# Analyse und Modellierung von Informationssystemen

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2014/2015

Schablonen (templates)

UML: Zustandsdiagramme

Zustandsmuster

Interpreter

# Schablonen (templates)

Eines der Grundprinzipien in der OO lautet „Programmieren gegen Interfaces“, d. h. eine Schnittstelle wird durch abstrakte Methoden beschrieben, die i. A. durch mehrere konkrete Klassen realisiert wird.

Hierbei muss jede konkrete Klasse alle abstrakten Methoden implementieren. Hierbei fällt in der Realität auf, dass häufig ein großer Teil der abstrakten Methoden identisch realisiert wird und die Variation zwischen den Implementierungen zwar wichtig, aber tatsächlich auf wenige Methoden beschränkt ist.

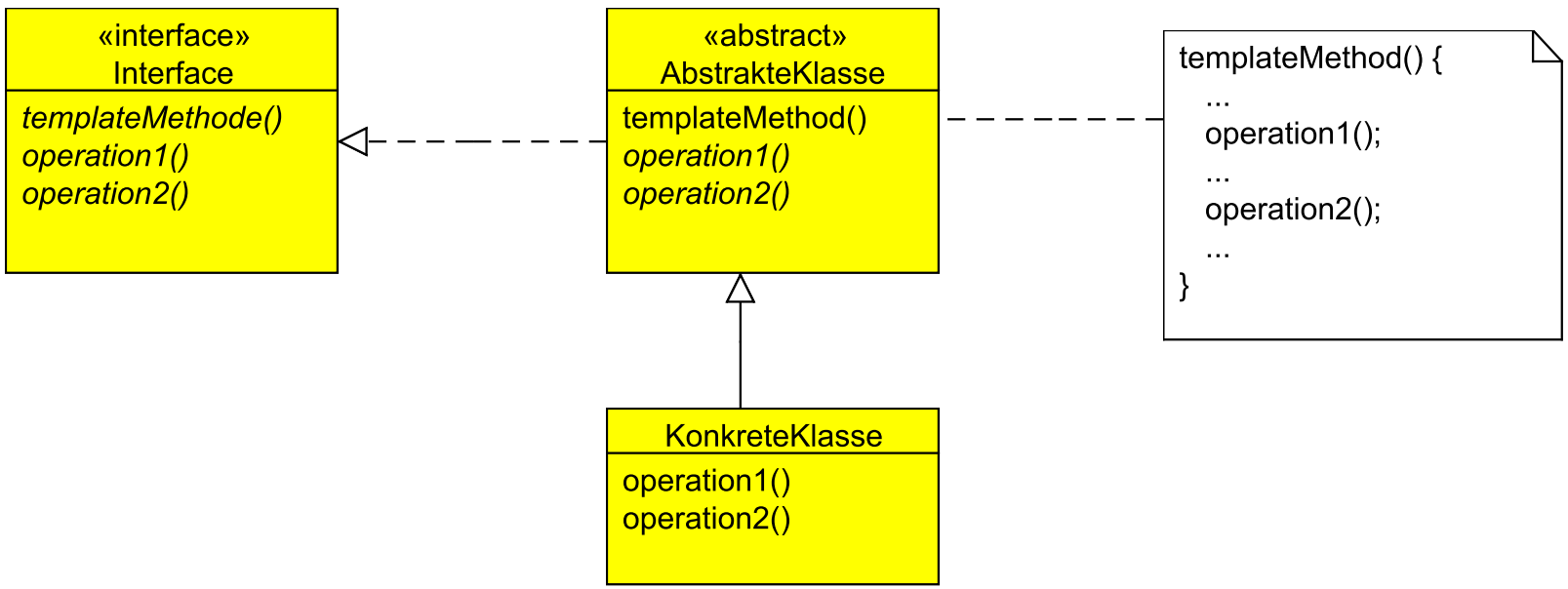
# Template-Methoden

Zur Vereinfachung kann man nun zwischen das (rein abstrakte) Interface und die konkreten Implementierungen Klassen dazwischen schalten, in denen ein Teil des Interfaces, die Template-Methoden, bereits implementiert ist (nämlich der, der den Implementierungen gemeinsam ist), während der andere Teil abstrakt bleibt und in den konkreten Klassen implementiert wird.

Alternativ können auch alle Methoden des Interfaces „blind“, also nichts tuend, implementiert sein.

Beiden Fällen ist gemeinsam, dass die Zwischenklassen sind rein dazu gedacht sind, dass die Implementierungen davon erben und die abstrakten Methoden implementieren bzw. die „blinden“ Methoden überladen.

# Klassendiagramm



# Beispiel

Ein Brettspiel läuft immer nach dem gleichen Prinzip ab, das Spielbrett mit seinen Figuren wird aufgebaut, dann ist jeder der $n$ Spieler der Reihe nach am Zug, bis einer gewonnen hat.

```
public interface Brettspiel {
    public void initialisiereBrett();
    public void naechsterZug(int spieler);
    public boolean gewonnen();
    public void spiele();
}
```

# Beispiel (Forts.)

Der Spielablauf ist also immer gleich, die Implementierung muss dann in Kindklassen erfolgen.

```
public abstract class AbstractSpiel implements Brettspiel {
    private int anzahlSpieler;

    public AbstractSpiel(int anzahlSpieler) {
        this.anzahlSpieler = anzahlSpieler;
    }

    @Override
    public void spiele() {
        initialisiereBrett();
        int spieler=1;
        while (! gewonnen()) {
            naechsterZug(spieler);
            spieler = (spieler % anzahlSpieler) + 1;
        }
    }
}
```

# Beispiel (Forts.)

Schach ist z. B. ein Brettspiel, bei dem die Spielerzahl im vorhinein mit 2 feststeht.

```
public class Schach extends AbstractSpiel {
    public Schach() {
        super(2);
    }

    @Override
    public void initialisiereBrett() {
        // TODO: Spielbrett und Figuren aufbauen
    }

    @Override
    public void naechsterZug(int spieler) {
        // TODO Spielfigur bewegen
    }

    @Override
    public boolean gewonnen() {
        // TODO Auf Matt testen
    }
}
```

# Beispiel aus der Praxis: `Action`

Das `Action`-Interface in Swing ist eine Erweiterung von `ActionListener`, die für eine Swingkomponente wie einen Button oder ein Menü-Eintrag neben der Event-Behandlung u. a. noch die Methoden `getValue(String key)` und `putValue(String key, String value)` implementiert, wobei hier insbesondere die Schlüssel „NAME“ und „ICON“ für Beschriftung und Icon sowie „ACCELERATOR_KEY“ (z. B. Strg-S bei Speicher-Aktionen) und „LONG_DESCRIPTION“ für Tooltips verwendet werden.

Tatsächlich wird in der Praxis immer von `AbstractAction` geerbt, das diese Methoden schon implementiert, so dass nur noch das eigentlich `actionPerformed` zu implementieren ist.

# Weitere Beispiele

Die Klasse `FilterReader` ist eine konkrete Klasse, die dazu gedacht ist, dass von ihr geerbt wird, wobei das „nichts“ tuende `read` überladen wird mit der gewünschten Funktionalität (z. B. Umwandlung der gelesenen Zeichen in Großbuchstaben).

`MouseListener` erklärt viele Methoden. Wenn also nur bei `mouseClicked` eine Funktion ausführen soll, müssen beim Implementieren des Interfaces die anderen Methoden leer implementiert werden. Hier ist es bequemer, von `MouseAdapter` zu erben, worin diese leere Implementierung schon erfolgt ist, und `mouseClicked` mit der gewünschten Funktionalität zu überladen.

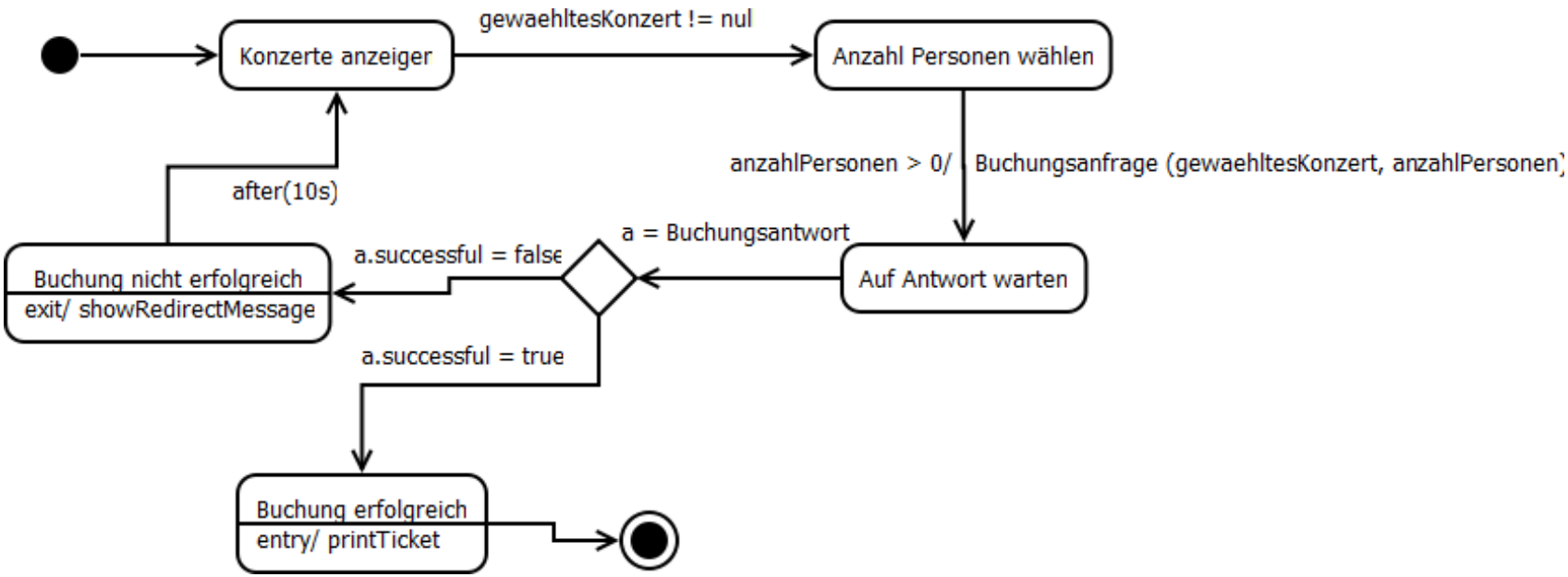
# Zustandsdiagramme in der UML

Zustandsdiagramme (*state machine diagrams*) sind Diagramme zur Spezifikation des Verhaltens von *Classifiern* (Elementen). Cassifier können sein: Klassen, Komponenten, Systeme, u. a.

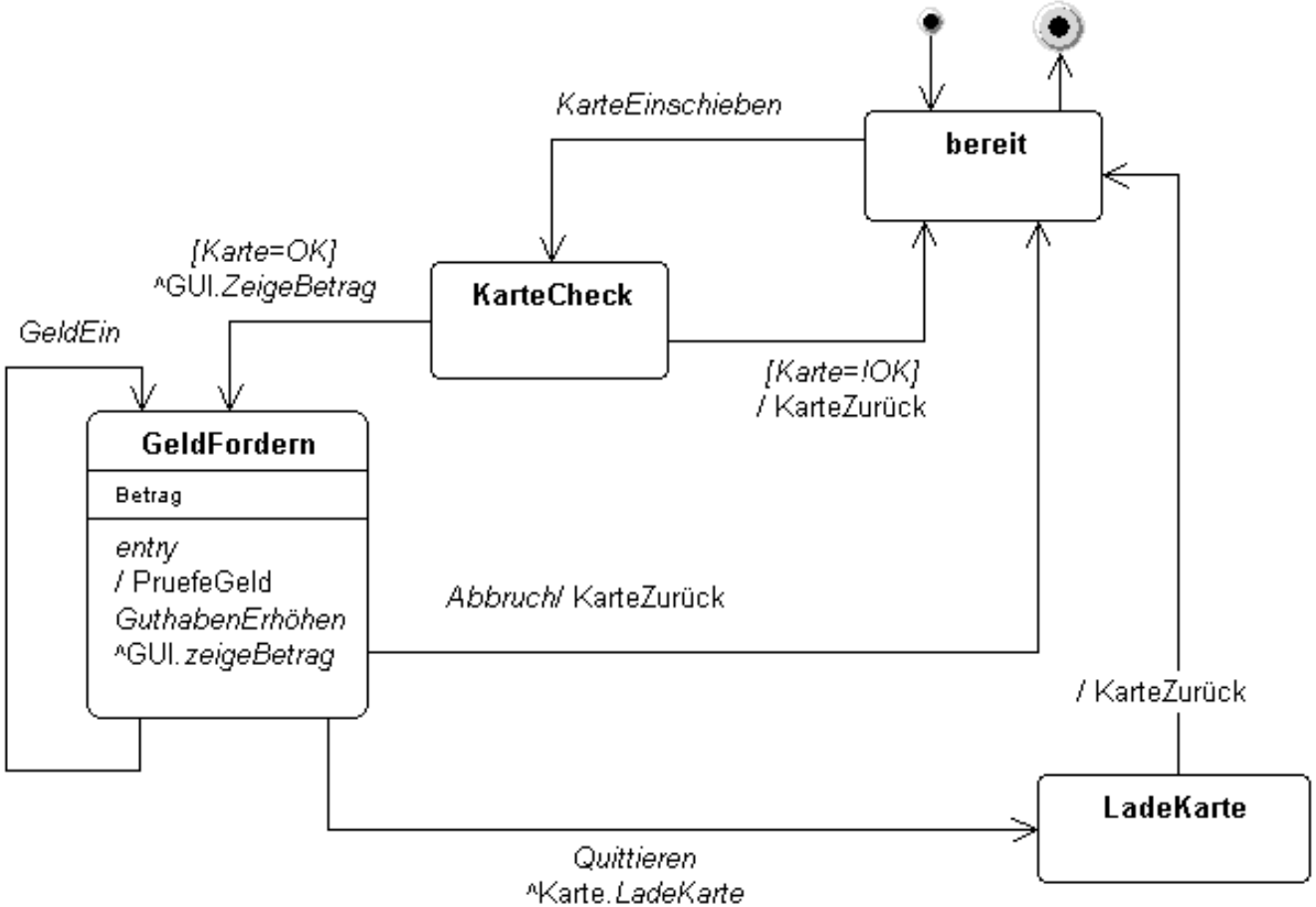
*Zustandsautomaten* beschreiben das Verhalten der Elemente während ihres Lebenszyklus durch Darstellung der möglichen Zustände und Zustandsübergänge.

Die UML 2.0 unterscheidet *behavioral state machines*, die Verhalten von Instanzen einer Klasse, Systemen oder Sysstemteilen beschreiben, und *protocol state machines*, die Protokolle, die von einem Systemelement realisiert werden, modellieren. Zustandsautomaten werden häufig zur Darstellung des Lebenszyklus von Objekten benutzt.

# Beispiel


Konzerte anzeiger
gewaehltesKonzert != nul
Anzahl Personen wählen
anzahlPersonen > 0/ Buchungsanfrage (gewaehltesKonzert, anzahlPersonen)
Auf Antwort warten
a = Buchungsantwort
a.successful = false
Buchung nicht erfolgreich
exit/ showRedirectMessage
after(10s)
a.successful = true
Buchung erfolgreich
entry/ printTicket

# Beispiel


KarteEinschieben
bereit
[Karte=OK]
^GUI.ZeigeBetrag
KarteCheck
GeldEin
[Karte=!OK]
/ KarteZurück
GeldFordern
Betrag
entry
/ PruefeGeld
GuthabenErhöhen
^GUI.zeigeBetrag
Abbruch/ KarteZurück
/ KarteZurück
LadeKarte
Quittieren
^Karte.LadeKarte

# Das Zustandsmuster (state)

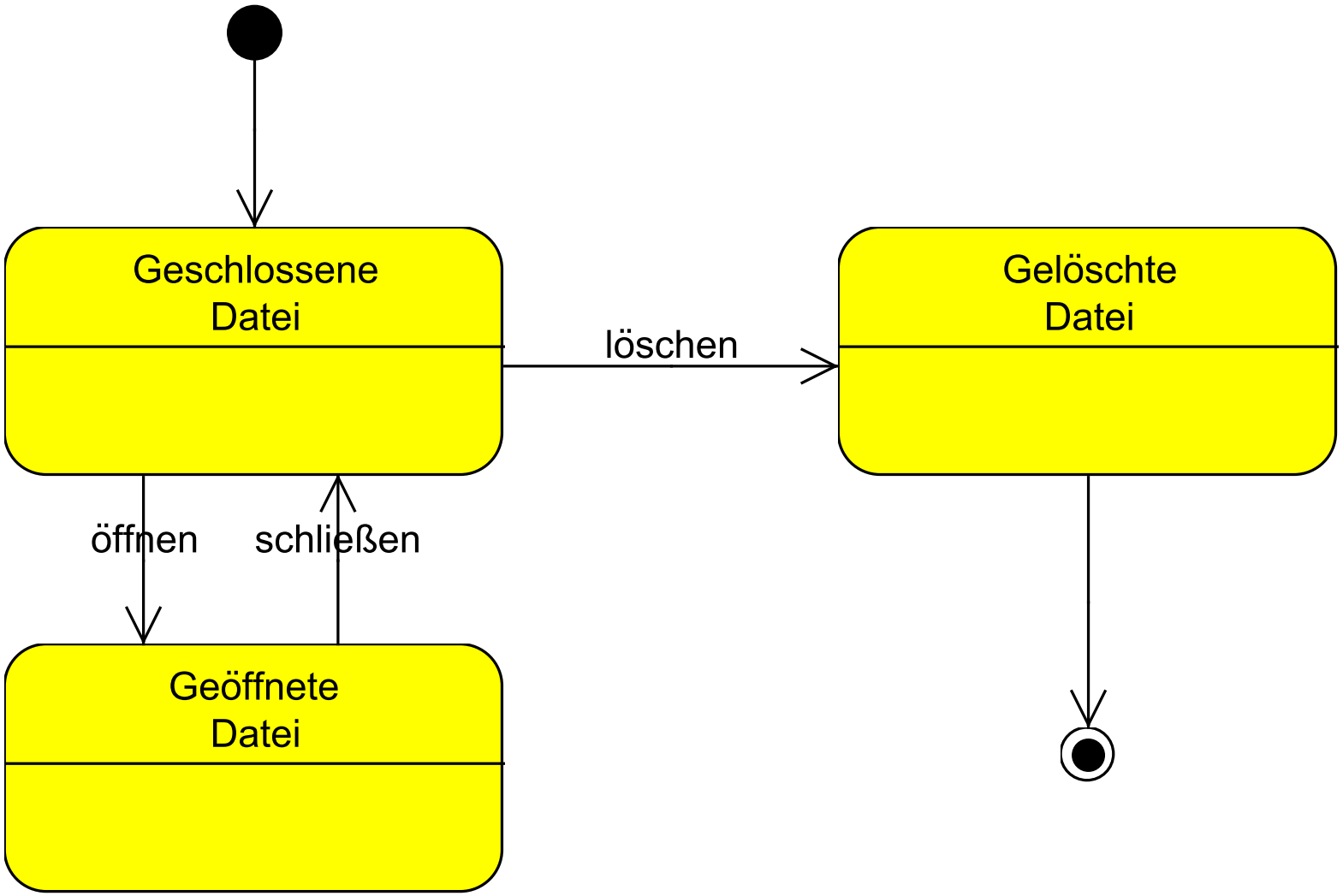
In der Praxis kommt es häufig vor, dass in einem System nur eine diskrete Menge verschiedener Zustände und nur definierte Übergänge zwischen diesen Zuständen erlaubt sind.

Hier spricht man von einer *Zustandsmaschine*.

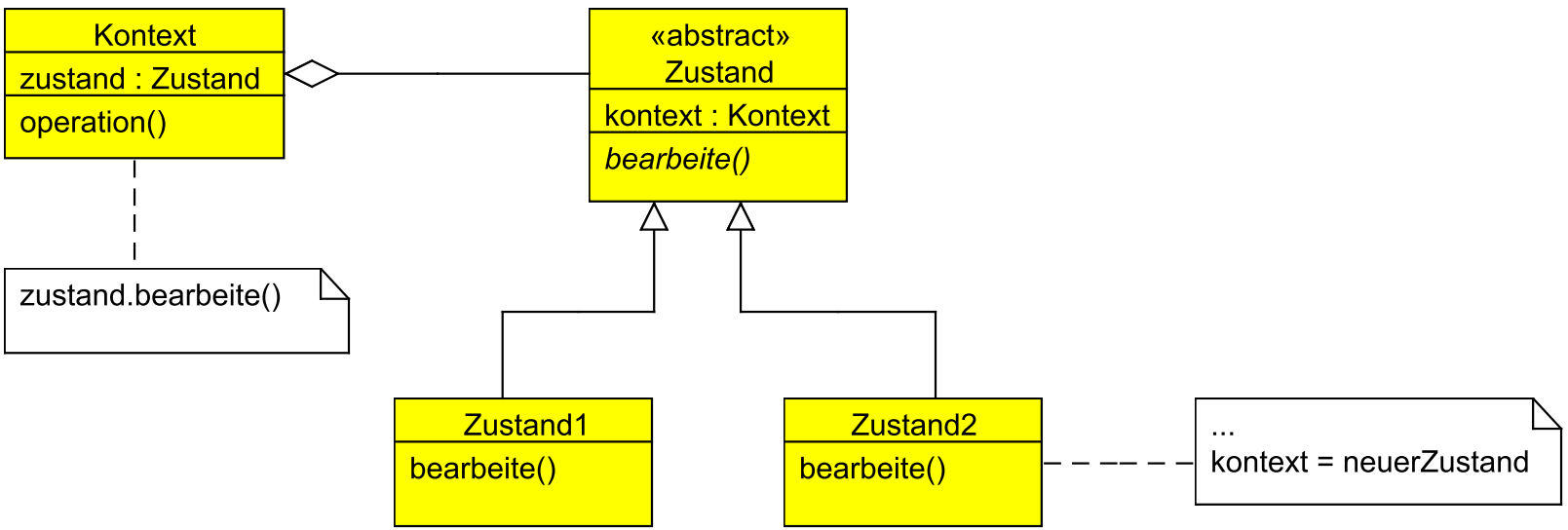
Beispiel: Für eine Datei sind die Operationen Öffnen, Schließen, Löschen erlaubt, wobei z. B. eine geöffnete Datei nicht gelöscht werden darf.

Während die je Zustand erlaubten Operationen und Übergänge auch in der Zustandsmaschine selbst implementiert werden können, sorgt das Zustandsmuster für eine Kapselung der Zustände in eigenen Klassen.

# Erlaubte Übergänge bei Dateien



# Klassendiagramm des Zustandsmusters

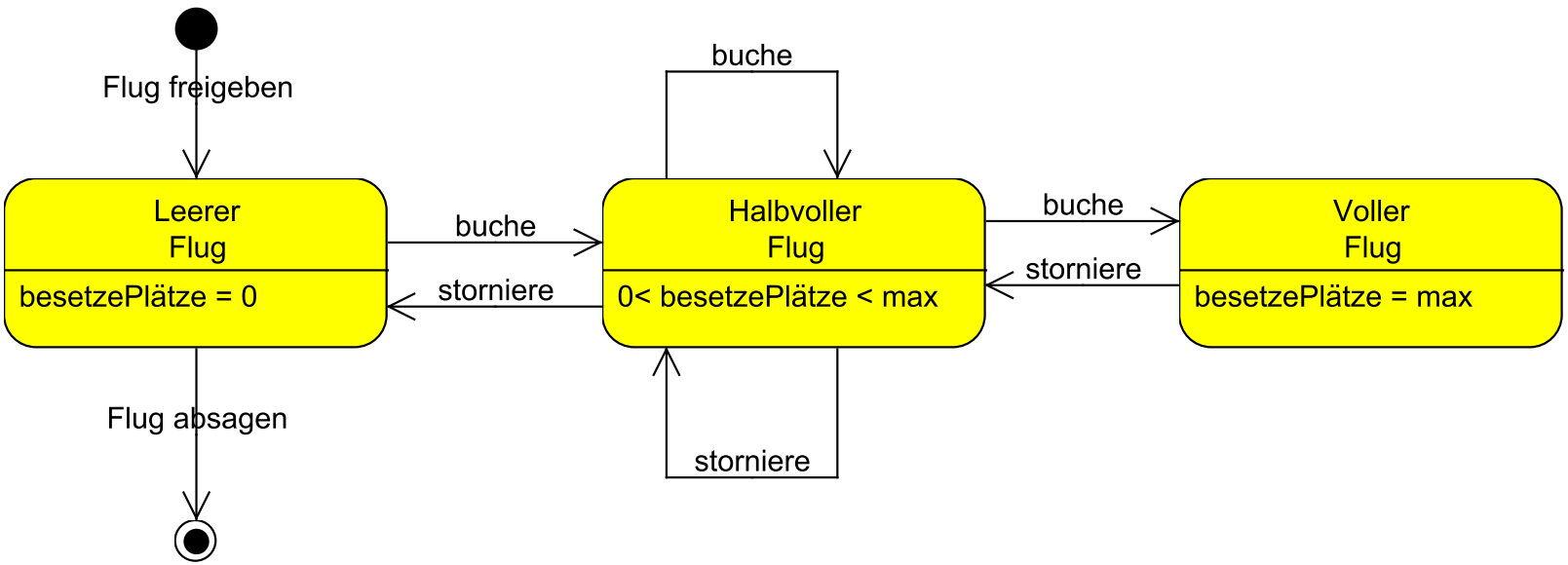


# Beispiel

Bei einem Reservierungssystem für einen Flug sind folgende Zustände erlaubt:

- Der Flug ist leer, Buchungen sind möglich, Stornierungen (mangels Buchungen) jedoch nicht,
- Der Flug ist halbvoll (was bedeutet: irgendwo zwischen leer und ausgebucht), sowohl Buchungen als auch Stornierungen sind möglich.
- Der Flug ist voll, es sind nur noch Stornierungen möglich.

# Zustandsdiagramm




Noch zu berücksichtigender Sonderfall: Flug mit Kapazität 1!

# Implementierung

```
public interface FlugInterface {
    public void buchePlatz();
    public void stornierePlatz();
}
```

Abstrakte Basis für den Zustand:

```
public abstract class FlugZustand implements FlugInterface {
    protected Flug flug;

    @Override
    public void buchePlatz() {
        throw new RuntimeException("nicht erlaubt");
    }

    @Override
    public void stornierePlatz() {
        throw new RuntimeException("nicht erlaubt");
    }
}
```

# Die Zustandsmaschine

```
public class Flug implements FlugInterface {
    FlugZustand zustand;
    int max;
    int gebucht = 0;

    public Flug(int max) {
        this.max = max;
        zustand = new LeererFlug(this);
    }

    @Override
    public void buchePlatz() {
        zustand.buchePlatz();
    }

    @Override
    public void stornierePlatz() {
        zustand.stornierePlatz();
    }
}
```

# Zustand: Leerer Flug

```
public class LeererFlug extends FlugZustand {
    public LeererFlug(Flug flug) {
        this.flug = flug;
    }

    @Override
    public void buchePlatz() {
        flug.gebucht++;
        if (flug.gebucht<flug.max) {
            flug.zustand = new HalbvollerFlug(flug);
        } else {
            flug.zustand = new VollerFlug(flug);
        }
    }
}
```

Nur Buchungen, Übergang zu halbvollem Flug (bei Kapazität 1 direkt zu vollem Flug)

# Zustand: Voller Flug

```
public class VollerFlug extends FlugZustand {
    public VollerFlug(Flug flug) {
        this.flug = flug;
    }

    @Override
    public void stornierePlatz() {
        flug.gebucht--;
        if (flug.gebucht>0) {
            flug.zustand = new HalbvollerFlug(flug);
        } else {
            flug.zustand = new LeererFlug(flug);
        }
    }
}
```

Nur Stornierungen, Übergang zu halbvollem Flug (bei Kapazität 1 direkt zu leerem Flug)

# Zustand: Halbvoller Flug

```
public class HalbvollerFlug extends FlugZustand {
    public HalbvollerFlug(Flug flug) {
        this.flug = flug;
    }

    @Override
    public void buchePlatz() {
        flug.gebucht++;
        if (flug.gebucht==flug.max) {
            flug.zustand = new VollerFlug(flug);
        }
    }

    @Override
    public void stornierePlatz() {
        flug.gebucht--;
        if (flug.gebucht==0) {
            flug.zustand = new LeererFlug(flug);
        }
    }
}
```

Buchungen und Stornierungen möglich.

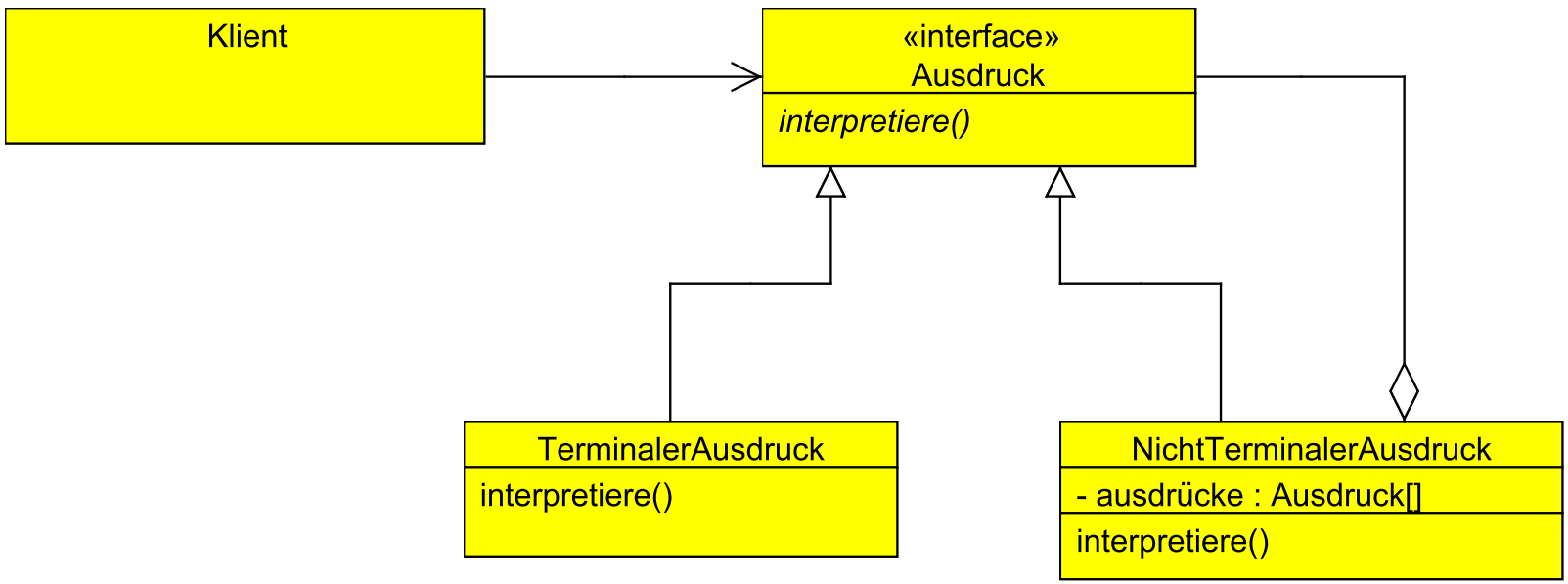
# Das Interpreter-Muster

Das Interpreter-Muster ist eine Abwandlung des Kompositums.

Analog zum Kompositum besteht der Interpreter aus einer Klassenstruktur, die einen Baum aufstellt. Im Vordergrund steht aber das Verhalten, dass im Rahmen des Musters durch einen *Parser* (der nicht Bestandteil des eigentlichen Entwurfsmusters ist) ein Ausdruck syntaktisch analysiert und ein Baum von Ausdrücken aufgebaut wird, der letztlich erlaubt, das Ergebnis des Ausdrucks zu bestimmen.

Beispiel: Ein mathematischer Ausdruck besteht aus Zahlen, unären Operatoren (z. B. Quadratfunktion) und binären Operatoren (Addition, Subtraktion).

# Schaubild


Klient
«interface»
Ausdruck
interpretiere()
TerminalerAusdruck
interpretiere()
NichtTerminalerAusdruck
- ausdrücke : Ausdruck[]
interpretiere()

# Beispiel

In der Umgekehrt Polnischen Notation als Darstellungsmethode mathematischer Formeln wird mit einem Stack gearbeitet, wobei die Operatoren hinter dem bzw. den Argument(en) stehen, auf die sie angewendet werden.

Beispiel:
`3 square 4 square plus 5 square minus`
steht für die Formel $3^2 + 4^2 - 5^2$ (und ergibt den Wert 0).

# Implementierung

Interface Expression:

```
public interface Expression {
    public double calc();
    public void visit(Deque<Expression> stack);
}
```

Das Interface besteht aus zwei Methoden, einer zum Berechnen des arithmetischen Ausdrucks (das eigentliche Interpertieren) und einer Methode, mit der Ausdrücke, die Operatoren sind, sich beim Parsen ihre Argumente vom Stack holen können (hierfür wird das Besucher-Muster benutzt).

# Implementierung (Forts.)

Klasse für Zahlen:

```
public class Number implements Expression {
    private double number;

    public Number(double number) {
        this.number = number;
    }

    @Override
    public double calc() {
        return number;
    }

    @Override
    public void visit(Deque<Expression> stack) {
        throw new RuntimeException("Nicht implementiert");
    }
}
```

# Implementierung (Forts.)

Abstrakte Basis für unäre Operatoren:

```
public abstract class UnaryOperator implements Expression {
    protected Expression operand;

    @Override
    public void visit(Deque<Expression> stack) {
        operand = stack.pop();
    }
}
```

und eine Implementierung

```
public class Square extends UnaryOperator {
    @Override
    public double calc() {
        double value = operand.calc();
        return value*value;
    }
}
```

# Implementierung (Forts.)

Abstrakte Basis für binäre Operatoren:

```
public abstract class BinaryOperator implements Expression {
    protected Expression left;
    protected Expression right;

    @Override
    public void visit(Deque<Expression> stack) {
        right = stack.pop();
        left = stack.pop();
    }
}
```

und eine Implementierung

```
public class Plus extends BinaryOperator {
    @Override
    public double calc() {
        return left.calc()+right.calc();
    }
}
```

# Der Parser

```
public class Parser {
    Deque<Expression> stack;

    public Expression parse(String line) {
        stack = new LinkedList<Expression>();
        String[] tokens = line.split(" ");
        for (String token : tokens) {
            Expression expression = null;
            try {
                // Check for floating point number
                double value = Double.valueOf(token);
                expression = new Number(value);
            } catch (NumberFormatException e) {
                // Create Expression instance for token
                expression = TokenMap.valueOf(token).getExpression();
                expression.visit(stack);
            }
            stack.push(expression);
        }
        if (stack.size()!=1) throw new RuntimeException("Stack length is r
        return stack.pop();
    }
}
```

# Operatorenfabrik

```
public enum TokenMap {
    plus(Plus.class),
    minus(Minus.class),
    square(Square.class);

    private final Class<? extends Expression> expression;
    TokenMap(Class<? extends Expression> expression) {
        this.expression = expression;
    }
    public Expression getExpression() {
        try {
            return expression.newInstance();
        } catch (InstantiationException e) {
        } catch (IllegalAccessException e) {
        }
        return null;
    }
}
```