WIKIPEDIA

Zustand (Entwurfsmuster)

Der **Zustand** (englisch *state*) ist ein Entwurfsmuster aus dem Bereich der Softwareentwicklung, das zur Kategorie der Verhaltensmuster (englisch *behavioral design patterns*) gehört. Das Zustandsmuster wird zur Kapselung unterschiedlicher, zustandsabhängiger Verhaltensweisen eines Objektes eingesetzt.[1]

Das Zustandsmuster ist eines der sogenannten "GoF"-Muster, d. h. es ist eines der im Buch *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software* aufgeführten Entwurfsmuster ("GoF" steht für "Gang of Four" oder "Viererbande" nach den vier Autoren dieses 1994 veröffentlichten Buches). Das Zustandsmuster ist auch bekannt als *Objekte für Zustände* (*objects for states*).

Inhaltsverzeichnis

Verwendung

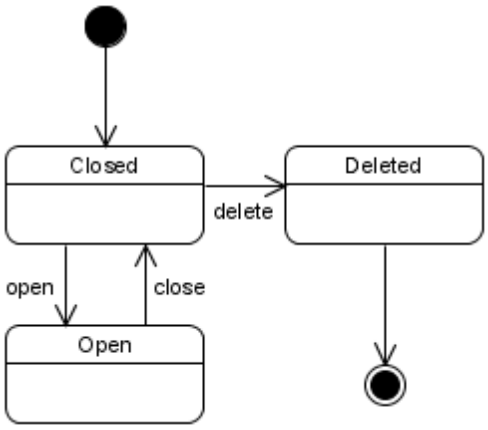
Grundsätzlich gilt, dass das Verhalten eines Objekts abhängig von seinem Zustand ist. Durch die übliche Implementierung soll vermieden werden, die Zustände eines Objekts und das davon abhängige Verhalten in einer großen `switch`-Anweisung (basierend auf enumerierten Konstanten) zu kodieren. Jeder Fall der `switch`-Anweisung soll in einer eigenen Klasse implementiert werden, so dass der Zustand des Objektes selbst wieder ein Objekt ist, das unabhängig von anderen Objekten ist.

Problem

Zustandsautomat

Für ein Objekt sind verschiedene Zustände, die möglichen Übergänge zwischen diesen Zuständen und das davon abhängige Verhalten zu definieren. Dies ist hier in Form eines endlichen Automaten dargestellt. Dabei zeigt der schwarze Kreis auf den Startzustand und der schwarze Kreis mit der weißen Umrandung auf den Endzustand. Die

gerichteten Kanten (Pfeile) zwischen den Zuständen `Closed`, `Open` und `Deleted` definieren den Zustandswechsel.



Hierarchischer Zustandsautomat

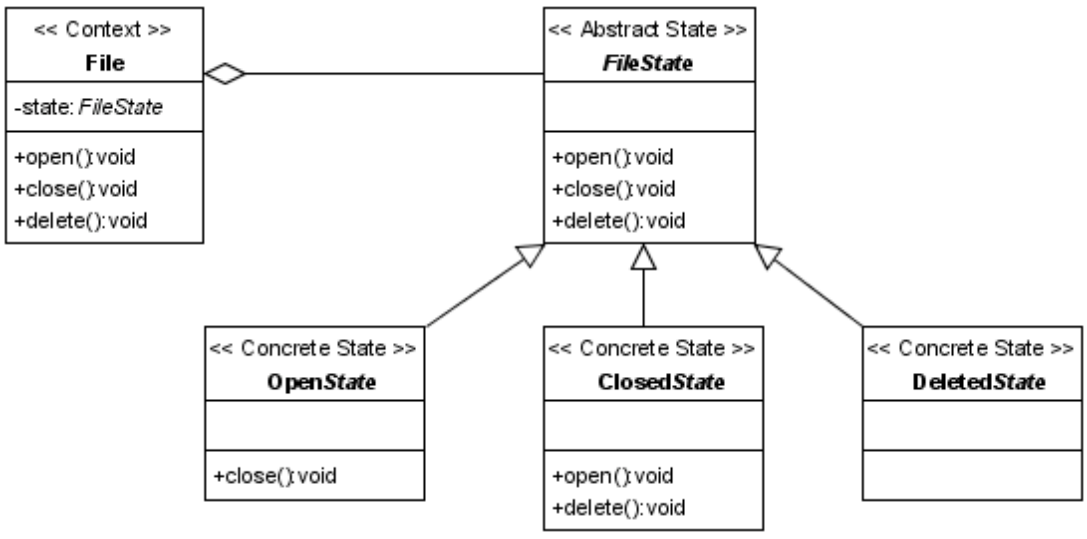
Ein einzelner Zustand eines Objektes kann wiederum in eine Anzahl verschiedener Zustände aufgeteilt werden. Den Zustand `Open` kann man beispielsweise unterteilen in *Read* und *Write*. Sie bilden einen zusammengesetzten Zustand `Open`. `Closed` sowie `Deleted` betrachtet man unabhängig vom zusammengesetzten Zustand `Open`. Diese Zustände kann man in einer Hierarchie anordnen. `Open`, `Closed` und `Deleted` sind in der ersten Ebene. In der zweiten Ebene befinden sich *Read* und *Write*, die dem Zustand `Open` zugeordnet sind.

Lösung

Einfache Zustände

Das zustandsabhängige Verhalten des Objekts wird in separate Klassen ausgelagert, wobei für jeden möglichen Zustand eine eigene Klasse eingeführt wird, die das Verhalten des Objekts in diesem Zustand definiert. Damit der Kontext die separaten Zustandsklassen einheitlich behandeln kann, wird eine gemeinsame Abstrahierung dieser Klassen definiert.

Bei einem Zustandsübergang tauscht der Kontext das von ihm verwendete Zustandsobjekt aus.



Akteure

Im Entwurfsmuster des Zustandes spielen drei Akteure eine Rolle. Der *Kontext* definiert die clientseitige Schnittstelle und verwaltet die separaten Zustandsklassen. Außerdem tauscht er diese bei einem Zustandsübergang aus.

Der *Zustand* definiert eine einheitliche Schnittstelle aller Zustandsobjekte und implementiert gegebenenfalls ein Standardverhalten. Beispielsweise kann im abstrakten Zustand die Ausführung jeglichen Verhaltens gesperrt werden. Das Verhalten kann in diesem Falle nur dann ausgeführt werden, wenn es vom konkreten Zustand durch Überschreiben der entsprechenden Methode *freigeschaltet* wurde.

Der *konkrete Zustand* wiederum implementiert das Verhalten, das mit dem Zustand des Kontextobjektes verbunden ist.

Varianten

- Für den Akteur *Zustand* kann eine Schnittstelle anstatt einer abstrakten Klasse definiert werden.
- Können mehrere Kontexte die gleichen Statusobjekte verwenden (sofern die Status durch ihre jeweiligen Klassen und nicht durch Instanzen definierbar sind bzw. Eigenschaftswerte in den Kontext ausgelagert werden können [z. B. der Dateiname]), kann Speicherplatz eingespart werden.

Vor- und Nachteile

Ein Vorteil des Systems ist, dass komplexe und schwer zu lesende Bedingungsanweisungen vermieden werden können. Außerdem können neue Zustände und neues Verhalten auf einfache Weise hinzugefügt werden. Die Wartbarkeit wird erhöht und Zustandsobjekte können wiederverwendet werden.

Auf der anderen Seite rechtfertigt der Nutzen bei sehr einfachem zustandsbehaftetem Verhalten unter Umständen nicht den teils beträchtlichen Implementierungsaufwand. Kann das Objekt sehr viele Zustände annehmen, in denen jeweils nur sehr wenige Aktionen erlaubt sind, muss dennoch jeder Zustand für jede Aktion der anderen Zustände Code enthalten, um die Schnittstelle korrekt zu implementieren, auch wenn in diesen jeweils nur eine Ausnahmebehandlung stattfindet. In einer großen Bedingungsanweisung ließe sich die Ausnahmebehandlung hingegen in einem gemeinsamen "sonst"-Zweig vereinen.

Beispiele

Prinzipiell kann jedes zustandsabhängige Verhalten durch dieses Entwurfsmuster abgebildet werden. Beispielsweise wird es für die Verwaltung von Sessions oder von Ein- und Ausgabeströmen, bei zustandbehafteten Bedienelementen einer grafischen Benutzeroberfläche oder bei Parkautomaten verwendet.

Java

Hier ist ein Beispiel für das Verhaltensmuster Zustand:

```
interface Statelike {

    void writeName(final StateContext STATE_CONTEXT, final String NAME);
}

class StateA implements Statelike {

    @Override
    public void writeName(final StateContext STATE_CONTEXT, final String NAME) {
        System.out.println(NAME.toLowerCase());
        STATE_CONTEXT.setState(new StateB());
    }
}
```

```
class StateB implements Statelike {
    /** State counter */
    private int count = 0;

    @Override
    public void writeName(final StateContext STATE_CONTEXT, final String NAME) {
        System.out.println(NAME.toUpperCase());
        // Change state after StateB's writeName() gets invoked twice
        if(++count > 1) {
            STATE_CONTEXT.setState(new StateA());
        }
    }
}
```

Die Kontextklasse hat eine Zustandsvariable, die sie hier als `StateA` in einem Anfangszustand instanziiert. In seinen Methoden verwendet sie die entsprechenden Methoden des Zustandsobjekts.

```
public class StateContext {
    private Statelike myState;

    public StateContext() {
        setState(new StateA());
    }

    public void setState(final Statelike NEW_STATE) {
        myState = NEW_STATE;
    }

    public void writeName(final String NAME) {
        myState.writeName(this, NAME);
    }
}
```

Der Test unten soll auch die Verwendung veranschaulichen:

```
public class TestClientState {
    public static void main(String[] args) {
        final StateContext SC = new StateContext();

        SC.writeName("Montag");
        SC.writeName("Dienstag");
        SC.writeName("Mittwoch");
        SC.writeName("Donnerstag");
        SC.writeName("Freitag");
        SC.writeName("Samstag");
        SC.writeName("Sonntag");
    }
}
```

Gemäß obigem Code ist die Ausgabe der `main()`-Methode von `TestClientState`:

```
montag
DIENSTAG
MITTWOCH
donnerstag
FREITAG
SAMSTAG
sonntag
```

Weblinks

Wikibooks: Zustand – Implementierung des Zustands in unterschiedlichen Sprachen

- State Design Pattern (http://www.philipphauer.de/study/se/design-pattern/state.php) – Einsteigerfreundliche Einführung
- Zustandsautomat in Java mithilfe des State Patterns (http://www.benjamin-erb.de/archives/226-Zustandsautomat-in-Java-mithilfe-des-State-Patterns.html) – Beispielimplementierung des Zustandsmusters in Java

Einzelnachweise

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*. 5. Auflage. Addison-Wesley, 1996, ISBN 3-8273-1862-9, S. 398 (englisch: *Design Patterns. Elements of Reusable Object-Oriented Software.*).