Proseminar Design Patterns
Prof. Dr. J. L. Keedy
Dr. G. Menger
Abt. Rechnerstrukturen

Universität Ulm
SS 2005

State Pattern/Zustandsmuster

Friedrich Hoermann

Inhaltsverzeichnis

1. Einleitung Seite 3

2. Pattern Struktur Seite 4
   2.1 Schema
   2.2 Beteiligte Klassen
   2.3 Interaktionen

3. Beispielidee Seite 5
   3.1 Idee
   3.2 Implementierung in Java
   3.3 Erweiterung des Beispiels

4. Alternative Implementierungsmöglichkeiten Seite 14
   4.1 Zustandsübergänge
   4.2 Erzeugen und Löschen von Zustandsobjekten
   4.3 Verwenden von dynamischer Vererbung

5. Allgemeine Anwendbarkeit des Patterns Seite 15
   5.1 Sinnvolle Anwendungen des Patterns
   5.2 Sinnlose Anwendungen des Patterns

6. Vorteile/Nachteile des Patterns Seite 17
   6.1 Vorteile
   6.2 Nachteile

7. Weitere Anwendungsbeispiele Seite 17

8. Literatur-Verzeichnis Seite 18

1. Einleitung

In dieser Ausarbeitung soll an einem eigens dazu entwickelten Beispiel die Funktionsweise des Zustandsmusters erläutert werden. Dabei wird auf die Vor- und Nachteile sowie die Probleme bei der Implementierung des Musters eingegangen.
Die Arbeit bezieht sich auf das Buch „Entwurfsmuster" von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides.

Das Zustandsmuster gehört in die Kategorie der Verhaltensmuster (Behaviour Patterns). Dabei ändert sich das Verhalten eines Objekts in Abhängigkeit von seinem internen Zustand. Nach außen sieht dies so aus, als ob das Objekt seine Klasse gewechselt hätte.

Nun stellt sich die Frage, ob ein Objekt nicht fast immer in Abhängigkeit von seinem Zustand reagiert. Ein einfacher Stapelspeicher (Stack) macht Beispielsweise nichts anderes: Wenn man ein Element hinzufügen will und er bereits voll ist, reagiert er anders als in halbvollem Zustand. Ist er leer und möchte man ein Element herunternehmen, so bekommt man kein Element zurückgeliefert im Gegensatz zum vollen Zustand.

Wofür soll dieses Muster also nützlich sein? Wo liegen die Unterschiede zwischen konventioneller Implementierung und der Implementierung mit dem Zustandsmuster? An welcher Stelle ist die Anwendung nicht sinnvoll? In der vorliegenden Ausarbeitung soll diesen Fragestellungen nachgegangen werden.

2. Pattern Struktur

Zu Beginn wird auf die allgemeine Pattern Struktur eingegangen und die einzelnen beteiligten Klassen und deren Aufgaben werden kurz erläutert.

2.1 Schema

Abbildung 1 zeigt die Struktur des Zustandsmusters [Gamma].

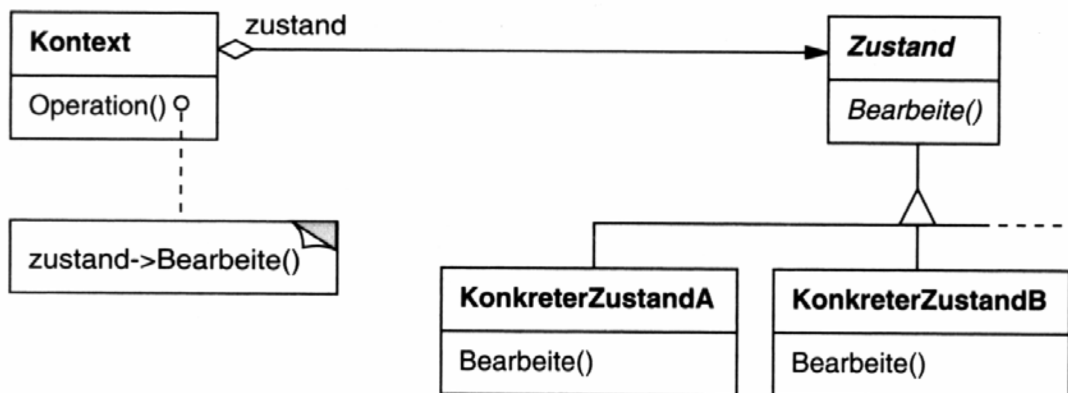

Abbildung 1

2.2 Beteiligte Klassen

Der **Kontext** stellt die Schnittstelle zum Klienten dar. Er verwaltet eine Instanz einer **KonkreterZustand**-Unterklasse, welche den aktuellen Zustand repräsentiert.
Der abstrakte ***Zustand*** stellt eine einheitliche Schnittstelle aller konkreten Zustandsobjekte dar. In ihm kann eventuell auch ein Standardverhalten implementiert werden.
In den Klassen **KonkreterZustandA,B,...** wird das konkrete Verhalten in den einzelnen Zuständen implementiert. Jede davon implementiert ein anderes Verhalten.

2.3 Interaktionen

Das Kontextobjekt leitet zustandsspezifische Anfragen an das aktuelle **KonkreterZustand**-Objekt weiter. Dabei kann das Kontextobjekt sich selbst als Argument übergeben. Dies ermöglicht dem **KonkreterZustand**-Objekt eventuell Änderungen am Kontextobjekt vorzunehmen (z.B. Änderung des aktuellen Zustands) bzw. darauf zuzugreifen. An welcher Stelle (im **Kontext** oder in der **KonkreterZustand**-Unterklasse) ein Zustandsübergang stattfindet ist nicht festgelegt. Sowohl die Kontextklasse als auch die **KonkreterZustand**-Unterklassen können entscheiden, welche Zustände nacheinander folgen und unter welchen Bedingungen sie dies tun. Welche Vor- und Nachteile die verschiedenen Möglichkeiten haben, wird in einem Beispiel im nächsten Kapitel aufgezeigt.

3. Beispiel für ein Objekt mit wechselnden Zuständen

3.1 Idee

Nun wird an Hand eines kleinen Beispiels gezeigt, was genau die Idee hinter dem Zustandspattern ist. Man stelle sich eine Tür vor, welche über Schalter gesteuert werden kann. Ein Schalter dient zum Öffnen der Tür, ein anderer um sie zu schließen. Die motorisch (elektronisch) angesteuerte Tür reagiert, je nach Zustand in dem sie sich gerade befindet, anders. Wenn sie geöffnet ist, und man betätigt den Schließen-Schalter, so schließt sie sich. Ist sie geschlossen, öffnet sie sich, wenn man den Öffnen-Schalter betätigt. Zur einfacheren Darstellung wird vorerst nur auf die Zustände Offen und Geschlossen eingegangen.
Soll diese Tür nun ohne Kenntnis des Patterns modelliert und implementiert werden, könnte man wie folgt vorgehen (siehe Abbildung 2).

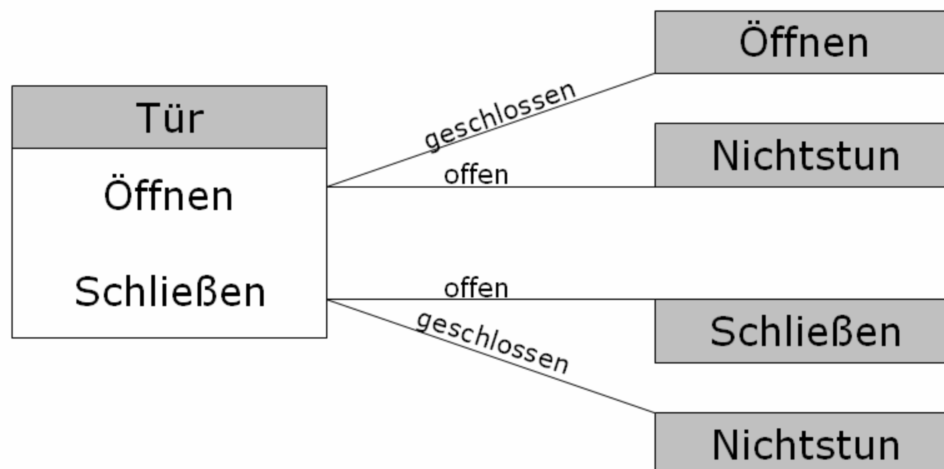

Abbildung 2

Links befindet sich eine Klasse Tür mit zwei Methoden zum Öffnen und Schließen der Tür. In diesen Methoden wird je nach Zustand in dem sich das Türobjekt gerade befindet unterschiedlich reagiert. Ruft man beispielsweise die Methode **Öffnen** auf und die Tür ist geschlossen, so wird sie geöffnet. Ist sie aber bereits geöffnet, folgt keine Aktion. Bei der Methode **Schließen** wird dies analog gehandhabt. Die Bedingungsanweisungen in den beiden Methoden können mit If-else oder Case-Switch Blöcken abgehandelt werden.
Würde es mehr als die beiden Zustände Offen und Geschlossen geben, so müsste man eine weitere Methode einführen und den neuen Zustand auch in den schon vorhandenen Methoden berücksichtigen. Hierbei kann es sehr schnell zu unübersichtlichen großen Codestücken kommen, welche insbesondere bei einer Wartung schwer zu überblicken sind. Dieses Problem soll nun mit dem State Pattern verhindert werden. Dazu wird wie folgt vorgegangen.

Die beiden Methoden **Öffnen** und **Schließen** werden beibehalten. Die abstrakte **TuerZustand**-Klasse deklariert eine Schnittstelle für die einzelnen konkreten Zustände. Unterklassen dieser **TuerZustand**-Klasse implementieren das konkrete zustandsspezifische Verhalten. Dies ergibt die in Abbildung 3 veranschaulichte Pattern Struktur.

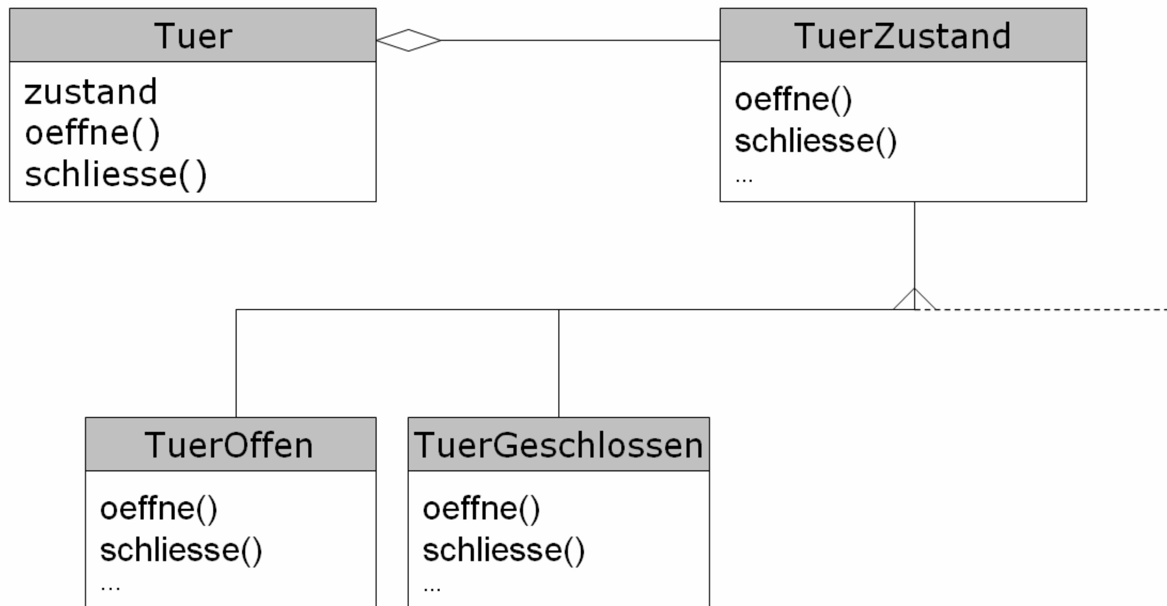

Abbildung 3

Jeder Zweig der Bedingungsanweisungen der beiden Methoden **Öffnen** und **Schließen** wird in eine eigene Klasse ausgelagert. Diese Klassen implementieren alle die gleichen Methoden **oeffne()** und **schliesse()**, in welchen wiederum die unterschiedlichen Reaktionen je nach Zustand implementiert sind. Der Klient ruft die Methoden seines Türobjekts genau gleich auf wie im Beispiel ohne Pattern. Lediglich die Implementierungen unterscheiden sich stark voneinander.
Im nächsten Abschnitt wird auf die konkrete Java-Implementierung eingegangen.

3.2 Implementierung des Beispiels in Java

Im Folgenden wird der Java-Code für das vorgestellte Beispiel betrachtet; zunächst ohne Kenntnis des Patterns.

```
public class Tuer{
     public static final int GESCHLOSSEN=1;
     public static final int OFFEN=2;

     public int zustand=GESCHLOSSEN;

     public void oeffne()
    {
          if(zustand==GESCHLOSSEN)
               zustand=OFFEN;
    }
     public void schliesse()
    {
          if(zustand==OFFEN)
               zustand=GESCHLOSSEN;
    }
}
```

Testklasse

```
public static void main(String[] args)
{
     Tuer test=new Tuer();
     System.out.println("Tuerzustand= "+test.zustand);
     test.oeffne();
     System.out.println("Tuerzustand= "+test.zustand);
     test.schliesse();
     System.out.println("Tuerzustand= "+test.zustand);
     test.schliesse();
     System.out.println("Tuerzustand= "+test.zustand);
}
```

Ausgabe der Testklasse

```
Tuerzustand= 1
Tuerzustand= 2
Tuerzustand= 1
Tuerzustand= 1
```

Beim Erzeugen eines Tuer-Objekts werden zuerst die beiden Variablen, die die einzelnen Zustände repräsentieren, initialisiert (in diesem Beispiel nur zwei, dies ist beliebig erweiterbar, dazu später mehr).
Als nächstes setzt man den Initialzustand auf Geschlossen.
Nun folgen die beiden Methoden um die Tür zu öffnen bzw. um sie zu schließen. Man sieht in jeder Methode die Bedingungsanweisungsblöcke zur Reaktion auf die unterschiedlichen Zustände des Tuer-Objekts.

Würde man nicht nur zwei Arten von Zuständen unterscheiden, so wüchsen die Bedingungsanweisungen der einzelnen Methoden stark an. Vor allem bei einer nachträglichen Änderung oder Erweiterung der Zustände könnte dies große Probleme verursachen. Mehr dazu in Kapitel 3.3.

Nach Anwendung des Patterns kommt nun folgender Code zustande (eine Möglichkeit):

Kontext

```
public class Tuer{
     public final TuerZustand GESCHLOSSEN=new
TuerGeschlossen(this);
     public final TuerZustand OFFEN=new TuerOffen(this);

     public TuerZustand zustand=GESCHLOSSEN;

     public void oeffne(){
          zustand.oeffne();
    }
    public void schliesse(){
          zustand.schliesse();
    }
}
```

Abstrakter Zustand

```
public abstract class TuerZustand{
     public Tuer tuer;
     public TuerZustand(Tuer tuer){
          this.tuer=tuer;}
    abstract void oeffne();
    abstract void schliesse();
}
```

Konkreter Zustand (Offen)

```
public class TuerOffen extends TuerZustand{
     public TuerOffen(Tuer tuer){
          super(tuer);
    }
    //Tür ist bereits offen -> nichtstun
    public void oeffne(){
    }
    //Tür ist geöffnet -> schließen
    public void schliesse(){
         tuer.zustand=tuer.GESCHLOSSEN;
    }
}
```

Konkreter Zustand (Geschlossen)
```
public class TuerGeschlossen extends TuerZustand{
    public TuerGeschlossen(Tuer tuer){
          super(tuer);
    }
    //Tür ist  geschlossen -> öffnen
     public void oeffne(){
         tuer.zustand=tuer.OFFEN;
    }



    //Tür ist bereits geschlossen -> nichtstun
    public void schliesse(){
    }
}
```

Testklasse
```
public static void main(String[] args)
{
     Tuer test=new Tuer();
     System.out.println("Tuerzustand= "+test.zustand);
     test.oeffne();
     System.out.println("Tuerzustand= "+test.zustand);
     test.schliesse();
     System.out.println("Tuerzustand= "+test.zustand);
     test.schliesse();
     System.out.println("Tuerzustand= "+test.zustand);
}
```

Ausgabe der Testklasse
```
Tuerzustand= TuerGeschlossen
Tuerzustand= TuerOffen
Tuerzustand= TuerGeschlossen
Tuerzustand= TuerGeschlossen
```

Im Kontext werden zuerst die beiden Türzustandsobjekte **GESCHLOSSEN** und **OFFEN** erzeugt.
Dann wird der Initialzustand eines Tür-Objekts festgelegt (hier: GESCHLOSSEN).
Die Methoden **oeffnen()** und **schliessen()** des Kontextobjekts rufen die **oeffnen()-** und **schliessen()-**Methoden des konkreten Zustandsobjekts, in dessen zugehörigem Zustand sich das Tür-Objekt gerade befindet, auf.

Im abstrakten Zustand **TuerZustand** werden die Methoden **oeffnen()** und **schliessen()** abstrakt implementiert. Die konkrete Implementierung muss von den konkreten Zustandsklassen übernommen werden. An dieser Stelle wäre auch die Implementierung eines Standardverhaltens, welches für alle Zustände gilt, möglich.

Beispielsweise wäre es denkbar, dass ein Licht leuchtet, egal in welchem Zustand sich die Tür befindet. Eine solche universelle Funktion könnte hier angesteuert werden.

Die Implementierung der konkreten Zustände wird hier am Beispiel der Zustände **TuerOffen** und **TuerGeschlossen** gezeigt. Die Methoden **oeffnen()** und **schliessen()** reagieren je nach Zustand unterschiedlich.

Noch ist nicht klar, wozu diese Implementierungsmöglichkeit gut sein soll. Der Code wurde umfangreicher und unübersichtlicher. Bei einer Erweiterung der möglichen Zustände wird jedoch der Vorteil deutlich.

3.3 Erweiterung des Beispiels

Nun sollen zu den beiden vorhanden Zuständen noch zwei weitere hinzugefügt werden. Bei der Tür wäre denkbar, dass sie nicht nur Offen und Geschlossen sein kann, sondern auch Zwischenzustände wie Oeffnet und Schliesst existieren. Dabei soll, wenn sich die Tür gerade schließt und der Öffnen-Schalter betätigt wird, die Tür wieder geöffnet werden. Wird der Schließen-Schalter betätigt, so soll der Zustand belassen werden. Wenn sich die Tür gerade öffnet und der Öffnen-Schalter betätigt wird, soll ebenfalls nichts geschehen. Bei Betätigung des Schließen-Schalter soll sich die Tür jedoch schließen. In der Praxis müssten die Übergänge zwischen Schliesst und Geschlossen und Öffnet und Offen automatisch geschehen, beispielsweise durch ein Feedback von der Tür. Dies wird hier jedoch nicht beachtet, da es nur um die Erweiterungsmöglichkeiten geht. Die dafür notwendigen Übergänge wurden in der Testklasse künstlich erzeugt.

Abbildung 4 zeigt das zugehörige Strukturdiagramm.

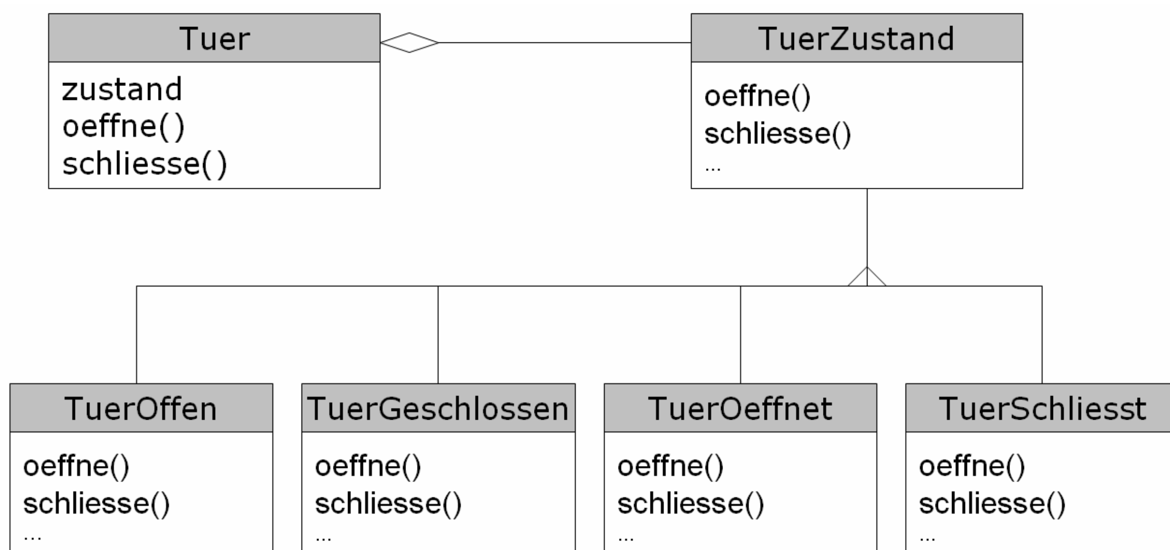

Abbildung 4

Zunächst folgt der Code ohne Pattern mit Erweiterung um die oben genannten Zustände. Änderungen wurden fett markiert:

```
public class Tuer
{
     public static final int GESCHLOSSEN=1;
     public static final int OFFEN=2;
     public static final int OEFFNET=3;
     public static final int SCHLIESST=4;

     public int zustand=GESCHLOSSEN;

     public void oeffne()
     {
          if(zustand==GESCHLOSSEN)
          zustand=OEFFNET;
          if(zustand==SCHLIESST)
          zustand=OEFFNET;
     }



     public void schliesse()
     {
          if(zustand==OFFEN)
          zustand=SCHLIESST;
          if(zustand==OEFFNET)
          zustand=SCHLIESST;
     }
}
```

Testklasse

```
public class state
 {
    public static void main(String[] args)
    {
        Tuer test=new Tuer();
        System.out.println("Tuerzustand= "+test.zustand);
        //Tür öffnen.
        test.oeffne();
        System.out.println("Tuerzustand= "+test.zustand);
        //Übergang in Zustand Offen "künstlich" erzeugt.
        test.zustand=2;
        System.out.println("Tuerzustand= "+test.zustand);
        //Tür wieder schließen
        test.schliesse();
        System.out.println("Tuerzustand= "+test.zustand);
        //Wärend des schließens wieder öffnen gedrückt
        test.oeffne();
        System.out.println("Tuerzustand= "+test.zustand);
    }}
```

Ausgabe der Testklasse

```
Tuerzustand= 1
Tuerzustand= 3
Tuerzustand= 2
Tuerzustand= 4
Tuerzustand= 3
```

Nun das erweiterte Beispiel mit dem Pattern:

Kontext

```
public class Tuer{
     public final TuerZustand GESCHLOSSEN=new
     TuerGeschlossen(this);
     public final TuerZustand OFFEN=new TuerOffen(this);
     public final TuerZustand OEFFNET=new TuerOeffnet(this);
     public final TuerZustand SCHLIESST=new
     TuerSchliesst(this);

     public TuerZustand zustand=GESCHLOSSEN;

     public void oeffne(){
          zustand.oeffne();
     }
     public void schliesse(){
          zustand.schliesse();
     }
}
```

Abstrakter Zustand (unverändert)

```
public abstract class TuerZustand{
     public Tuer tuer;
     public TuerZustand(Tuer tuer){
          this.tuer=tuer;}
    abstract void oeffne();
    abstract void schliesse();
}
```

Konkrete Zustände (Offen, Geschlossen, Oeffnet, Schliesst)

```
public class TuerOffen extends TuerZustand{
     public TuerOffen(Tuer tuer){
          super(tuer);
    }
    //Tür ist bereits offen -> nichtstun
     public void oeffne(){
    }
```

```
    //Tür ist geöffnet -> schließen
    public void schliesse(){
         tuer.zustand=tuer.SCHLIESST;
    }
}

public class TuerGeschlossen extends TuerZustand{
     public TuerGeschlossen(Tuer tuer){
          super(tuer);
     }
     //Tür ist  geschlossen -> öffnen
     public void oeffne(){
         tuer.zustand=tuer.OEFFNET;
     }
     //Tür ist bereits geschlossen -> nichtstun
     public void schliesse(){
     }
}

public class TuerOeffnet extends TuerZustand{
    public TuerOeffnet(Tuer tuer){
          super(tuer);
    }
    //Tür öffnet sich bereits -> nichtstun
     public void oeffne(){
    }

    //Tür öffnet sich gerade -> schließen
    public void schliesse(){
        tuer.zustand=tuer.SCHLIESST;
    }
}

public class TuerSchliesst extends TuerZustand{
    public TuerSchliesst(Tuer tuer){
          super(tuer);
    }
    //Tür schließt sich gerade -> öffnen
     public void oeffne(){
         tuer.zustand=tuer.OEFFNET;
    }
    //Tür schließt sich bereits -> nichtstun
    public void schliesse(){
    }
}
```

Testklasse
```
public static void main(String[] args)
{
	Tuer test=new Tuer();
	System.out.println("Tuerzustand= "+test.zustand);
	test.oeffne();
	System.out.println("Tuerzustand= "+test.zustand);
	//Übergang in Zustand Offen "künstlich" erzeugt.
	test.zustand=test.OFFEN;
	System.out.println("Tuerzustand= "+test.zustand);
	test.schliesse();
	System.out.println("Tuerzustand= "+test.zustand);
	test.schliesse();
	System.out.println("Tuerzustand= "+test.zustand);
}
```

Ausgabe der Testklasse
```
Tuerzustand= TuerGeschlossen
Tuerzustand= TuerOeffnet
Tuerzustand= TuerOffen
Tuerzustand= TuerSchliesst
Tuerzustand= TuerSchliesst
```

Nach dieser Erweiterung wird nun der Vorteil deutlich sichtbar: Bei der herkömmlichen Implementierung müssen die Methoden stark verändert und erweitert werden. Wohingegen die Änderungen bei Verwendung des Patterns minimal sind. Sie laufen hauptsächlich in den neu hinzugekommenen Klassen ab. Des Weiteren kann man nicht immer ohne Probleme Änderungen in den Methoden vornehmen, da der Quellcode nicht immer frei verfügbar ist. Bei der Lösung mit Pattern wird der Teil, in dem die Übergänge stattfinden, vom Kontext abgekapselt. Würden die einzelnen Zustandsobjekte in diesem Beispiel nicht bereits im Kontext erzeugt werden, so hätte man keine Änderungen im Kontext und dem abstrakten Zustand durchzuführen und würde nur an den konkreten Zuständen direkt Modifikationen vornehmen.

Dieses Code-Beispiel stellt nur eine Möglichkeit der Implementierung des Patterns dar, weitere werden im folgenden Abschnitt dargestellt.

4. Alternative Implementierungsmöglichkeiten

4.1 Zustandsübergänge

Die Zustandsübergänge können sowohl in den konkreten Zustandsunterklassen als auch im Kontext stattfinden. Das Pattern schreibt nicht vor, an welcher Stelle sie stattzufinden haben. Im Beispiel aus Kapitel 3 geschieht der Übergang direkt im konkreten Zustandsobjekt, was aber nicht so sein muss.

Wenn sich die Kriterien nicht ändern, können die Zustandsübergänge direkt im Kontextobjekt platziert werden. Normalerweise ist es aber besser, in den Zustandsklassen selbst den jeweiligen Nachfolgezustand sowie den Zeitpunkt, zu dem der Wechsel stattfinden soll, anzugeben, weil möglicherweise nichts im Kontext geändert werden kann (kein freier Zugang zum Quellcode). Darüber hinaus würden im Kontext alle Übergänge zusammentreffen und somit wieder unübersichtlichen Code ergeben. Wenn jedoch neue Zustände hinzugefügt werden sollen, sind bei beiden Möglichkeiten in den bereits vorhandenen Klassen Änderungen notwendig. Bei der ersten Möglichkeit ist dies aber meist einfacher zu lösen, wie im Beispiel aus Kapitel 3 ersichtlich.

4.2 Erzeugen und Löschen von Zustandsobjekten

Die einzelnen konkreten Zustandsobjekte werden entweder nur auf Bedarf erzeugt und danach gelöscht, oder sie werden im Voraus erzeugt und niemals gelöscht (siehe Türbeispiel). Die erste Möglichkeit bietet sich an, wenn nicht bekannt ist, wann die Übergänge erfolgen und in welchem Zeitabstand. Die zweite Möglichkeit sollte dann angewandt werden, wenn die Übergänge sehr oft und schnell hintereinander auftreten, was (solange die Zustandsobjekte nicht extrem groß sind) der Performance zuträglich sein.

4.3 Verwenden von dynamischer Vererbung

„Die Änderung des Verhaltens einer bestimmten Anfrage kann durch die Laufzeitänderung der Klasse eines Objekts erreicht werden." [GAMMA S.403]. Dies ist aber in den allermeisten objektorientierten Programmiersprachen nicht möglich. Als Ausnahme nennt Gamma Self und andere delegationsbasierte Sprachen.

5. Allgemeine Anwendbarkeit

5.1 Sinnvolle Anwendungen des Patterns

Das Pattern kann immer angewandt werden, wenn das Verhalten eines Objekts von seinem Zustand abhängt und es sich zur Laufzeit und in Abhängigkeit dieses Zustands ändern muss, oder wenn die Operationen der Klassen große mehrteilige Bedingungsanweisungen besitzen, die vom Objektzustand abhängen. Das Pattern lagert jeden Zweig dieser Anweisungen in eine extra Klasse aus.
Dabei muss jedoch beachtet werden, dass die konkreten Zustände fest definiert und die Übergänge zwischen ihnen eindeutig sind.

5.2 Sinnlose Anwendungen des Patterns

Es macht nicht immer Sinn das Pattern anzuwenden.
Ein Stapelspeicher beispielsweise wird weiterhin auf herkömmliche Art und Weise implementiert werden, da man hier keine fest definierbaren Zustände vorfindet. Dazu enthält der folgende Abschnitt eine kleine Überlegung zur Veranschaulichung des Problems.

Abbildung 5 zeigt die Struktur des Patterns, bei Anwendung auf einen Stapelspeicher.

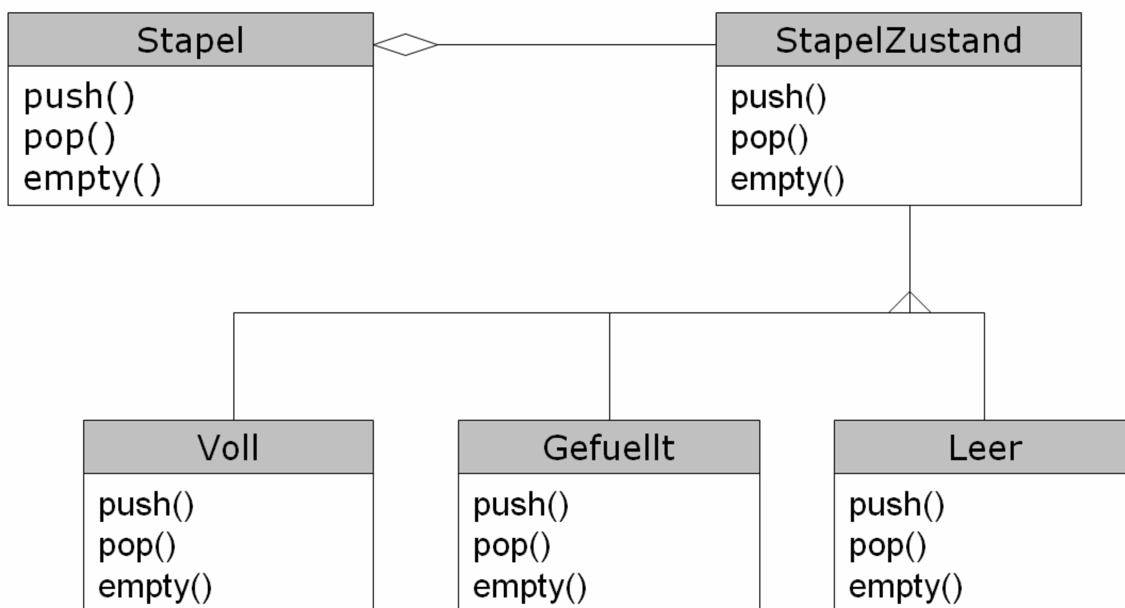

Abbildung 5

Der Zustand Gefuellt stellt einen Zwischenzustand zwischen Voll und Leer dar. Bei der Implementierung wird es in diesem Zustand zu Problemen kommen, da nicht genau definiert werden kann, wann ein Zustandsübergang erfolgt.
Erzeugt man beispielsweise ein neues Stapel-Objekt mit Platz für fünf Objekte, so befindet sich dieses im Initialzustand Leer. Nun wird mit push() etwas auf den Stapel gelegt. Der Zustand ändert sich zu Gefuellt. Wenn nun noch einmal push() angewandt wird, bleibt der Zustand jedoch in Gefuellt, da der Stapel noch nicht voll ist. Es wäre zwar möglich, dies zu implementieren, aber dazu würden sehr komplexe Bedingungsanweisungen in der Klasse Gefuellt benötigt.
Daraus wird ersichtlich, dass in Fällen, in denen keine exakt definierbaren Zustände vorhanden sind, die Anwendung des Patterns nicht sinnvoll ist.

6. Vor- und Nachteile des Patterns

6.1 Vorteile

Mit Hilfe des Patterns können komplexe Bedingungsanweisungen vermieden werden. Daraus folgt eine bessere Lesbarkeit des Codes.
Neue Zustände können mit relativ geringem Aufwand hinzugefügt werden.
Je nach Platzierung der Zustandsübergänge müssen entweder die konkreten Zustandsklassen oder die Kontextklasse geändert werden. Im obigen Beispiel ist eine Änderung der konkreten Zustände notwendig, um einen neuen Zustand einbinden zu können.
Ein weiterer Vorteil ist, dass man die einzelnen Zustandsobjekte wieder verwenden kann.
Der größte Vorteil jedoch ist die stärkere Modularisierung des Programms. Anstatt einer großen Tuer-Klasse hat man nun viele kleine (einfachere) Zustandsklassen. Dadurch wird der Code übersichtlicher und leichter verständlich für Außenstehende.

6.2 Nachteile

Ein Argument gegen das State Pattern ist sicherlich der bei einfachen Bedingungsanweisungen nicht gerechtfertigte Aufwand-/Nutzenfaktor. Da man jedoch im Normalfall kein „Einmalprogramm" schreibt, kann dies häufig nicht schon im Voraus eingeschätzt werden. Man sollte nur zur Methode ohne das Pattern greifen, wenn man genau weiß, dass der Code nie oder nur in geringem Maße erweitert werden soll.

7. Weitere Beispiele

Gamma beschreibt noch ein weiteres Beispiel zur Anwendung des Patterns, das TCP-IP Protokoll. Dazu soll man sich ein TCPVerbindungsobjekt vorstellen, welches eine Netzwerkverbindung repräsentiert. Ein solches Objekt kann sich in einem von mehreren Zuständen befinden (Etabliert, Beendet, Bereit, usw.). Erhält es von anderen Objekten eine Anfrage erhält, dann hängt sein Verhalten vom aktuellen Zustand ab.

Eine weitere gute Anwendung für das Pattern sind beispielsweise Bedienungselemente in Zeichenprogrammen. Je nach gewähltem Werkzeug reagiert die Zeichenfläche unterschiedlich. Der Editor ändert seinen Zustand. Hat man zum Beispiel eine Methode, die auf einen Mausklick reagiert, so wird diese in den unterschiedlichen konkreten Zustandsklassen unterschiedlich implementiert.
Wählt man dann ein Malwerkzeug, so ändert sich der Zustand in „Strichzeichnen" und ein Strich wird gezeichnet; bei einem anderen Werkzeug eventuell ein Rechteck aufgezogen. „Wir können das Zustandsmuster dazu verwenden, das Verhalten des Editors in Abhängigkeit vom aktiven Tool zu steuern."[GAMMA S.408]

Dazu wird eine abstrakte Klasse Tool definiert, von welcher dann Unterklassen für die Implementierung der verschiedenen Verhaltensweisen des Tools zu bilden sind. Der Zeicheneditor hat ein aktuelles Tool-Objekt und leitet an dieses seine Anfragen weiter. Wird ein neues Tool gewählt, ersetzt er dieses Objekt, was zu einem anderen Verhalten des Zeicheneditors führt [GAMMA].

8. Literatur-Verzeichnis

[GAMMA] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Entwurfsmuster, Addison-Wesley, 2004