verhalten in einer modernen objektorientierten Sprache sinnvoll umsetzbar ist. Eine verbreitete und anerkannte Antwort auf diese Frage ist das Zustandsmuster, welches erstmalig von Gamma et al. beschrieben und publiziert wurde (vgl. [GHJV95]).

In diesem Paper wollen wir das Zustands-Entwurfsmuster in Bezug auf die Programmiersprache Java analysieren und feststellen, wie das Muster sich auf die modernen Sprachmittel aktueller Java-Versionen übertragen lässt. Weiterhin wollen wir weitere Möglichkeiten untersuchen, zustandsbasiertes Verhalten in Java umzusetzen.

1.2. Thematische Eingrenzung

Bei unseren Untersuchungen beschränken wir uns auf Lösungsansätze für die Modellierung zustandsbasierter Systeme in Java. Die Umsetzung dieser Lösungen in anderen

Programmiersprachen oder gar unter anderen Programmierparadigmen als denen der Objektorientierung lassen wir außen vor.

Weil einige der berührten Themen fortgeschrittener Natur sind, setzen wir für das tiefgehende Verständnis dieses Papers zumindest solide Grundlagen in den folgenden Bereichen voraus:

- Prinzipien der objektorientierten Modellierung
- Grundidee und Zweck objektorientierter Entwurfsmuster
- Umgang mit den Sprachmitteln von Java

Die Kenntnis gängiger Zustandsautomaten-Modelle wie des endlichen Automaten kann weiterhin hilfreich sein.

2. Ein einfaches zustandsbasiertes System

In der Praxis werden zustandsbasierte Systeme oft sehr komplex. Mit steigenden Anforderungen müssen dann neue Zustände eingebaut und neues Verhalten entwickelt werden. Dies bringt die Notwendigkeit mit sich, saubere und flexible Methoden der Umsetzung des Zustandsverhaltens zu beherrschen.

Dennoch werden wir die verschiedenen Methoden in diesem Paper an einem einfachen und überschaubaren System demonstrieren. Auch wenn damit die Vorteile der verschiedenen Methoden nicht so sehr ins Auge fallen, sind wir der Meinung, dass ein kleines Beispiel der Verständlichkeit mehr nützt als ein riesiges System mit einer großen Menge von Zuständen.

Die folgende Spezifikation beschreibt das Beispiel, welches wir für dieses Paper ausgewählt haben:

> *Eine Lampe mit Kippschalter habe drei Zustände, „Volle Helligkeit“, „Halbe Helligkeit“ und „Licht aus“. Es gibt zwei mögliche Aktionen für Klienten des Systems, „Kippschalter links drücken“ und „Kippschalter rechts drücken“ (im Folgenden: „drücke links“ und „drücke rechts“).*
>
> *Befindet sich der Kippschalter in der Mittelstellung, so hat die Lampe den Zustand „Licht aus“. Die Aktion „drücke links“ führt dann zu einem Zustandswechsel nach „Volle Helligkeit“, „drücke rechts“ analog nach „Halbe Helligkeit“.*
>
> *Im Zustand „Volle Helligkeit“ bewirkt die Aktion „drücke links“ nichts, „drücke rechts“ führt zu einem Zustandswechsel nach „Licht aus“. Analoges gilt für den Zustand „Halbe Helligkeit“, „drücke rechts“ führt zu keinem Zustandsübergang und „drücke links“ zu einem Zustandswechsel nach „Licht aus“.*
>
> *Die aktuelle Helligkeit der Lampe kann als Zahl zwischen 0,0 und 1,0 abgefragt werden. Im Zustand „Licht aus“ liegt sie bei 0,0; im Zustand „Halbe Helligkeit“ bei 0,5 und im Zustand „Volle Helligkeit“ bei 1,0.*
>
> *Bei Inbetriebnahme des Systems ist „Licht aus“ der Startzustand.*

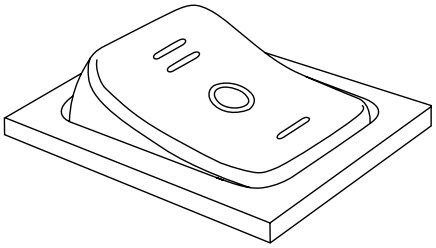


Abbildung 1: Schematische Darstellung des Kippschalters im Zustand „Halbe Helligkeit“

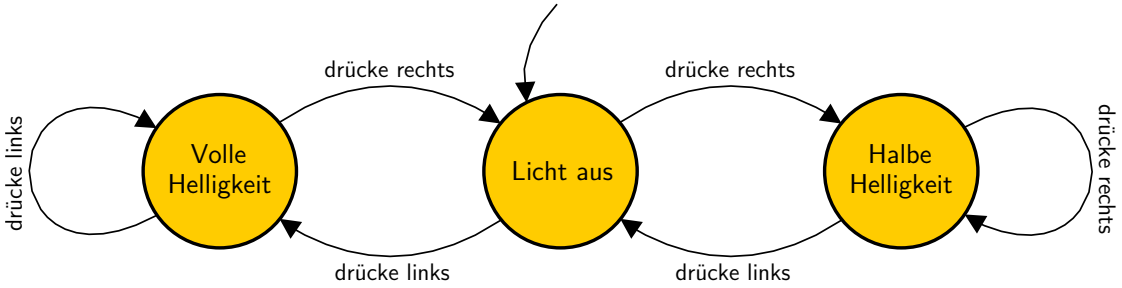


Abbildung 2: Zustandsdiagramm des Lichtschalters

Der genannte Kippschalter ist ein gängiger 3-Positionen-Kippschalter mit entsprechenden Markierungen (Abbildung 1).

Diese Spezifikation lässt sich nicht nur in Textform, sondern auch z.B. als äquivalentes Zustandsdiagramm ausdrücken (Abbildung 2).

In den nun folgenden Abschnitten wollen wir einige Möglichkeiten erläutern, diese Spezifikation in Java zu modellieren. Dabei werden wir die verschiedenen Ansätze auf Kriterien wie Fehleranfälligkeit und Erweiterbarkeit untersuchen.

3. Lösungsansatz 1: Die naive Umsetzung

Dieser erste Ansatz entspricht der intuitiven Herangehensweise. Es ist die Lösung, die normalerweise entsteht, wenn keine bewusste Entscheidung für eine bestimmte Umsetzung von zustandsbasiertem Verhalten gefallen ist.

3.1. Entwurf

Hierbei wird der aktuelle Zustand der Lampe in einem einfachen Zustandsfeld gehalten.

Im verbreitetsten Fall wählt die implementierende Person dafür den Typ `int` mit der Begründung, dass `boolean` nur zwei mögliche Werte hat, wogegen im vorliegenden Fall drei gebraucht werden. Die aktuelle Helligkeit der Lampe wird dann bei Bedarf ermittelt, indem jedem Zustand eine Helligkeit zugewiesen wird. Dazu dient häufig ein `switch`-Konstrukt. Eine solche Modellierung kann programmiert etwa dem Code von Listing 1 entsprechen.

Etwas sauberer und weniger fehleranfällig lässt der Code sich gestalten, indem statt eines Zustandsfelds vom Typ `int` ein privater innerer Aufzählungstyp verwendet wird. Diese Verbesserung löst einige der Probleme des Entwurfs, jedoch bei weitem nicht alle. Dieser Ansatz ist in Listing 2 umgesetzt.

3.2. Eigenschaften

Dieser Entwurf entspricht der verbreiteten intuitiven Herangehensweise und ist deshalb in den meisten Fällen auch ohne Kenntnis irgendwelcher Entwurfsmuster sofort verständlich.

Negativ fällt auf, dass die Spezifikation der Zustände von der des zustandsabhängigen Verhaltens vergleichsweise weit entfernt ist. Würde weiteres zustandsabhängiges Verhalten neben der `gibHelligkeit`-Methode nötig werden, so müssten an mehreren Stellen in der Klasse alle Zustände und ihr jeweiliges spezifisches Verhalten aufgezählt werden.

Das ist ein deutliches Zeichen für niedrige Kohäsion und läuft anerkannten Designprinzipien zuwider. Durch die immer wieder nötige Aufzählung der Zustände wird die Wartung bei veränderten Anforderungen stark erschwert.

Bei der Variante mit einer `int`-Variable kommt eine gesteigerte Fehleranfälligkeit hinzu. Durch Programmierfehler kann das Zustandsfeld einen ungültigen Wert annehmen, was undefinierte Konsequenzen haben kann. Um das zu verhindern, muss an allen Stellen, an denen der Zustand abgefragt wird, auf eventuelle fehlerhafte Werte reagiert werden (in unserem Beispiel in Listing 1 wurde darauf geachtet). Wird ein Aufzählungstyp verwendet, wird diese komplette Fehlerklasse ausgeschlossen.

Zusammenfassung

- \+ intuitiv schnell erfassbar
- – niedrige Kohäsion
- – erschwerte Wart- und Erweiterbarkeit
- – (ohne Aufzählungstyp: gesteigerte Anfälligkeit für Programmierfehler)

4. Lösungsansatz 2: Das Zustandsmuster

Bei Gamma et al. findet sich ein alternativer Vorschlag für die Implementation von zustandsbasierten Systemen (vgl. [GHJV95]). Das Entwurfsmuster „Zustand“ versucht, mit objektorientierten Mitteln die Zustände und ihr jeweiliges Verhalten besser zu kapseln und so die Kohäsion zu steigern und die Erweiterung zu erleichtern.

Die abstrakte Beschreibung des Musters ist sehr allgemein gehalten und macht keinen Gebrauch von Java-spezifischen Sprachfeatures. Wir empfehlen zum Selbststudium die Aufbereitung des Themas durch Freeman et al., welche das Muster anhand eines konkreten Beispiels mit Java-Code erklärt (vgl. [FFBS04]).

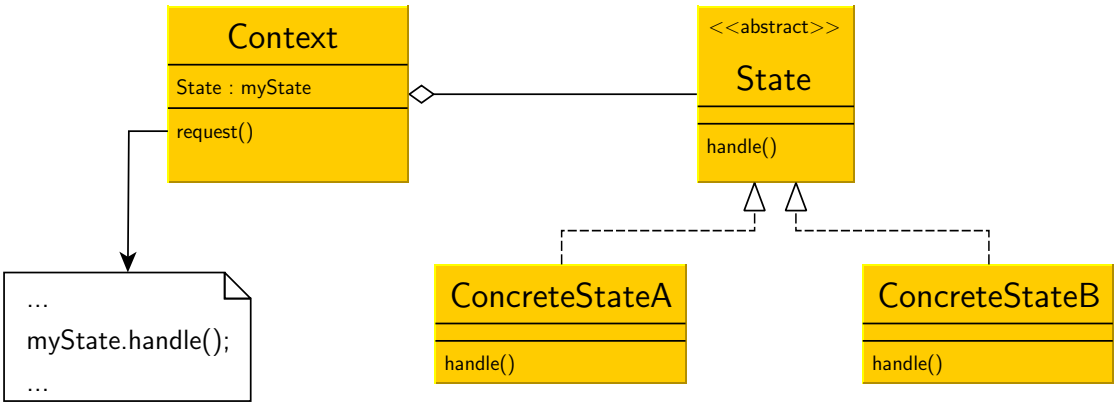


Abbildung 3: Klassendiagramm des Zustandsmusters nach Gamma et al.

4.1. Beschreibung des Musters

Das Zustandsmuster (engl.: *state pattern*), auch bekannt als „Objekte für Zustände“ (engl.: *objects for states*), gehört zur Kategorie der Verhaltensmuster (engl.: *behavioral patterns*) und ist eines der Muster der „Gang of Four“. Der Sinn des Zustandsmusters ist es, die Veränderung des Zustandes eines Objekts durch die scheinbare Änderung seiner Klasse darzustellen, indem Teile des Codes, der von dem Objekt ausgeführt wird, zur Laufzeit austauschbar gemacht werden.

Man kann dieses Entwurfsmuster prinzipiell überall dort anwenden, wo das Verhalten eines Objekts sich abhängig von seinem Zustand zur Laufzeit ändert. Der Einsatz lohnt sich insbesondere dann, wenn in einer Klasse an vielen Stellen im Code die gleiche Zustandsunterscheidung mittels `if`- oder `switch`-Konstrukten erfolgt.

Das Zustandsmuster besteht aus drei Teilen (vgl. Abbildung 3).

1. Den Rahmen bildet eine Kontextklasse, welche nach außen hin den fachlichen Gegenstand modelliert und intern ein Exemplar einer Zustandsklasse hält.

2. Weiterhin gibt es ein Interface oder eine abstrakte Klasse, die vorgibt, welche Operationen von jedem konkreten Zustand implementiert werden müssen. Des Weiteren ermöglicht der dadurch definierte Typ es dem Kontext, durch polymorphe Zuweisungen zwischen den konkreten Zuständen zu wechseln.

3. Außerdem müssen mindestens eine (sinnvollerweise mehrere) konkrete Zustandsklassen existieren, von denen jede für sich einen Zustand mit seinem Verhalten implementiert. Dieser Zustand ist zumeist durch eine Anzahl von zustandsspezifischen Konstanten und/oder einprogrammierte, je nach Zustand verschiedene Algorithmen verkörpert.

4.2. Entwurf

Nimmt man das vorangegangene Beispiel des Kippschalters, so kann man an der naiven Umsetzung erkennen, dass durch die vielen notwendigen `if`-Abfragen (in anderen Beispielen auch `switch`/`case`-Konstrukte) der Quelltext langsam unübersichtlich und kompliziert wird. Um solche Abfragen zu vermeiden, wird in diesem Muster jeder Zustand durch eine eigene Klasse abgebildet.

Bei unserem Beispiel sind das drei Zustandsklassen: „Volle Helligkeit“, „Halbe Helligkeit“ und „Licht aus“. Nun sind diese Zustandsklassen prinzipbedingt nicht völlig unterschiedlich. Ihre Operationen bilden eine einheitliche Schnittstelle. Deshalb definiert man ein Interface oder eine abstrakte Klasse, welche die gemeinsamen Operationen der konkreten Zustandsklassen vorgibt.

In unseren Quelltextbeispielen (Listings 3 und 4) wäre dies das Interface `LampeZustand`. Unsere konkreten Zustandsklassen sind entsprechend `LampeZustandHalbeHelligkeit` sowie `LampeZustandVolleHelligkeit` und `LampeZustandLichtAus`. Bei der Klasse `Lampe` handelt es sich um eine sogenannte Kontextklasse, welche die Zustandsklassen verwendet. Als Startzustand hält die Kontextklasse in einer Exemplarvariablen die Referenz auf ein Exemplar der vordefinierten Startzustandsklasse. Diese Startzustandklasse ist eine der konkreten Zustandsklassen. Ändert sich der Zustand, so wird die Referenz auf ein Exemplar der entsprechenden neuen Zustandsklasse umgebogen. Dadurch wird der aktuelle Zustand gespeichert. In den Listings 3 und 4 wäre das entsprechend die Variable `zustand`, die zu Beginn ein Exemplar der Klasse `LampeZustandLichtAus` hält.

Im Zustandsmuster ist nicht vorgegeben, wann Objekte erzeugt und zerstört werden sollten. Hierbei bieten sich zwei Möglichkeiten an:

- Eine Möglichkeit ist das Erzeugen des Zustandsobjektes bei Bedarf und die Zerstörung bei Zustandsänderung. Diese Alternative lässt sich den meisten Ausprägungen des Zustandsmusters einfach realisieren.

- Dann gibt es noch die Option, in der Kontextklasse ein Exemplar jeder Zustandsklasse zu erzeugen und die Referenzen z.B. in Exemplarvariablen zu speichern. Nun finden bei Zustandsübergängen Zuweisungen zur entsprechenden Variable statt. Bei dieser Möglichkeit werden die Objekte nie bzw. erst am Ende der Ausführung des Programms zerstört.

Die erste Alternative sollte man bevorzugen, wenn noch nicht klar ist, welche Zustände zur Laufzeit benötigt werden und die Frequenz der Zustandsübergänge nicht sehr hoch ist. Wechseln die Zustände jedoch häufig, so sollte man eher die zweite Option umsetzen.

Können mehrere Kontexte die gleichen Zustandsobjekte verwenden, kann Speicherplatz eingespart werden. Solange die Zustandsklassen selbst keinen veränderbaren Zustand besitzen, ist das theoretisch möglich. Es bringt jedoch auch die Gefahr mit sich, dass eine unnötig enge Kopplung zwischen zwei verschiedenen Kontexten entsteht.

Nun gibt das Zustandsmuster allerdings nicht vor, welche Klasse oder Klassen die Transitionsverwaltung zu übernehmen haben. Deshalb gibt es mehrere Möglichkeiten, welche Klassen die Transitionslogik beinhalten können. Welche der beiden Möglichkeiten

der Transitionsverwaltung genutzt werden sollte, hängt von der Anzahl der Zustände, von der Ausprägung des zustandsabhängigen Verhaltens und nicht zuletzt von den Vorlieben des Programmierers ab.

4.2.1. Dezentrale Transitionsverwaltung

Hierbei steckt die gesamte Transitionslogik in den konkreten Zustandsklassen. Dies bedeutet, dass die Kontextklasse Methodenaufrufe, die zu einem Zustandswechsel führen, an die aktive konkrete Zustandsklasse weitergibt.

In unserem Beispiel bedeutet das Delegation der Aufrufe der Methoden `drueckeLinks`, `drueckeRechts` und `gibHelligkeit` an die aktive konkrete Zustandsklasse über die Exemplarvariable `zustand`. Somit werden solche Methoden auch in dem Zustandsinterface oder der abstrakten Klasse `Zustand` vorgegeben. Erst in den konkreten Zustandsklassen wird dann definiert, was bei entsprechenden Aufrufen geschehen soll. Im konkreten Zustand sind Zustandsabfragen nun natürlich unnötig geworden und es kann direkt reagiert werden.

Die Methode `gibHelligkeit` aus unserem Beispiel gibt einen zustandsabhängigen Wert zurück und speichert somit zustandsrelevante Daten. Hingegen sind die beiden anderen Methoden `drueckeLinks` und `drueckeRechts` zustandsverändernde Methoden, denn sie geben jeweils einen Zustand zurück (sich selbst oder einen anderen) und beinhalten somit die Transitionslogik.

Dies bedeutet allerdings auch, dass jede Zustandsklasse ihre Nachfolgezustände kennt und führt in vielen Fällen zu zyklischen Abhängigkeiten. Solange sich die Implementation an den Rahmen des Zustandsmusters hält, sehen wir jedoch keinen Grund, den Entwurf dogmatisch abzulehnen. Zyklische Abhängigkeiten werden schließlich nur zum Problem, wenn mehrere Klassen gegenseitig Funktionalität der jeweils anderen Klasse verwenden, was hier nicht der Fall ist.

Diese Lösung entspricht der aus Listing 3 und ist in Abbildung 4 in Form eines Klassendiagramms veranschaulicht.

4.2.2. Zentrale Transitionsverwaltung

Bei dieser Möglichkeit steckt die Transitionslogik in der Kontextklasse. Hierbei werden alle zustandsverändernden Methodenaufrufe nicht nach unten delegiert, sondern nur das zustandsspezifische Verhalten wie in unserem Beispiel `gibHelligkeit`.

Die Zustände werden hierbei wieder durch `if`-Konstrukte abgefragt. Hier kennen die konkreten Zustandsklassen einander nicht. Das Zustandsinterface in diesem Beispiel gibt nun auch nur noch die Methode `gibHelligkeit` vor, welche auch als einzige in den konkreten Zustandsklassen implementiert wird.

Nun mag einem das Zustandsmuster in der Umsetzung dieses Beispiels als übertriebener Arbeitsaufwand vorkommen. Gibt es allerdings weiteres zustandsabhängiges Verhalten zusätzlich zu `gibHelligkeit`, so erspart dieses Vorgehen viele Abfragen.

In Listing 4 ist diese Variante umgesetzt. Abbildung 5 ist das dazugehörige Klassendiagramm.

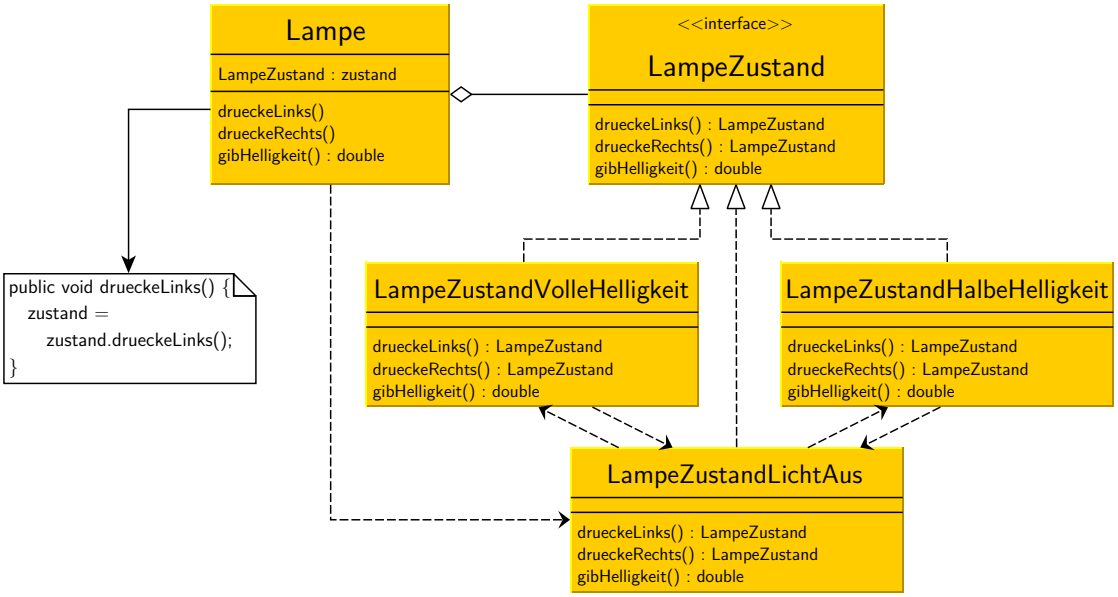


Abbildung 4: Klassendiagramm des Entwurfs aus Listing 3

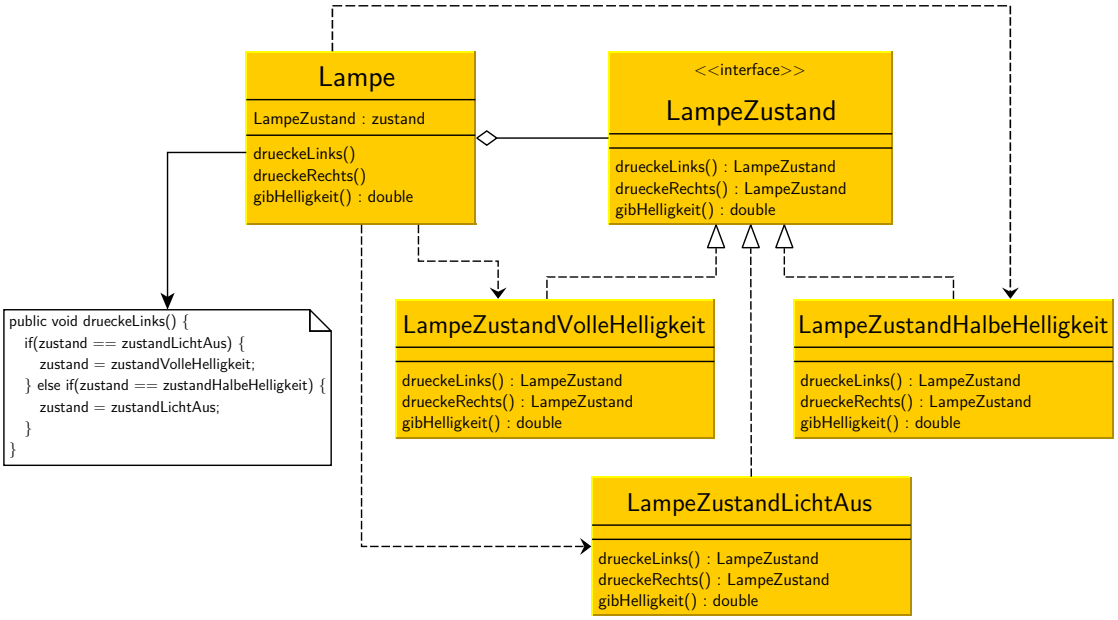


Abbildung 5: Klassendiagramm des Entwurfs aus Listing 4

4.3. Eigenschaften

Das Zustandsmuster lokalisiert zustandsspezifisches Verhalten: Durch das Aufteilen in Kontext und konkrete Zustandsklassen befindet sich das zustandspezifische Verhalten immer in den konkreten Zustandsklassen.

Wird ein neuer Zustand hinzugefügt, müssen lediglich zwei Dinge beachtet werden: Es muss eine neue konkrete Zustandsklasse angelegt werden, die entsprechende Methoden bereitstellt, welche das zustandsabhängige Verhalten beinhalten. Außerdem muss die Transitionslogik um die neu hinzugekommenen Zustandsübergänge erweitert werden. Dies erleichtert die Wartbarkeit.

Komplexe und unübersichtliche Schachtelungen von Bedingungsanweisungen können vermieden werden. Zudem spiegelt sich die Struktur des spezifizierenden Zustandsdiagramms bei dezentraler Transitionsverwaltung im Klassendiagramm wider: Zustandsübergänge sind in Form von „benutzt“-Beziehungen zwischen den konkreten Zustandsklassen im Klassendiagramm zu sehen. Bei zentraler Transitionslogik sind die Zustandsübergänge strikt vom zustandsspezifischen Verhalten getrennt.

Es bleibt zu bedenken, dass der Einsatz des Zustandsmusters meist zu einer komplexeren Struktur der Software und einer höheren Anzahl kleiner Klassen führt. Ist das zustandsspezifische Verhalten sehr überschaubar, lohnt sich der Mehraufwand in der Programmstruktur ggf. nicht. In solchen Fällen kann eine der Varianten, die wir in den folgenden Abschnitten vorstellen, angebrachter sein.

Zusammenfassung

+ saubere Entkopplung der Zustände untereinander und vom Kontext
+ gesteigerte Übersichtlichkeit der beteiligten Klassen
+ einfache Wart- und Erweiterbarkeit
+ langjährige Bewährtheit des Musters
− struktureller Mehraufwand

4.4. Verwandte Muster

Es gibt einige Entwurfsmuster, die häufig gemeinsam mit (oder anstelle vom) Zustandsmuster zum Einsatz kommen. Um in der Hinsicht ein wenig Klarheit zu schaffen, werden wir sie an dieser Stelle kurz beschreiben.

- Das **Fliegengewichtsmuster** (engl.: *flyweight pattern*) ist eines der strukturellen Muster (*structural patterns*) beschrieben durch Gamma et al. (vgl. [GHJV95]). Man setzt das Fliegengewicht ein, wenn es eine sehr hohe Zahl von identischen Objekten gibt, die durch die konsequente Anwendung dieses Musters durch ein einziges Objekt ersetzt werden können. Dieses Muster beinhaltet zwar einen recht großen Verwaltungsaufwand, aber es kann Speicherplatz gespart werden. Durch die

gemeinsame Nutzung von Zustandsobjekten durch verschiedene Kontexte kann in Kombination mit dem Zustandsmuster der Speicheraufwand reduziert werden.

- Das **Einzelstückmuster** (engl.: *singleton pattern*) ist ein von Gamma et al. beschriebenes erzeugendes Muster (*creational pattern*). Dieses Muster findet Verwendung, wenn nur eine begrenzte Anzahl von Exemplaren (in den meisten Fällen nur ein Exemplar) einer Klasse existieren soll. Um die Anzahl zu beschränken, wird meist eine statische Instanziierungsoperation definiert. Kombiniert mit dem Zustandsmuster kann z.B. die Anzahl der Exemplare der konkreten Zustandsklassen beschränkt werden.

- Das **Strategiemuster** (engl.: *strategy pattern*) ist ebenso wie das Zustandsmuster ein Verhaltensmuster und ähnelt diesem sehr stark. Es wird verwendet, wenn sich verwandte Klassen nur in Teilen ihres Verhaltens unterscheiden, um dieses Verhalten zu externalisieren und vom Rest der Klassen zu entkoppeln, welche dann vereinheitlicht werden können. Strukturell sind Strategiemuster und Zustandsmuster kaum zu unterscheiden, die Einsatzzwecke sind jedoch verschieden.

5. Lösungsansatz 3: Erweiterte Möglichkeiten von Java-Enums

Aufzählungstypen („Enums“) bieten in Java viele weiterführende, kaum bekannte Möglichkeiten (vgl. [GJSB05]). Die einzelnen Werte eines Aufzählungstypen sind in Java tatsächlich Exemplare einer ansonsten nicht instanziierbaren Klasse. Diese Klasse kann über weiterführende Logik verfügen. Sogar die einzelnen Werte können mit spezifischem Verhalten versehen werden.

Mit diesen Mitteln lässt sich eine dritte Möglichkeit etablieren, zustandsspezifisches Verhalten in Java elegant umzusetzen.

5.1. Entwurf

Ähnlich wie bei der Realisierung des Zustandsmusters mit dezentraler Transitionsverwaltung hält die Lampe ihren aktuellen Zustand in einem Zustandsfeld. In diesem Fall ist `LampeZustand` jedoch kein Interface, sondern ein Enum.

Dieser Enum definiert die drei möglichen Zustände. Dem Typ `LampeZustand` werden drei Operationen zugeordnet, die den zwei Aktionen „drücke links“ und „drücke rechts“ sowie der Abfrage der Helligkeit entsprechen. Jedem der drei Werte des Enums werden spezielle Implementationen dieser Operationen zugeordnet, so dass man an ihnen die Nachfolgezustände sowie die Helligkeit abfragen kann.

Dieser Entwurf entspricht der Implementation in Listing 5.

5.2. Eigenschaften

Diese Variante bringt wie das ursprüngliche Zustandsmuster eine enge Kopplung der einzelnen Zustände an ihr jeweiliges Verhalten mit sich.

Je nach Umfang und Komplexität des Systems kann es ein großer Nachteil sein, dass alle Zustände samt ihres kompletten Verhaltens innerhalb eines Aufzählungstyps angegeben werden. Dieser liegt bekanntermaßen innerhalb einer Übersetzungseinheit und einer Datei. Damit geht die lose Kopplung der verschiedenen Zustände teilweise verloren. Andererseits kann bei kleineren Systemen die höhere Übersichtlichkeit und der geringere strukturelle Mehraufwand auch ein Vorteil sein.

Als Auswirkung dieser Implementation ist weiterhin erwähnenswert, dass durch die Benutzung eines Aufzählungstyps für das komplette Verhalten an keiner Stelle im Code irgendwelche expliziten Instanziierungen stehen. Die einzigen Objekte, die erzeugt werden, sind die zu Beginn erzeugten und statisch verfügbaren Exemplare des Aufzählungstyps.

Zusammenfassung

+ enge Kopplung jedes Zustands mit seinem Verhalten
+ hohe Übersichtlichkeit bei kleinen Systemen
+ keine strukturelle Komplexität auf Klassenebene
− keine Aufteilung der Zustände in verschiedene Dateien möglich

6. Weitere Ansätze

Die in diesem Abschnitt behandelten Umsetzungen verfolgen weitere Herangehensweisen, die sich von den bereits behandelten so sehr unterscheiden, dass sie unserer Meinung nach erwähnenswert sind. Sie sind bis dato weniger praxiserprobt als die bereits aufgeführten Lösungen, haben aber dafür spezielle Eigenschaften, die bei bestimmten Problemstellungen von Vorteil sein können.

6.1. Objektbasiertes Zustandsgeflecht

Bei diesem Ansatz gibt es anders als beim Zustandsmuster nicht für jeden konkreten Zustand eine Klasse, sondern eine allgemeine Zustandsklasse. Die Exemplare dieser Klasse dienen dann als Modellierung der Zustände.

Die Aktionen werden als Enum externalisiert. Es können zur Laufzeit beliebig viele Zustände erzeugt werden. Jeder Zustand hält eine Abbildung von Aktionen auf Nachfolgezustände. Es ist dann zur Laufzeit möglich, den Nachfolgezustand eines Zustandes bei einer bestimmten Aktion zu setzen und wieder abzufragen.

Dieser Ansatz ist nur dann implementierbar, wenn die verschiedenen Zustände keine individuelle Logik besitzen müssen. Alles, was die Zustände unterscheidet (hier die Helligkeit), muss ihnen bei der Erzeugung übergeben werden und somit in Form von Daten vorliegen. Es wäre wohl theoretisch denkbar, den verschiedenen Zuständen mittels Polymorphie unterschiedliches Verhalten zuzuordnen, indem man unterhalb der Zustandsklasse wieder ein Zustandsmuster implementiert. Das wäre allerdings aus Designsicht

kaum zu rechtfertigen, da in dem Fall auch gleich das Zustandsmuster hätte implementiert werden können.

Die Spezifikation der möglichen Zustände und Übergänge geschieht hier in der Klasse, die den Kontext bildet.

Diese Umsetzung bietet als einzige der hier vorgestellten Varianten die Möglichkeit, das Zustandsverhalten zur Laufzeit zu ändern. Alle anderen Vorgehensweisen spezifizieren die Zustände und Übergänge statisch zur Übersetzungszeit. Hier kann dagegen zu beliebigen Zeitpunkten in der Ausführung das Zustandsverhalten verändert werden. Diese Eigenschaft kann von Vorteil sein, falls ein System modelliert werden soll, dessen Zustandsspezifikation sich tatsächlich im Laufe der Zeit ändert.

Andererseits ist die fehlende statische Analysierbarkeit auch der größte Nachteil im Vergleich zu anderen Lösungen. Zur Übersetzungszeit ist nicht feststellbar, welche Zustände existieren und welche Übergänge möglich sind. Dadurch wird die Überschaubarkeit und die Beherrschbarkeit des Systems verringert.

Eine Umsetzung der Kippschalter-Lampe nach diesem Prinzip findet sich in Listing 6.

6.2. Enum mit Transitionstabelle

Diese Variante entspricht in etwa der bereits behandelten Enum-Umsetzung. Hier erhalten die einzelnen Werte des Aufzählungstyps jedoch kein spezielles Verhalten in Form von Code, sondern jeder der Zustände bekommt seine möglichen Nachfolgezustände als Konstruktorparameter übergeben. Im Konstruktor werden diese gesichert und später bei Zustandswechseln ausgelesen.

Bei der Erzeugung der Objekte, die hinter den Enum-Werten stehen, können keine Werte referenziert werden, die im Enum erst nach dem zu erzeugenden Wert aufgezählt werden. Aus diesem Grund werden die Nachfolgezustände als Strings übergeben und gehalten. Beim Zustandswechsel wird mit `valueOf` der passende Wert des Enum ermittelt.

Konzeptuell sind die Unterschiede zur Standard-Enum-Umsetzung nicht sehr groß. Auszeichnend für diese Variante ist jedoch die Trennung von Spezifikation und Implementationslogik. Um Zustände hinzuzufügen oder Übergänge zu verändern muss kein Code-Geflecht angepasst werden, die Spezifikation der Übergänge erfolgt quasi tabellarisch.

Das Kippschalter-Beispiel ist in Listing 7 auf diese Weise umgesetzt.

7. Zusammenfassung

Es existiert eine Vielzahl von Möglichkeiten, zustandsbasierte Systeme in Java zu programmieren, die alle ihre Vor- und Nachteile haben. Eine allgemeine, eindeutige Empfehlung kann es nicht geben. Es ist individuelles Feingefühl gefragt, wenn es darum geht, welche Umsetzung in einer bestimmten Situation am besten geeignet ist.

In besonders einfachen Situationen, in denen keine spätere Erweiterung zu erwarten ist, bietet evtl. die naive Umsetzung das beste Verhältnis von Aufwand und Nutzen.

Gibt es dagegen eine Vielzahl von Zuständen, die im Laufe der Zeit erweiterbar sein soll, und hat jeder Zustand ein komplexes eigenes Verhalten, kann einer der Entwürfe

nach dem Vorbild des Zustandsmusters die beste Lösung sein.

Rechtfertigt die Situation den strukturellen Mehraufwand des Zustandsmusters nicht, ist ggf. eine Umsetzung mit einem Aufzählungstyp oder eine der weiteren Möglichkeiten das Mittel der Wahl.

Wir hoffen, dass mit den Erkenntnissen aus diesem Paper die nötige Entwurfsentscheidung bewusst und in Kenntnis aller wichtigen Alternativen gefällt werden kann.

Quelltext-Listings

Listing 1: Naive Umsetzung

```java
public class Lampe {

    // 0 = Licht aus
    // 1 = Halbe Helligkeit
    // 2 = Volle Helligkeit
    private int zustand = 0;

    public void drueckeLinks() {
        switch (zustand) {
            case 0: zustand = 2;
                    break;
            case 1: zustand = 0;
                    break;
            case 2: // Zustand unverändert
                    break;
            default: System.out.println("Zustand fehlerhaft!");
        }
    }

    public void drueckeRechts() {
        switch (zustand) {
            case 0: zustand = 1;
                    break;
            case 1: // Zustand unverändert
                    break;
            case 2: zustand = 0;
                    break;
            default: System.out.println("Zustand fehlerhaft!");
        }
    }

    public double gibHelligkeit() {
        double helligkeit = 0.0;
        switch (zustand) {
            case 0: helligkeit = 0.0;
                    break;
            case 1: helligkeit = 0.5;
                    break;
            case 2: helligkeit = 1.0;
                    break;
            default: System.out.println("Zustand fehlerhaft!");
        }
        return helligkeit;
    }
}
```

Lampe.java

Listing 2: Naive Umsetzung mit innerem Aufzählungstyp

```java
public class Lampe {

    private enum LampeZustand {
        LICHT_AUS, HALBE_HELLIGKEIT, VOLLE_HELLIGKEIT;
    }

    private LampeZustand zustand = LampeZustand.LICHT_AUS;

    public void drueckeLinks() {
        if(zustand == LampeZustand.LICHT_AUS) {
            zustand = LampeZustand.VOLLE_HELLIGKEIT;
        } else if(zustand == LampeZustand.HALBE_HELLIGKEIT) {
            zustand = LampeZustand.LICHT_AUS;
        }
    }

    public void drueckeRechts() {
        if(zustand == LampeZustand.VOLLE_HELLIGKEIT) {
            zustand = LampeZustand.LICHT_AUS;
        } else if(zustand == LampeZustand.LICHT_AUS) {
            zustand = LampeZustand.HALBE_HELLIGKEIT;
        }
    }

    public double gibHelligkeit() {
        double helligkeit = 0.0;
        if(zustand == LampeZustand.LICHT_AUS) {
            helligkeit = 0.0;
        } else if(zustand == LampeZustand.HALBE_HELLIGKEIT) {
            helligkeit = 0.5;
        } else if(zustand == LampeZustand.VOLLE_HELLIGKEIT) {
            helligkeit = 1.0;
        }
        return helligkeit;
    }
}
```

Lampe.java

Listing 3: Zustandsmuster mit dezentraler Transitionsverwaltung

```java
public class Lampe {

    private LampeZustand zustand;

    public Lampe() {
        zustand = new LampeZustandLichtAus();
    }

    public void drueckeLinks() {
        zustand = zustand.drueckeLinks();
    }

    public void drueckeRechts() {
        zustand = zustand.drueckeRechts();
    }

    public double gibHelligkeit() {
        return zustand.gibHelligkeit();
    }
}
```

Lampe.java

```java
public interface LampeZustand {
    public LampeZustand drueckeLinks();

    public LampeZustand drueckeRechts();

    public double gibHelligkeit();
}
```

LampeZustand.java

```java
public class LampeZustandLichtAus implements LampeZustand {

    public LampeZustand drueckeLinks() {
        return new LampeZustandVolleHelligkeit();
    }

    public LampeZustand drueckeRechts() {
        return new LampeZustandHalbeHelligkeit();
    }

    public double gibHelligkeit() {
        return 0.0;
    }
}
```

LampeZustandLichtAus.java

```java
public class LampeZustandHalbeHelligkeit implements LampeZustand {

    public LampeZustand drueckeLinks() {
        return new LampeZustandLichtAus();
    }

    public LampeZustand drueckeRechts() {
        return this;
    }

    public double gibHelligkeit() {
        return 0.5;
    }
}
```

LampeZustandHalbeHelligkeit.java

```java
public class LampeZustandVolleHelligkeit implements LampeZustand {

    public LampeZustand drueckeLinks() {
        return this;
    }

    public LampeZustand drueckeRechts() {
        return new LampeZustandLichtAus();
    }

    public double gibHelligkeit() {
        return 1.0;
    }
}
```

LampeZustandVolleHelligkeit.java

Listing 4: Zustandsmuster mit zentraler Transitionsverwaltung

```java
public class Lampe {

    private LampeZustand zustand;

    private static final LampeZustand zustandLichtAus =
        new LampeZustandLichtAus();
    private static final LampeZustand zustandHalbeHelligkeit =
        new LampeZustandHalbeHelligkeit();
    private static final LampeZustand zustandVolleHelligkeit =
        new LampeZustandVolleHelligkeit();

    public Lampe() {
        zustand = new LampeZustandLichtAus();
    }

    public void drueckeLinks() {
        if(zustand == zustandLichtAus) {
            zustand = zustandVolleHelligkeit;
        } else if(zustand == zustandHalbeHelligkeit) {
            zustand = zustandLichtAus;
        }
    }

    public void drueckeRechts() {
        if(zustand == zustandVolleHelligkeit) {
            zustand = zustandLichtAus;
        } else if(zustand == zustandLichtAus) {
            zustand = zustandHalbeHelligkeit;
        }
    }

    public double gibHelligkeit() {
        return zustand.gibHelligkeit();
    }
}
```

Lampe.java

```java
public interface LampeZustand {
    public double gibHelligkeit();
}
```

LampeZustand.java

```java
public class LampeZustandLichtAus implements LampeZustand {
    public double gibHelligkeit() {
        return 0.0;
    }
}
```

LampeZustandLichtAus.java

```java
public class LampeZustandHalbeHelligkeit implements LampeZustand {
    public double gibHelligkeit() {
        return 0.5;
    }
}
```

LampeZustandHalbeHelligkeit.java

```java
public class LampeZustandVolleHelligkeit implements LampeZustand {
    public double gibHelligkeit() {
        return 1.0;
    }
}
```

LampeZustandVolleHelligkeit.java

Listing 5: Externer Zustand als Aufzählungstyp

```java
public class Lampe {

    private LampeZustand zustand;

    public Lampe() {
        zustand = LampeZustand.LICHT_AUS;
    }

    public void drueckeLinks() {
        zustand = zustand.drueckeLinks();
    }

    public void drueckeRechts() {
        zustand = zustand.drueckeLinks();
    }

    public double gibHelligkeit() {
        return zustand.gibHelligkeit();
    }
}
```

Lampe.java

```java
public enum LampeZustand {

    LICHT_AUS {
        public LampeZustand drueckeLinks() {
            return VOLLE_HELLIGKEIT;
        }

        public LampeZustand drueckeRechts() {
            return HALBE_HELLIGKEIT;
        }

        public double gibHelligkeit() {
            return 0.0;
        }
    },
    HALBE_HELLIGKEIT {
        public LampeZustand drueckeLinks() {
            return LICHT_AUS;
        }

        public LampeZustand drueckeRechts() {
            return HALBE_HELLIGKEIT;
        }

        public double gibHelligkeit() {
            return 0.5;
        }
```

```
    },
    VOLLE_HELLIGKEIT {
        public LampeZustand drueckeLinks() {
            return VOLLE_HELLIGKEIT;
        }

        public LampeZustand drueckeRechts() {
            return LICHT_AUS;
        }

        public double gibHelligkeit() {
            return 1.0;
        }
    };

    public LampeZustand drueckeLinks() {
        return this;
    }

    public LampeZustand drueckeRechts() {
        return this;
    }

    public double gibHelligkeit() {
        return 0.0;
    }
}
```

 LampeZustand.java

Listing 6: Objektbasiertes Zustandsgeflecht

```java
public class Lampe {

    private LampeZustand zustand;

    public Lampe() {

        // Zustände festlegen
        LampeZustand lichtAus = new LampeZustand(0.0);
        LampeZustand halbeHelligkeit = new LampeZustand(0.5);
        LampeZustand volleHelligkeit = new LampeZustand(1.0);

        // Übergänge festlegen
        lichtAus.setzeFolgezustand(LampeAktion.DRUECKE_LINKS,
            volleHelligkeit);
        lichtAus.setzeFolgezustand(LampeAktion.DRUECKE_RECHTS,
            halbeHelligkeit);
        halbeHelligkeit.setzeFolgezustand(LampeAktion.DRUECKE_LINKS,
            lichtAus);
        halbeHelligkeit.setzeFolgezustand(LampeAktion.DRUECKE_RECHTS,
            halbeHelligkeit);
        volleHelligkeit.setzeFolgezustand(LampeAktion.DRUECKE_LINKS,
            volleHelligkeit);
        volleHelligkeit.setzeFolgezustand(LampeAktion.DRUECKE_RECHTS,
            lichtAus);

        // Startzustand festlegen
        zustand = lichtAus;
    }

    public void drueckeLinks() {
        zustand = zustand.gibFolgezustand(LampeAktion.DRUECKE_LINKS);
    }

    public void drueckeRechts() {
        zustand = zustand.gibFolgezustand(LampeAktion.DRUECKE_RECHTS);
    }

    public double gibHelligkeit() {
        return zustand.gibHelligkeit();
    }
}
```

Lampe.java

```java
public enum LampeAktion {
    DRUECKE_LINKS, DRUECKE_RECHTS;
}
```

LampeAktion.java

```java
import java.util.HashMap;
import java.util.Map;

public class LampeZustand {

    private final double helligkeit;
    private final Map<LampeAktion, LampeZustand> uebergaenge =
        new HashMap<LampeAktion, LampeZustand>();

    public LampeZustand(double helligkeit) {
        this.helligkeit = helligkeit;

        // Default: kein Übergang, sondern im aktuellen Zustand bleiben
        for(final LampeAktion aktion : LampeAktion.values()) {
            uebergaenge.put(aktion, this);
        }
    }

    public LampeZustand gibFolgezustand(LampeAktion aktion) {
        return uebergaenge.get(aktion);
    }

    public void setzeFolgezustand(LampeAktion aktion,
        LampeZustand zustand) {
        uebergaenge.put(aktion, zustand);
    }

    public double gibHelligkeit() {
        return helligkeit;
    }
}
```

LampeZustand.java

Listing 7: Externer Zustand als Aufzählungstyp mit Transitionstabelle

```java
public class Lampe {

    private LampeZustand zustand;

    public Lampe() {
        zustand = LampeZustand.LICHT_AUS;
    }

    public void drueckeLinks() {
        zustand = zustand.drueckeLinks();
    }

    public void drueckeRechts() {
        zustand = zustand.drueckeLinks();
    }

    public double gibHelligkeit() {
        return zustand.gibHelligkeit();
    }
}
```

Lampe.java

```java
public enum LampeZustand {

    // Zustand(folgezustandLinks, folgezustandRechts, helligkeit)
    LICHT_AUS("VOLLE_HELLIGKEIT", "HALBE_HELLIGKEIT", 0.0),
    HALBE_HELLIGKEIT("LICHT_AUS", "HALBE_HELLIGKEIT", 0.5),
    VOLLE_HELLIGKEIT("VOLLE_HELLIGKEIT", "LICHT_AUS", 1.0);

    private final String links;
    private final String rechts;
    private final double helligkeit;

    private LampeZustand(String links, String rechts, double helligkeit)
    {
        this.links = links;
        this.rechts = rechts;
        this.helligkeit = helligkeit;
    }

    public LampeZustand drueckeLinks() {
        return valueOf(links);
    }

    public LampeZustand drueckeRechts() {
        return valueOf(rechts);
    }

    public double gibHelligkeit() {
```

```
        return helligkeit;
    }
}
```

LampeZustand.java

Literatur

[FFBS04] Freeman, Elisabeth ; Freeman, Eric ; Bates, Bert ; Sierra, Kathy: *Head First Design Patterns.* Sebastopol : O'Reilly Media, 2004

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston : Addison-Wesley, 1995

[GJSB05] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad: *Java Language Specification.* Third Edition. Boston : Addison-Wesley, 2005