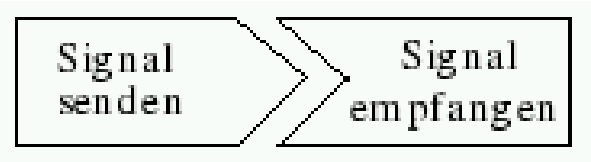
Analyse und Modellierung von Informationssystemen

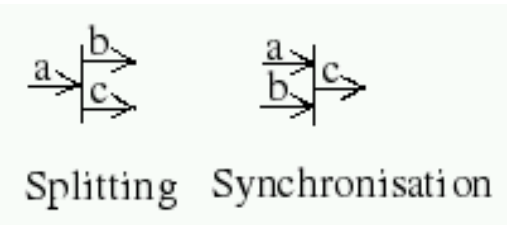
Dr. Klaus Höppner

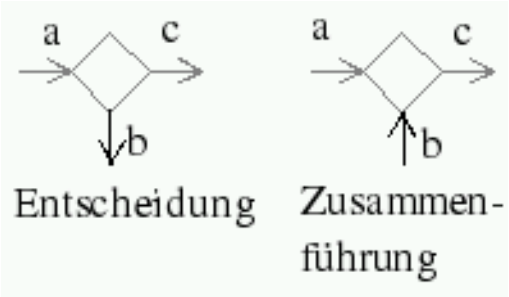
Hochschule Darmstadt – Wintersemester 2014/2015

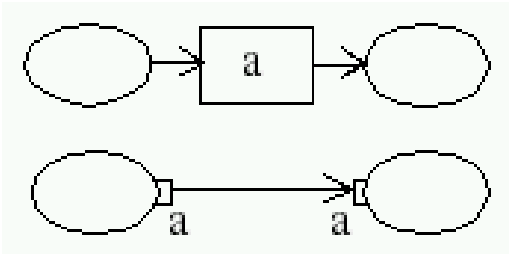
Aktivitätsdiagramme

Aktivitätsdiagramme sind Diagramme zur *Flussmodellierung* und stellen die Aktiväten eines Systems mit Aktionen, Kontroll- und Datenfluss dar.

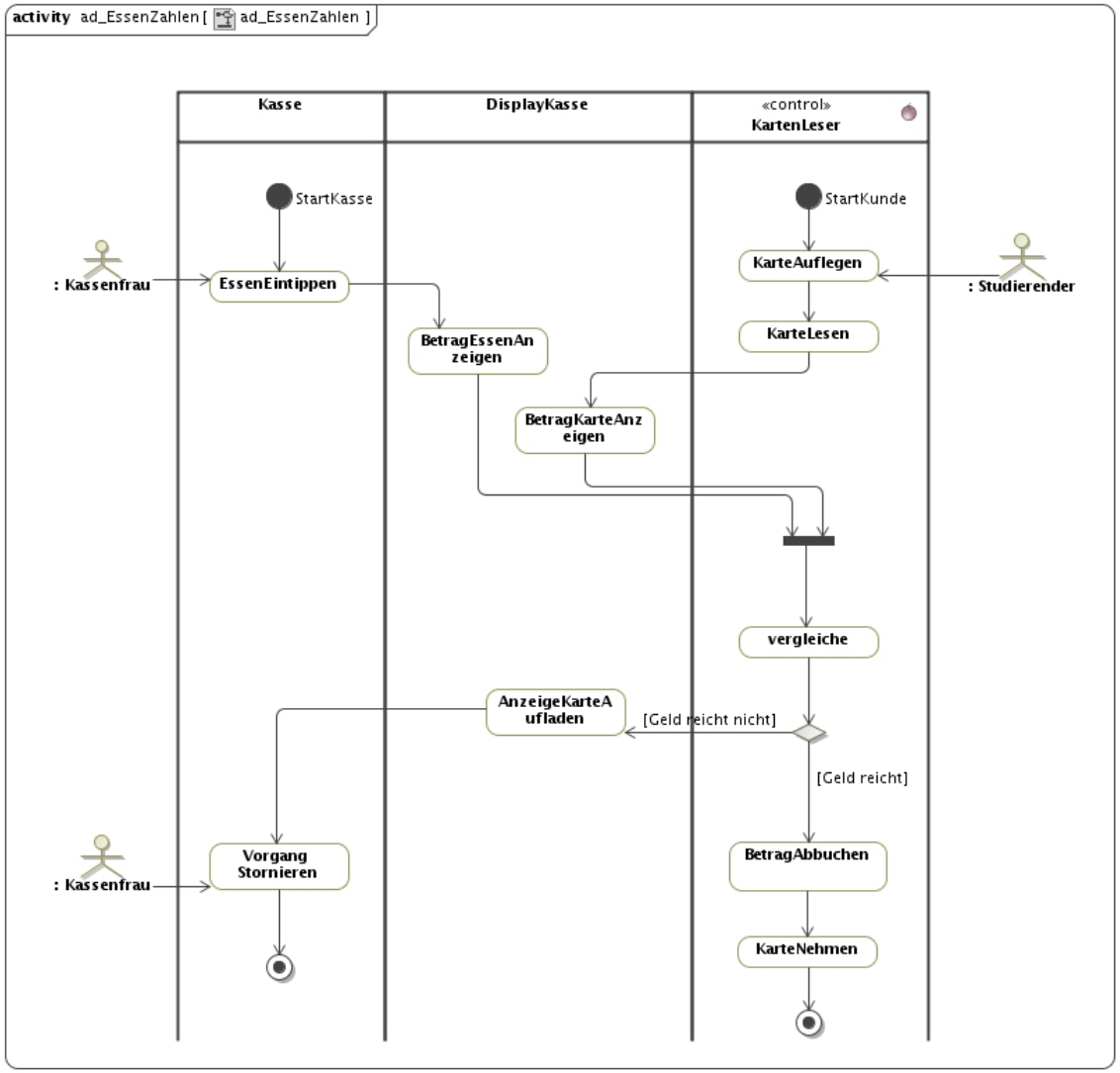




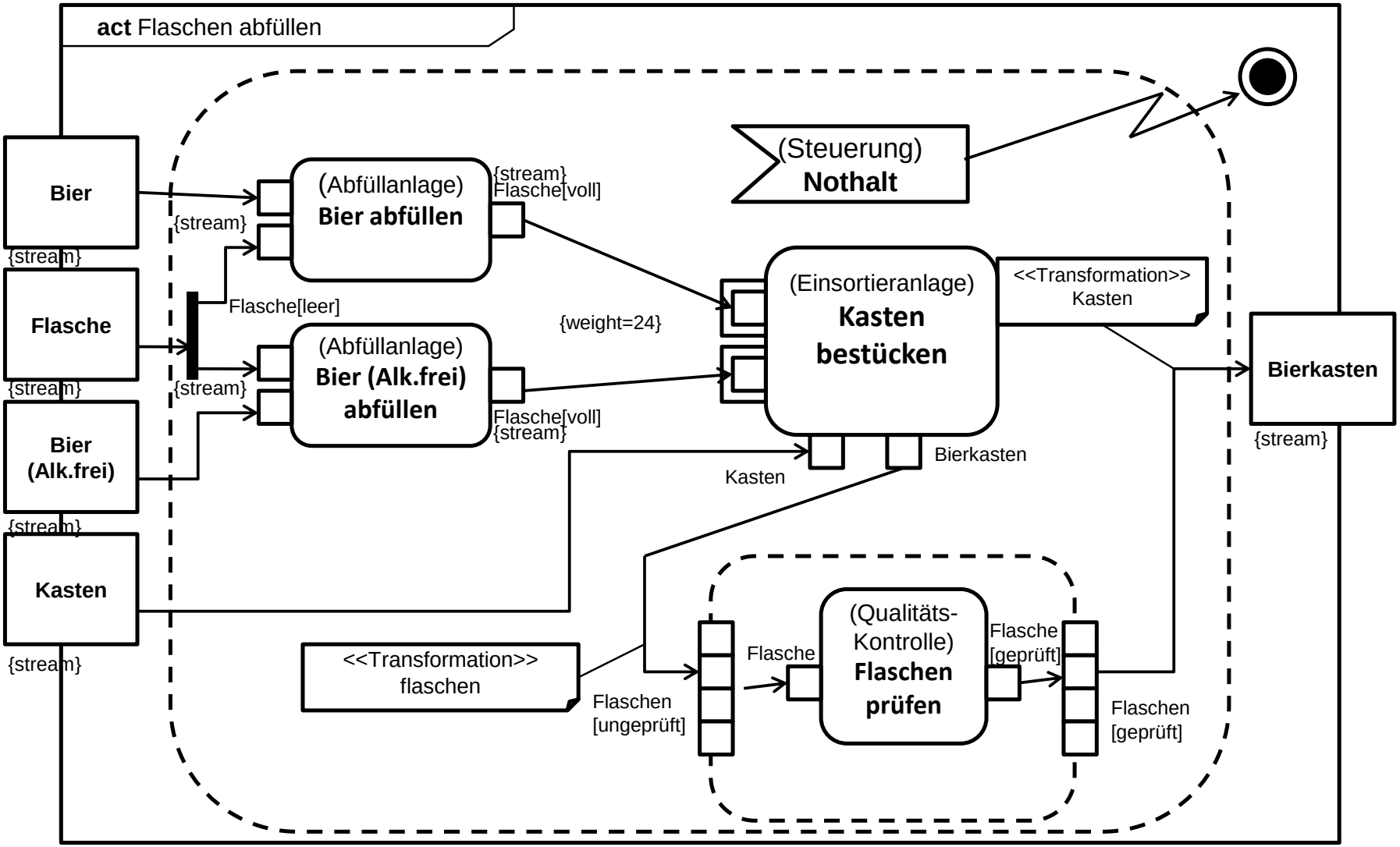




Beispiel: Zahlen in der Mensa

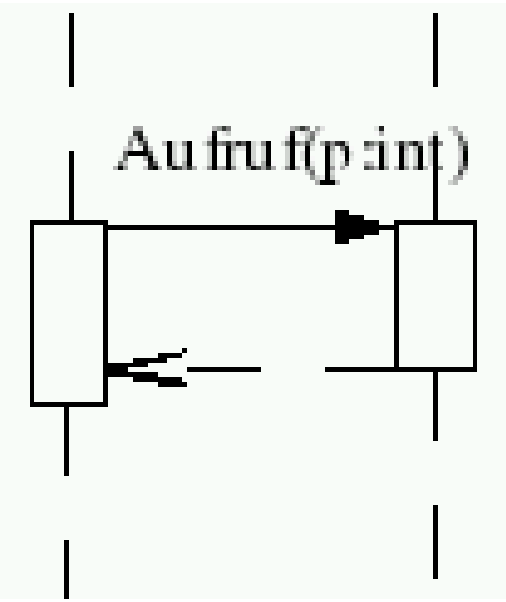


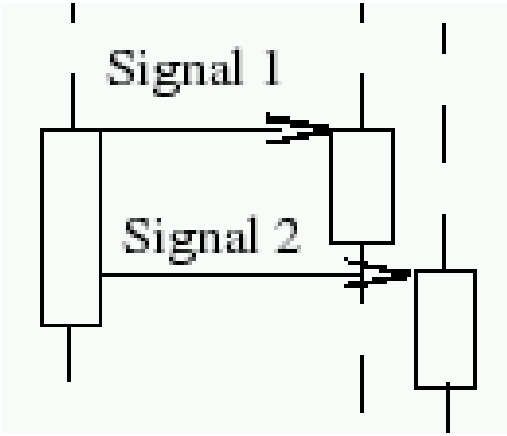
Beispiel: Abfüllen von Bierkästen

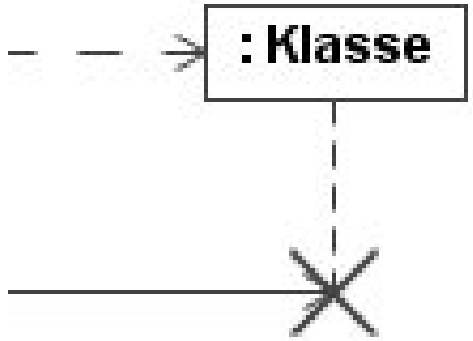


Sequenzdiagramme

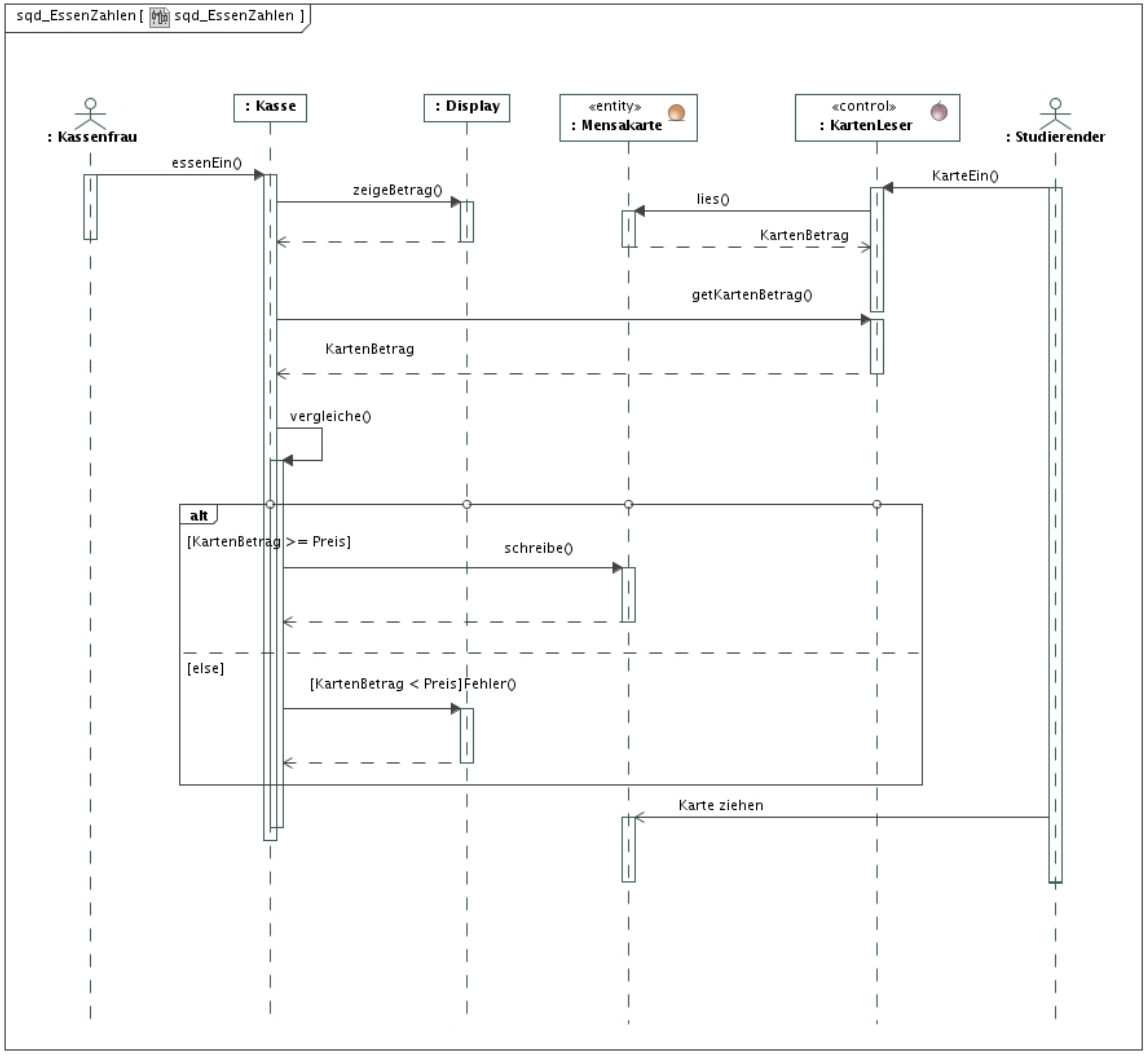
Ein Sequenzdiagramm beschreibt in Form von Lebenslinien den zeitlichen Ablauf und die Reihenfolge der Kommunikation von den an einer Szene beteiligten Objekten.
Besonders wichtig sind hierbei die Darstellung synchroner und asynchroner Aufrufe und der Erzeugung und Zerstörung von Objekten.







Beispiel: Zahlen in der Mensa



Überblick der Verhaltensmuster

Verhaltensmuster befassen sich mit der Zuweisung von Zuständigkeiten an Objekte, wobei hier nicht nur das strukturelle Muster interessiert, sondern die Interaktion zwischen den Objekten im Vordergrund steht.

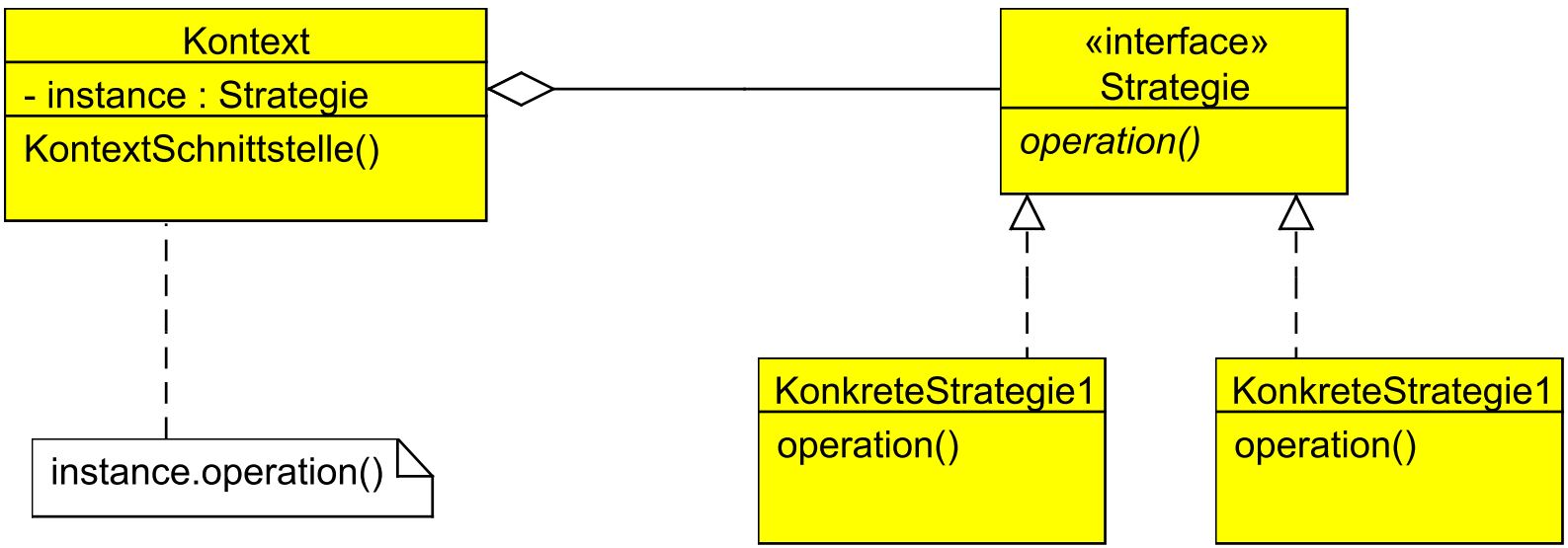
Hierbei finden wir das schon bei den Strukturmustern angewendete Prinzip der Komposition anstelle Vererbrung wieder. Einige der Verhaltensmuster vollenden das objektorientierte Prinzip der Kapselung:
Kapselung von Teil-Algorithmen in einer eigenen Klasse (*Strategiemuster*), Kapselung von Befehlen in einer Klasse (*Befehlsmuster*), Kapselung von Zuständen in einer Klasse (*Zustandsmuster*).

Strategiemuster

Das Strategiemuster (*Strategy*, auch bekannt als *Policy*) definiert eine Familie von Algorithmen, die in einer Klassenhierarchie gekapselt und untereinander austauschbar gemacht wird.

Beispiel: Ein `JPanel` in Swing kann verschiedene Layouts haben, die mit Hilfe der Methode `setLayout(LayoutManager)` ausgewählt werden können.

Schaubild


Kontext
- instance : Strategie
KontextSchnittstelle()
«interface»
Strategie
operation()
instance.operation()
KonkreteStrategie1
operation()
KonkreteStrategie1
operation()

Beispiel

Als Beispiel betrachten wir, dem Buch *Head First Design Patterns* folgend, ein Klassenmodell von Enten, die fliegen und quaken können.

Fliegen und quaken können dabei auf unterschiedliche Arten implementiert sein. Statt nun für jede Entenart eine Klasse zu schreiben, die die Implementierung beider Operationen direkt enthält (was dazu führen würde, dass für Entenarten, die gleich fliegen, aber unterschiedlich quaken, der Code für Fliegen dupliziert würde), wird eine Komposition gewählt, bei der die Flug- bzw. Quakoperation an ein Objekt delegiert wird, das die jeweilige Operation implementiert.

Implementierung des Beispiels

```
public abstract class Duck {
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public void setFlyBehavior (FlyBehavior fb) {
        flyBehavior = fb;
    }

    public void setQuackBehavior(QuackBehavior qb) {
        quackBehavior = qb;
    }

    abstract void display();

    public void performFly() {
        flyBehavior.fly();
    }

    public void performQuack() {
        quackBehavior.quack();
    }
}
```

Interfaces mit je einer Implementierung

```
public interface FlyBehavior {
    public void fly();
}

public class FlyWithWings implements FlyBehavior {
    public void fly() {
        System.out.println("I'm flying!!");
    }
}

public interface QuackBehavior {
    public void quack();
}

public class Quack implements QuackBehavior {
    public void quack() {
        System.out.println("Quack");
    }
}
```

Eine konkrete Ente

```
public class MallardDuck extends Duck {
    public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }

    public void display() {
        System.out.println("I'm a real Mallard duck");
    }
}
```

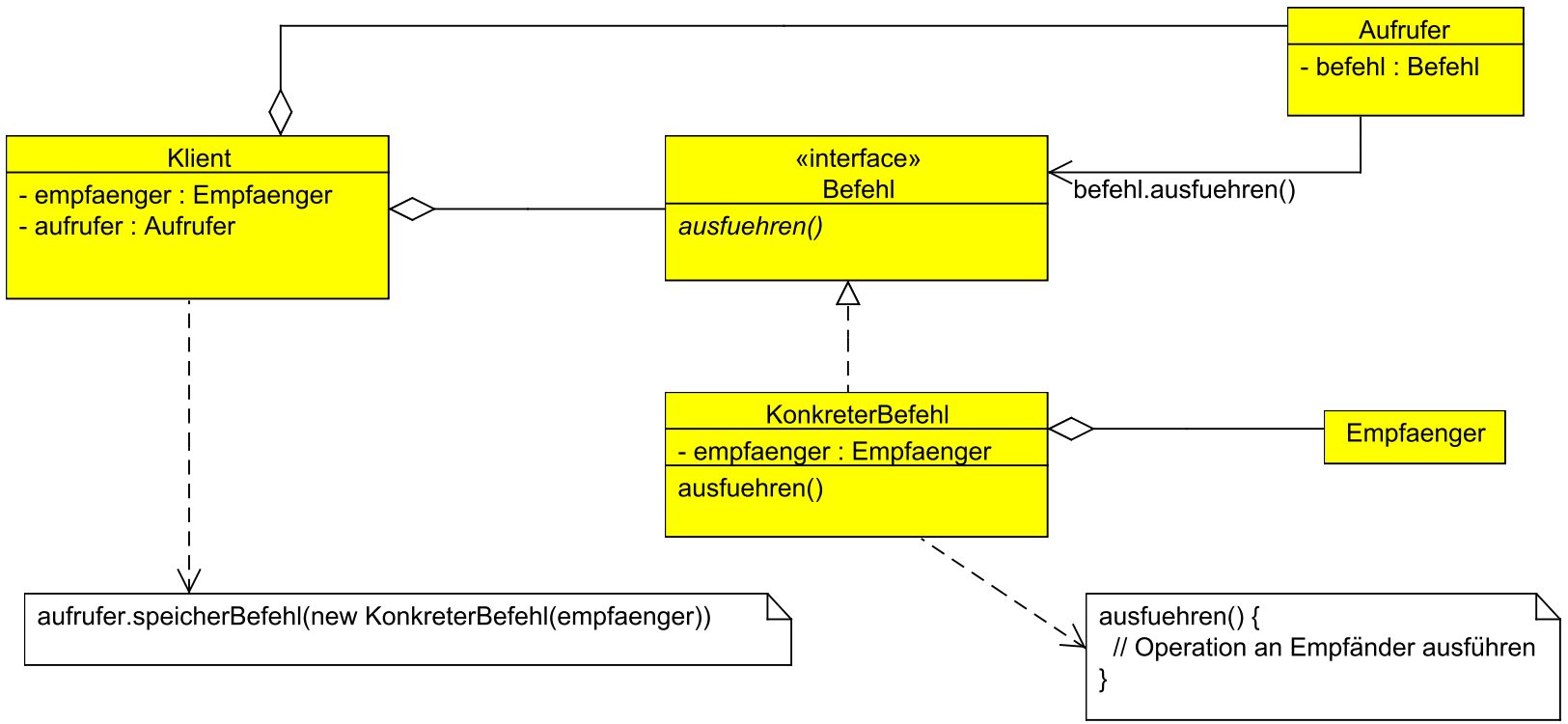
Die eigentlichen Flug- und Quackalgorithmen sind nun in eigenen Klassen gekapselt, und die Implementierung aus der eigentlichen Entenklasse ausgelagert. Als Nebeneffekt können über die Methoden `setXYZBehavior` die Algorithmen zur Laufzeit dynamisch gewechselt werden.

Das Befehlsmuster

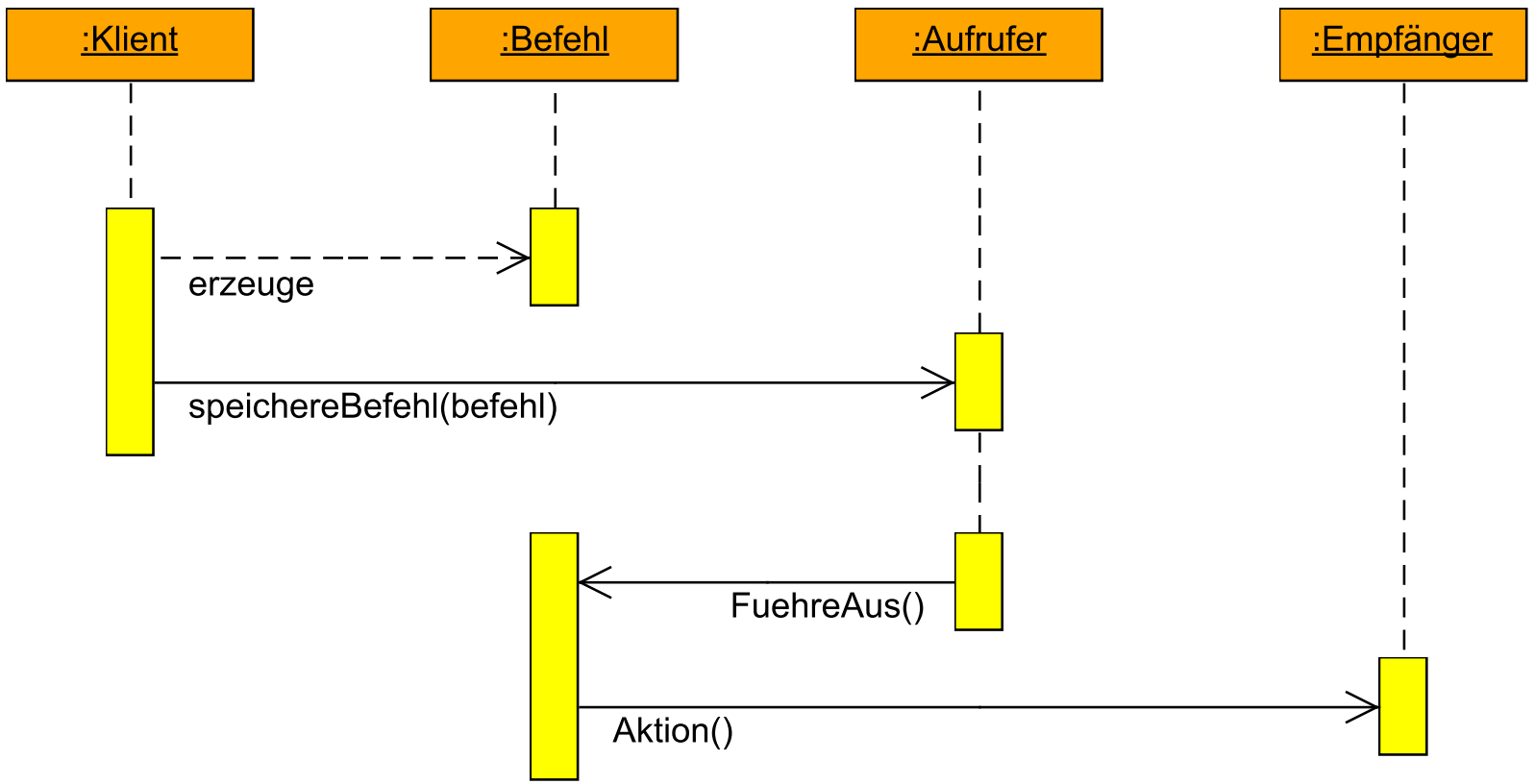
Das Befehlsmuster (*command* pattern) dient beim Ausführen von Befehlen der Entkopplung zwischen dem Aufrufer und dem Empfänger, auf den sich der Befehl bezieht. Hierfür wird die eigentliche Befehlsausführung ein einem Objekt gekapselt. Praktisch ist es hierdurch auch möglich, Befehle in einer Sequenz als Makro ausführen zu lassen und eine Redo-Undo-Historie von Befehlen zu bilden.

Beispiel: In Swing kann ein `JButton` sich auf eine `Action` beziehen, die neben der Button-Beschriftung auch einen ActionListener darstellt. Beim Klick auf den Button wird die Action ausgeführt, die sich dann i. A. auf andere Komponenten der GUI auswirkt.

Schaubild


Aufrufer
- befehl : Befehl
Klient
- empfaenger : Empfaenger
- aufrufer : Aufrufer
«interface»
Befehl
ausfuehren()
befehl.ausfuehren()
KonkreterBefehl
- empfaenger : Empfaenger
ausfuehren()
Empfaenger
aufrufer.speicherBefehl(new KonkreterBefehl(empfaenger))
ausfuehren() {
// Operation an Empfänder ausführen
}

Sequenzdiagramm



Beispiel

Die Technik eines Hauses besitzt Lampen, die ein oder aus sein können:

```
public class Licht {
    private boolean ein = false;

    public boolean isEin() {
        return ein;
    }

    public void setEin(boolean ein) {
        this.ein = ein;
    }
}
```

Beispiel (Forts.)

Hierfür wird nun eine Befehlsinterface definiert:

```
public interface Befehl {
    public void ausfuehren();
}
```

mit der Implementierung

```
public class LichtEinBefehl implements Befehl {
    private Licht licht;

    public LichtEinBefehl(Licht licht) {
        this.licht = licht;
    }

    @Override
    public void ausfuehren() {
        licht.setEin(true);
    }
}
```

und dem entgegengesetzten `LichtAusBefehl`

Beispiel (Forts.)

Nun wird noch ein Wartebefehl definiert

```
public class WarteBefehl implements Befehl {
    int millis;

    public WarteBefehl(int millis) {
        this.millis = millis;
    }

    @Override
    public void ausfuehren() {
        try {
            Thread.sleep(millis);
        } catch (InterruptedException e) {}
    }
}
```

Beispiel (Forts.)

Und eine Klasse für Befehlssequenzen:

```
public class Makro implements Befehl {
    List<Befehl> befehle = new ArrayList<Befehl>();

    public void addBefehl(Befehl befehl) {
        befehle.add(befehl);
    }

    @Override
    public void ausfuehren() {
        for (Befehl b : befehle) {
            b.ausfuehren();
        }
    }
}
```

Beispiel (Forts.)

Nun wird das Haus definiert

```
public class Haus {
    private Licht flurLicht = new Licht();
    private Befehl flurEin = new LichtEinBefehl(flurLicht);
    private Befehl flurAus = new LichtAusBefehl(flurLicht);

    public Befehl getFlurEin() {
        return flurEin;
    }
    public Befehl getFlurAus() {
        return flurAus;
    }
}
```

Anwendung

Hier wird eine Befehlssequenz angelegt, die 5 Sekunden lang das Flurlicht einschaltet (diese Sequenz könnten dann bei einem Tastschalter als Aufrufer hinterlegt werden)

```
Haus haus = new Haus();
Makro sequenz = new Makro();
sequenz.addBefehl(haus.getFlurEin());
sequenz.addBefehl(new WarteBefehl(5000));
sequenz.addBefehl(haus.getFlurAus());
sequenz.ausfuehren();
```

Realeres Beispiel

Erkennen Sie hier das Befehls-Muster?

```
public class MyFrame extends JFrame {
    int count = 0;
    JLabel label = new JLabel("Bisher 0 mal geklickt");

    public MyFrame() {
        super("Fenster");
        setLayout(new BoxLayout(this.getContentPane(), BoxLayout.Y_AXIS));
        add(label);
        Action a = new AbstractAction("Klick") {
            @Override
            public void actionPerformed(ActionEvent e) {
                label.setText(String.format("Bisher %d mal geklickt",
                                                    ++count));
            }
        };
        add(new JButton(a));
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }
}
```