# State Pattern

## Konzept

### Problematik

Soll eine Methode einer Klasse je nach Status dieser ein anderes Verhalten aufweisen, so gibt es verschiedene Ansätze dieses Problem zu lösen.

Ein einfacher Ansatz ist es, in der Methode der Klasse ein SWITCH-CASE-Statement einzubauen, um die verschiedenen Verhalten der Methode zu repräsentieren. Dies kann beispielsweise wie folgt aussehen:

public void doSomething() {

switch(currentState) {

case 0: currentState=1;

System.out.println(„Action 1“);

break;

case 1: currentState=2;

System.out.println(„Action 2“);

break;

[..]

default: currentState=0;

System.out.println(„Invalid state“);

break;

}

}

Ein solches SWITCH-CASE-Statement hat mehrere Nachteile. So wird die Methode schwer lesbar, sobald viele Stati auftauchen. Weiter ist es auf diese Weise schwer einem Status Folgestati zuzuweisen, welche später einfach angepasst werden können. Änderungen und das Einfügen neuer Stati werden schwerer.

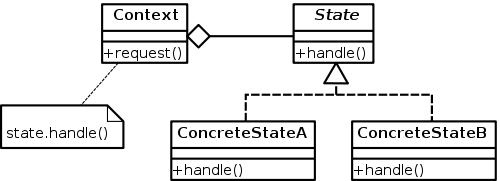
Aus diesen Gründen wird für Objekte, die je nach Status verschiedene Verhalten aufweisen sollen, das State-Pattern verwendet.

### Mechanismen des Patterns

Grundsätzlich gibt es beim State-Pattern drei Bestandteile:

* Ein **Kontext** (Context), also das Objekt, welches die Stati annehmen kann.
* Ein **Status (State) Interface**
* **Eine Klasse für jeden Status** (ConcreteState), der implementiert werden soll.

Die Zusammenhänge zwischen den Klassen ist in folgendem UML-Diagramm dargestellt:



Die Kontext-Klasse stellt gegen aussen Methoden zur Verfügung, welche sich je nach Status anders verhalten sollen. Intern wird dann der Aufruf der Methode an das konkrete Status-Objekt weitergeleitet, wo die effektive Aktion ausgeführt wird.

### Beispiel

Im folgenden konkreten Beispiel soll eine Maschine repräsentiert werden, die je nach Status („On“ oder „Off“) ein anderes Verhalten aufweist.

Das Beispiel verwenden als **Kontext (Context)** die folgende Klasse „Machine“, welche einen aktuellen Status „currentStatus“ und eine Methode „pushButton“ enthält. Weiter finden wir eine Methode „togglePower“, um den Status der Maschine zu ändern.

public class Machine {

public MachineState currentState = new OffState();

public void togglePower() {

currentState.togglePower(this);

}

public void pushButton() {

currentState.pushButton(this);

}

}

Das **Interface Status (State Interface)** wird im Beispiel repräsentiert durch das Interface „MachineState“, welches die beiden Methoden enthält, welche sich abhängig vom Status anders verhalten sollen:

public interface MachineState {

void togglePower(Machine m);

void pushButton(Machine m);

}

Als Übergabewert kann der Context angegeben werden. Dies erlaubt es der konkreten Methode, den Status des Kontext-Objektes anzupassen. So kann dann auch sichergestellt werden, das bestimmte States nur aus bestimmten anderen States heraus gesetzt werden können.

Weiter implementieren wir die beiden **konkreten Stati** „OnState“ und „OffState“:

public class OnState implements MachineState {

@Override

public void togglePower(Machine m) {

m.currentState = new OffState();

}

@Override

public void pushButton() {

System.out.println("Machine is on, do work!");

}

}

public class OffState implements MachineState {

@Override

public void togglePower(Machine m) {

m.currentState = new OnState();

}

@Override

public void pushButton() {

System.out.println("Machine is off.");

}

}

Wird in diesem Beispiel die Klasse „Machine“ instanziiert und die Methode „pushButton()“ aufgerufen, so wird „Machine is off“ ausgegeben. Wird aber erst die Methode „togglePower()“ aufgerufen, so wird „Machine is on, do work“ ausgegeben. Damit wird basierend auf dem Status der Maschine eine andere Methode ausgeführt.

## Vor- und Nachteile

Der Vorteil dieser Methode ist es, dass neue Stati sehr einfach realisiert werden können, indem eine Klasse lediglich das „State“-Interface implementiert. Die Kontext-Klasse bleibt vergleichsweise schlank. Methoden, deren Verhalten nicht vom Status abhängt können wie gewohnt implementiert werden. Weiter erlaubt es das Pattern, den Status des Kontext-Objekts aus dem Status heraus zu setzen.

Als Nachteil ist aufzuführen, dass es bei vielen möglichen Klassen entsprechend viele Status-Klassen gibt. Bereits eine einfache „State machine“ mit zwei Stati braucht 4 Klassen, um die Funktionalität mittels State Pattern zu implementieren. Die Implementation der „State machine“ ist auf verschiedene Klassen verteilt, was für den Unterhalt des Codes sicherlich nicht förderlich ist.

Weiter ist eine Erweiterung des State-Interfaces sehr aufwändig, die neue Methode muss in allen States nachgeführt werden, auch wenn diese States vielleicht nicht an dieser Methode beteiligt ist.

## Implementationsmöglichkeiten

### States als Klassen

Typischerweise werden die verschiedenen konkreten Stati als einzelne Klassen abgebildet. Dies erlaubt es, sowohl Methoden als auch Werte der Klasse je nach Status anders zu implementieren.

Das in diesem Dokument aufgeführte Beispiel verwendet diese Implementation.

Eine Schwierigkeit ist, dass sich States nicht direkt mit dem „==“-Operator vergleichen lassen, sondern der „instanceof“-Operator gebraucht werden muss. Alternativ kann eine Enumeration definiert werden, welche jeden Status enthält und welche dem konkreten State mitgegeben wird.

### States als Enumeration

Nachteilig bei der Implementierung der States als Klasse ist der Overhead wenn nur bestimmte Werte anders sind, aber das Verhalten der Methoden gleich ist.

Folgendes Beispiel zeigt die Implementation mit einer Enumeration:

<TODO>s