RUHR-UNIVERSITÄT BOCHUM
Lehrstuhl für Informatik im Bauwesen

RUB

Programmierung und Programmiersprachen

Sommersemester 2023

Strategie-Muster

Entwurfsmuster – Strategie-Muster

Erweitern von Funktionalität

Warum ist dieser Code schlecht?

```
class Ente {
    String typ;
    public void anzeigen() {
        if(this.typ.equals("StockEnte")) {
            // Zeichne eine Stockente
        } else if(this.typ.equals("MoorEnte")) {
            // Zeichne eine Moorente
        }
    }
    public void quaken() {
        if(this.typ.equals("StockEnte")) {
            // Quake wie eine Stockente
        } else if(this.typ.equals("MoorEnte")) {
            // Quake wie eine Moorente
        }
    }
}
```

Entwurfsmuster – Strategie-Muster

Erweitern von Funktionalität

Warum ist dieser Code schlecht?

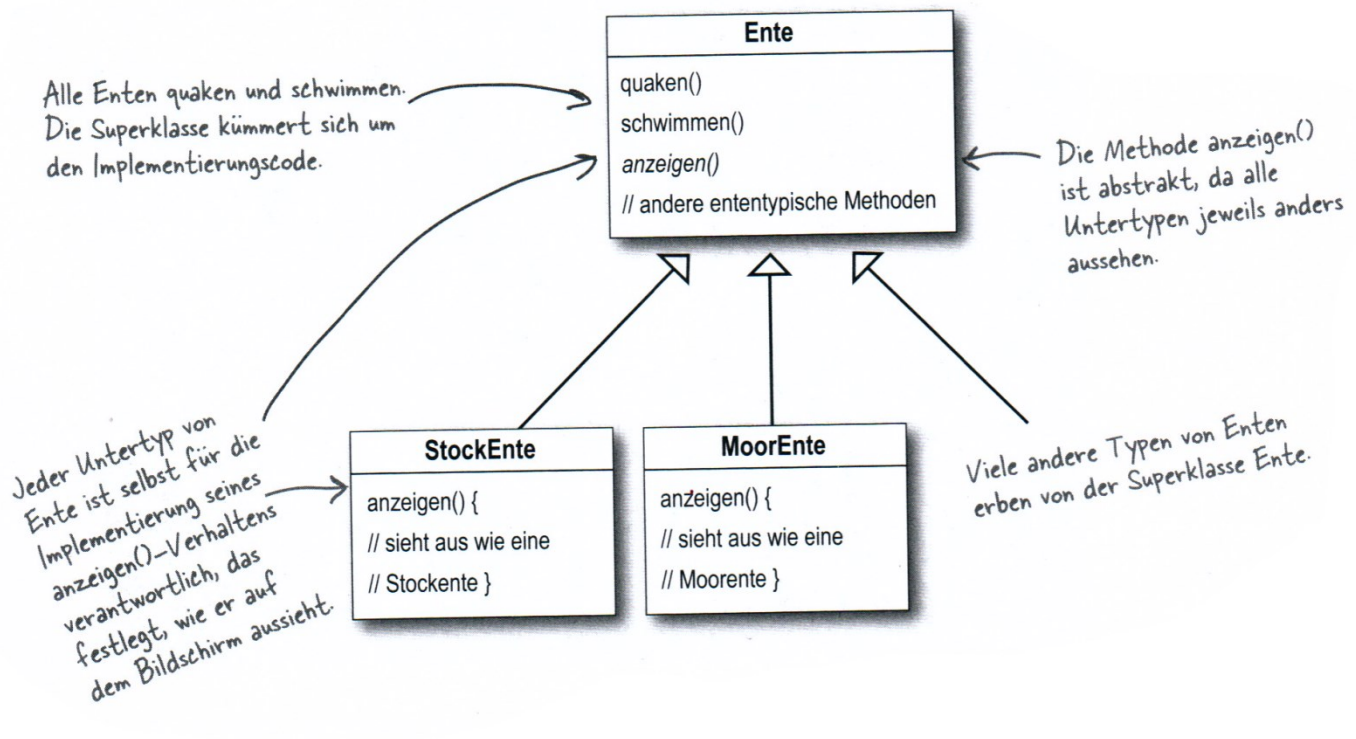
```
class Ente {
    String typ;
    public void anzeigen() {
        if(this.typ.equals("StockEnte")) {
            // Zeichne eine Stockente
        } else if(this.typ.equals("MoorEnte")) {
            // Zeichne eine Moorente
        }
    }
    public void quaken() {
        if(this.typ.equals("StockEnte")) {
            // Quake wie eine Stockente
        } else if(this.typ.equals("MoorEnte")) {
            // Quake wie eine Moorente
        }
    }
}
```

- Code-Duplikation
  - ➢ Fehleranfällig bei Änderungen
- Neue Typen hinzufügen ist unübersichtlich und benötigt Zugriff auf die Klasse
  - ➢ Erweiterung erschwert
- Unklar welche Typen unterstützt werden
  - ➢ Undurchsichtige Nutzung

Entwurfsmuster – Strategie-Muster

Erweiterung durch Vererbung

Naiver Ansatz: Abstraktion



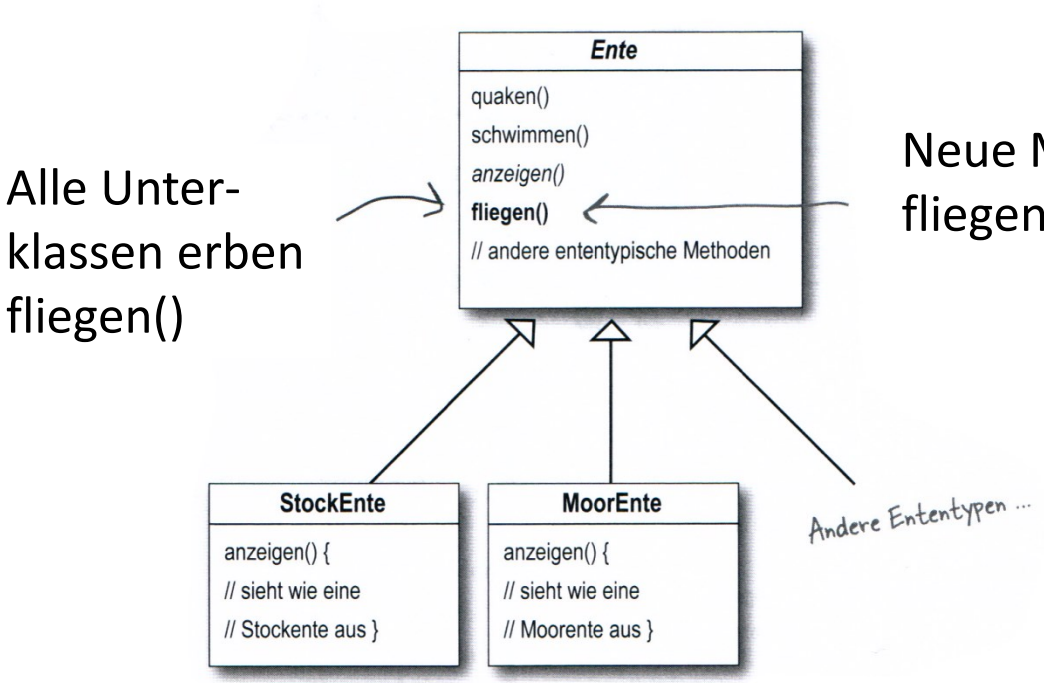
Freeman et al. (2005) Entwurfsmuster von Kopf bis Fuß. O'Reilly

Entwurfsmuster – Strategie-Muster

Erweiterung durch Vererbung

Vorteile der Vererbung sind, dass wenn eine neue Methode für alle Klassen hinzugefügt werden soll, kann dies in der Oberklasse erfolgen

Es soll eine Methode fliegen() hinzugefügt werden



Freeman et al. (2005) Entwurfsmuster von Kopf bis Fuß. O'Reilly

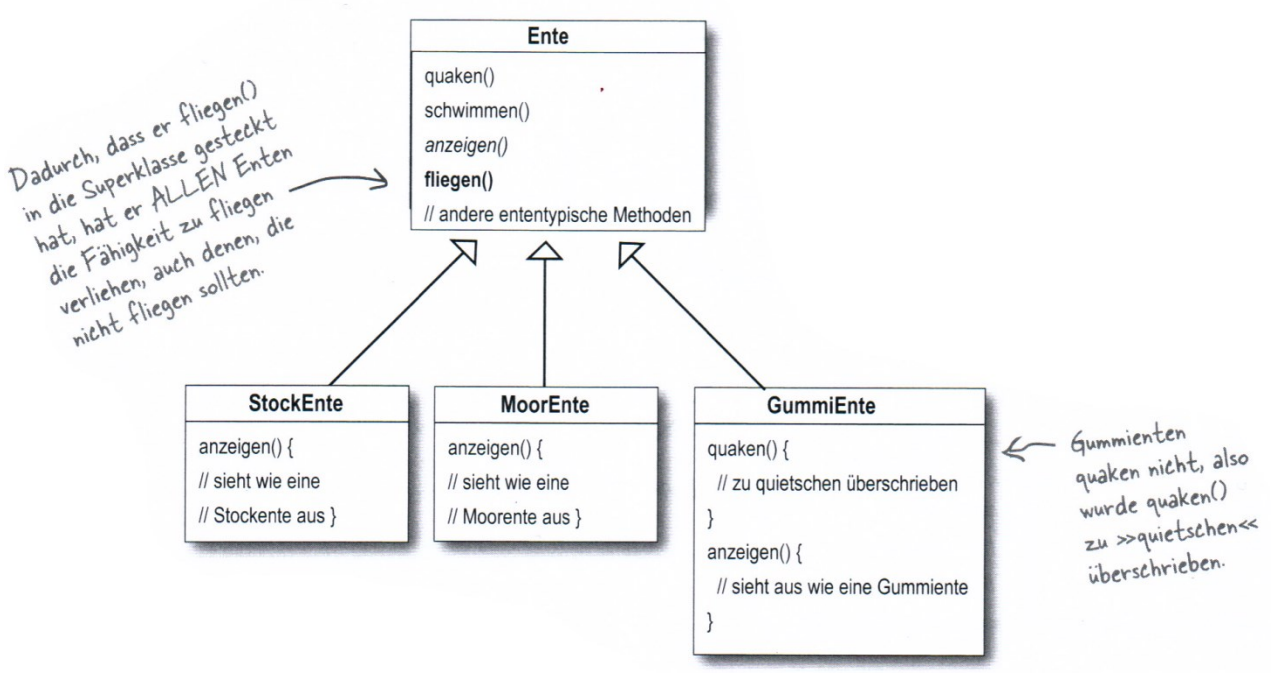
Entwurfsmuster – Strategie-Muster

Erweiterung durch Vererbung

Was passiert, wenn neue Unterklassen ein bestimmtes Verhalten (Methode) gar nicht besitzen?

Es wird eine Unterklasse GummiEnte hinzugefügt, die nicht quakt sondern quietscht und außerdem nicht fliegen kann

Methode quaken() kann überschrieben werden



Freeman et al. (2005) Entwurfsmuster von Kopf bis Fuß. O'Reilly

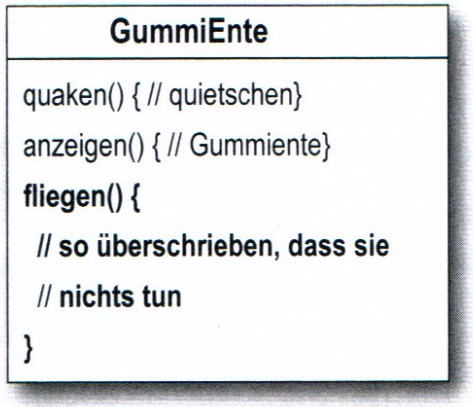
Entwurfsmuster – Strategie-Muster

Erweiterung durch Vererbung

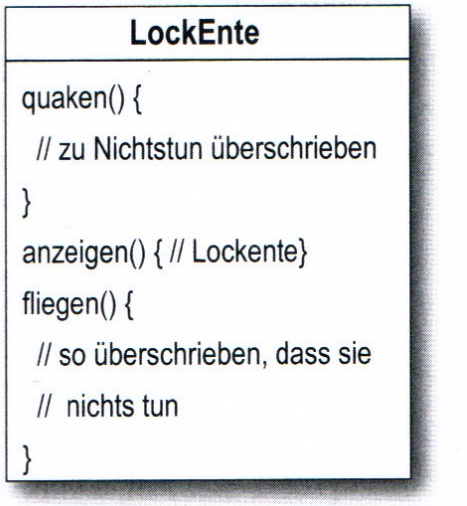
Was passiert, wenn neue Unterklassen ein bestimmtes Verhalten (Methode) gar nicht besitzen?

Es wird eine Unterklasse GummiEnte hinzugefügt, die nicht quakt sondern quietscht und außerdem nicht fliegen kann

Methode fliegen() kann leer überschrieben werden



Gilt auch für weitere Unterklassen



Freeman et al. (2005) Entwurfsmuster von Kopf bis Fuß. O'Reilly

Entwurfsmuster – Strategie-Muster

Probleme der Vererbung bei Erweiterung

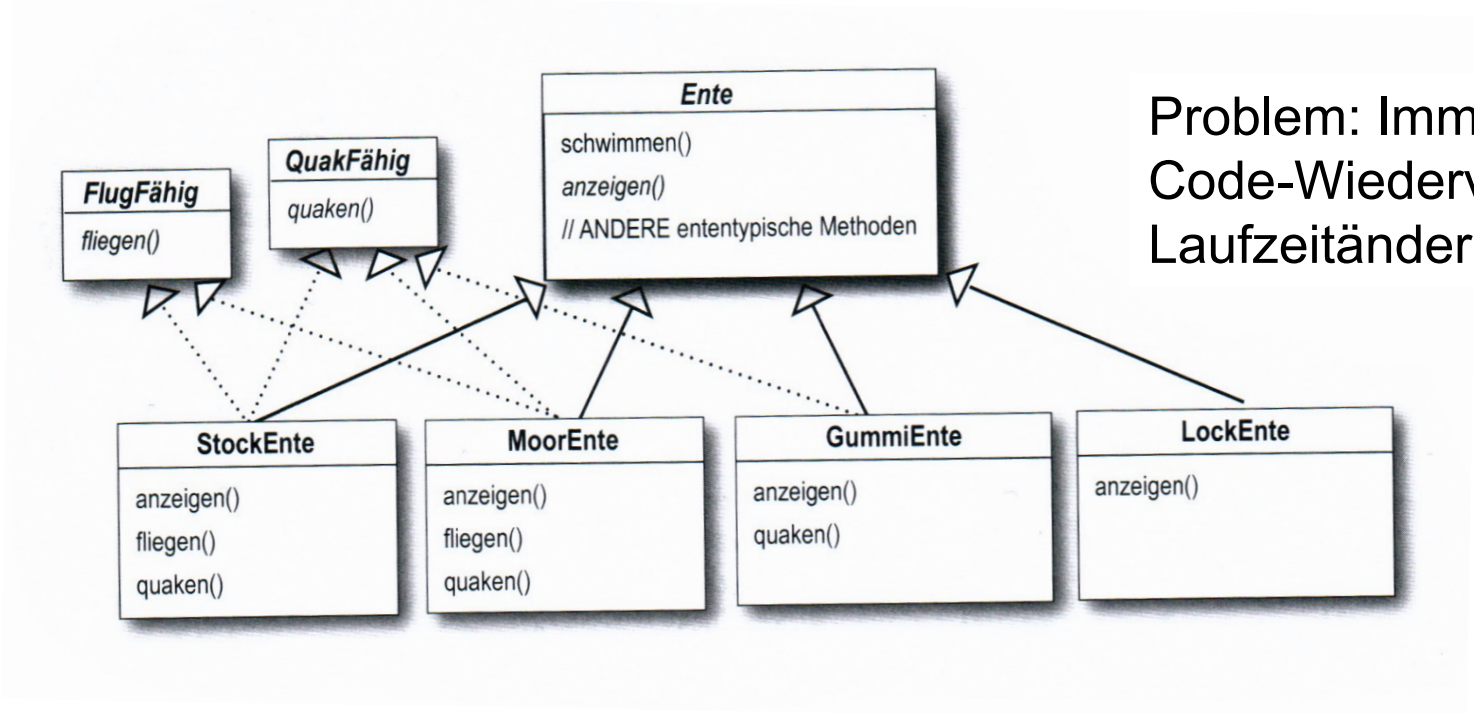
Was sind die generellen Probleme?

- Alle Unterklassen müssen immer alle Funktionen implementieren
  - Bei vielen Funktionen unübersichtlich
- Verhaltensänderung zur Laufzeit nicht möglich
  - Bsp.: Was wenn eine BabyEnte fliegen lernt?
- *Code Duplication*
  - Eine Ente, die wie eine MoorEnte fliegt und wie eine GummiEnte quakt kann keinen Code wiederverwenden

Entwurfsmuster – Strategie-Muster

Verwendung von Interfaces

Eine Lösung könnte die Verwendung von unterschiedlichen Schnittstellen sein, die nur bestimmte Unterklassen implementieren



Problem: Immer noch keine Code-Wiederverwendung und Laufzeitänderung möglich

Freeman et al. (2005) Entwurfsmuster von Kopf bis Fuß. O'Reilly

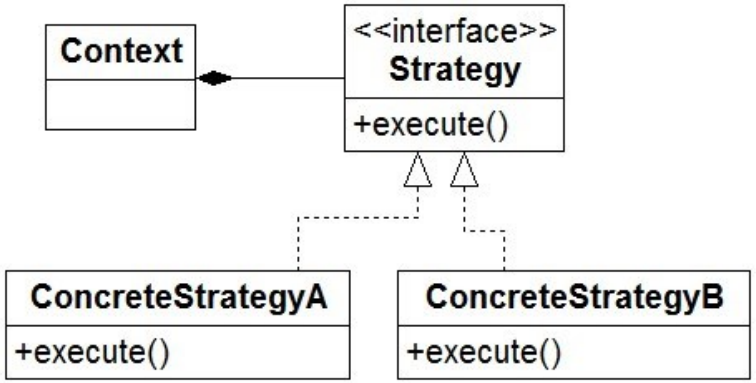
Entwurfsmuster – Strategie-Muster

Strategie-Muster

Prinzipiell gibt es immer Code, der im Laufe der Zeit verändert oder erweitert wird. Mit Hilfe des Strategie-Muster sollen Aspekte einer Anwendung, die sich ändern können, gekapselt werden

Die Verwendung von Strategien bietet sich an, wenn

- viele verwandte Klassen sich nur in ihrem Verhalten unterscheiden
- unterschiedliche Varianten eines Algorithmus benötigt werden
- verschiedene Verhaltensweisen wieder-verwendet werden sollen



Entwurfsmuster – Strategie-Muster

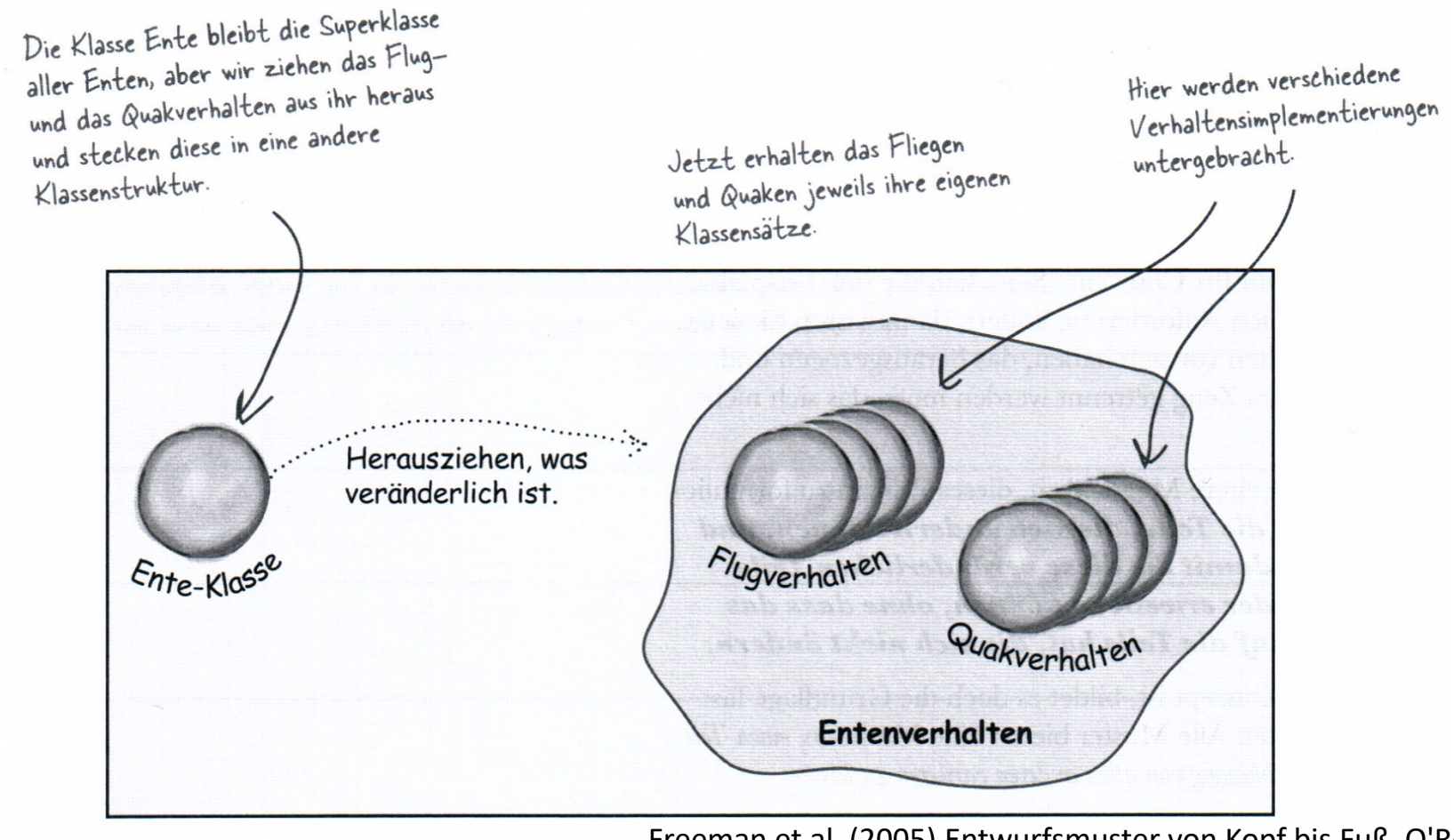
Strategie-Muster

Beim Strategie-Muster wird ein bestimmtes Verhalten nicht direkt in der Unterklasse umgesetzt, sondern es wird für jedes Verhalten eine eigene Klasse auf Basis einer Schnittstelle implementiert

Wesentlicher Vorteil ist, dass die Klassen (Context), die ein Verhalten (Strategy) verwenden, nichts über die Implementierung wissen müssen

Entwurfsmuster – Strategie-Muster

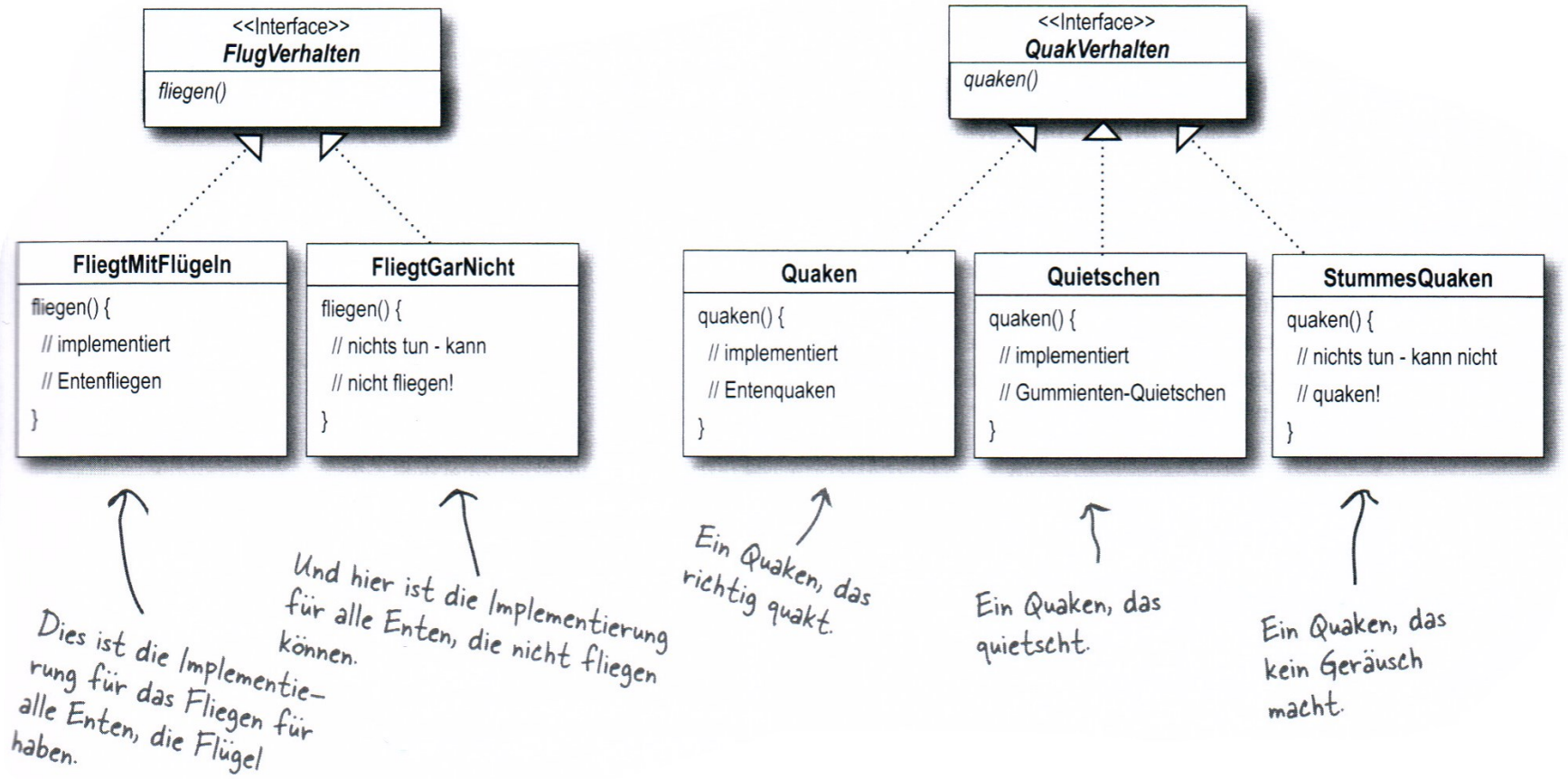
Beispiel: Entenverhalten



Freeman et al. (2005) Entwurfsmuster von Kopf bis Fuß. O'Reilly

Entwurfsmuster – Strategie-Muster

Beispiel: Entenverhalten



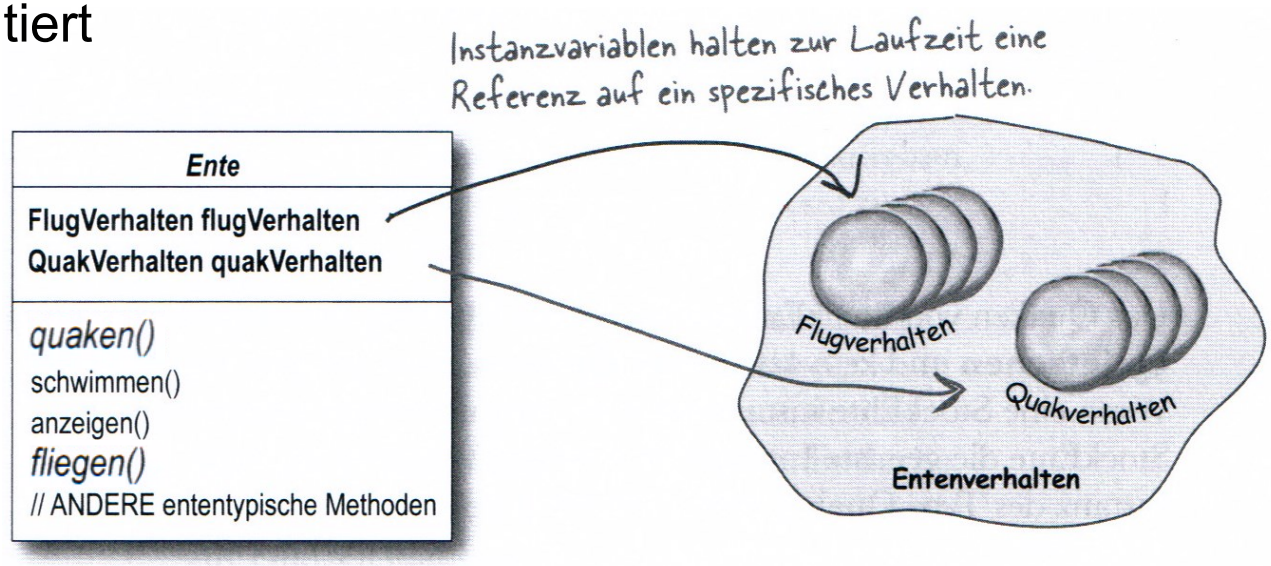
Freeman et al. (2005) Entwurfsmuster von Kopf bis Fuß. O'Reilly

Entwurfsmuster – Strategie-Muster

Integration von Strategien

Für jede Schnittstelle müssen jetzt in der Oberklasse eine eigene Instanzvariablen der entsprechenden Schnittstelle (Komposition) hinzugefügt werden

Die Aufrufe der Strategien werden an die Instanzvariablen delegiert und nur in der Oberklasse implementiert



Freeman et al. (2005) Entwurfsmuster von Kopf bis Fuß. O'Reilly

Entwurfsmuster – Strategie-Muster

Beispiel: Entenverhalten

```
public abstract class Ente {

    // Attribute Verhalten
    protected QuakVerhalten quakVerhalten;

    ...

    // Quaken
    public void quaken() {
        this.quakVerhalten.quaken();
    }

    ...
}
```

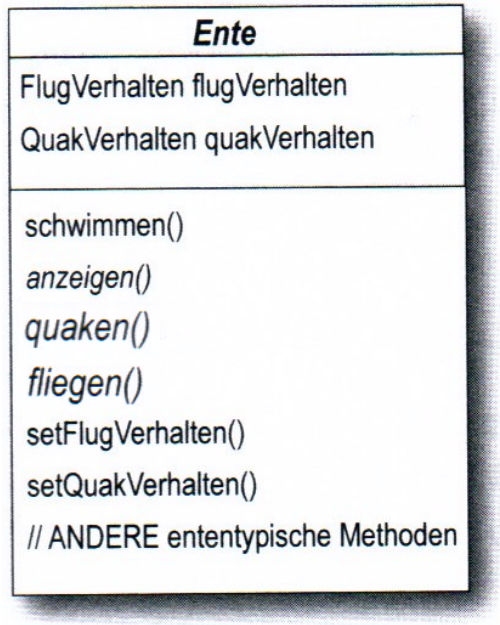
Entwurfsmuster – Strategie-Muster

Integration von Strategien

Die speziellen Strategien werden in den Unterklassen in der Regel innerhalb des Konstruktor erzeugt.

Soll das Verhalten zur Laufzeit geändert werden, sollten auch `set`- Methoden zum ändern der Strategien vorgesehen werden

```
public class GummiEnte extends Ente {

    // Konstruktor
    public GummiEnte() {
        super.quakVerhalten = new Quietschen();
        super.flugVerhalten = new FliegtGarNicht();
    }

    ...
}
```



Freeman et al. (2005) Entwurfsmuster von Kopf bis Fuß. O'Reilly

Entwurfsmuster – Strategie-Muster

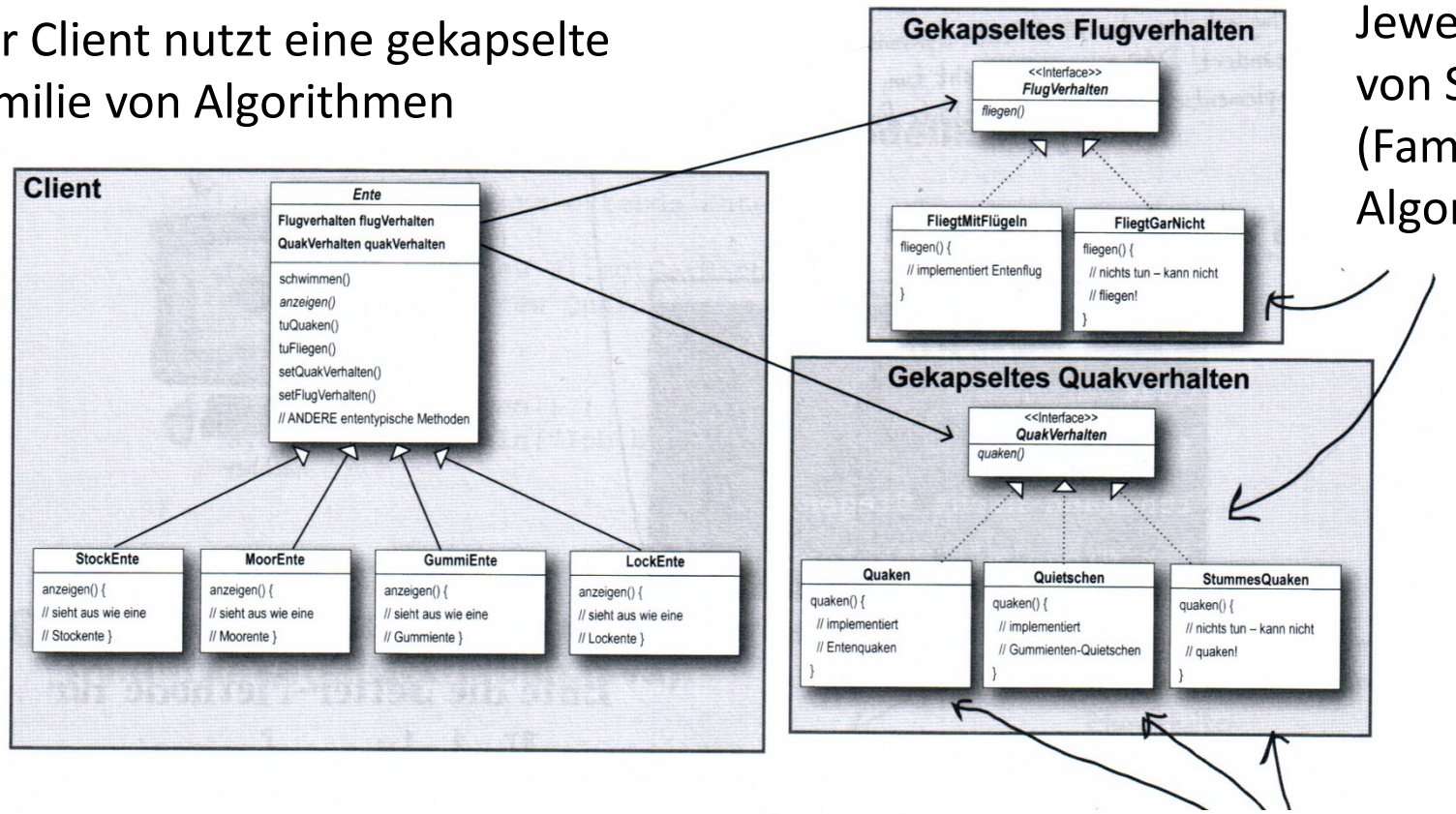
Beispiel: Entenverhalten

```
public abstract class Ente {

    // Attribute Verhalten
    protected QuakVerhalten quakVerhalten;

    ...

    // Quaken
    public void quaken() {
        this.quakVerhalten.quaken();
    }

    ...

    // Quaken setzen
    public void setQuaken(QuakVerhalten quakVerhalten) {
        this.quakVerhalten = quakVerhalten;
    }
}
```

Entwurfsmuster – Strategie-Muster

Beispiel: Entenverhalten

Der Client nutzt eine gekapselte Familie von Algorithmen



Jeweils ein Satz von Strategien (Familie von Algorithmen)

Algorithmen können ausgetauscht werden

Entwurfsmuster – Strategie-Muster

Fazit

In einigen Fällen sollte die **Komposition** der **Vererbung** vorgezogen werden. Dadurch ist das Verhalten zur Laufzeit änderbar und wiederverwendbar.

Durch die Verwendung von Schnittstellen kann die Implementierung von verschiedenen Strategien aufgeteilt werden.

Gekapselte Algorithmen lassen sich sehr gut für andere Problemstellungen wiederverwenden.

Entwurfsmuster – Strategie-Muster

Lambda-Funktionen

Ab Java 8 gibt es eine Kurzschreibweise für Interface-Realisierung mit nur einer Funktion.

```
public interface RabattStrategie {
    public double berechneRabatt(double b);
}

public class Produkt {
    RabattStrategie rabattStrategie;
    public Produkt(RabattStrategie rabattStrategie) {
        this.rabattStrategie = rabattStrategie;
    }
}
```

Lambda-Funktion

```
    ...
    // 20% auf Rasenmaeher
    Produkt rasenmaeher = new Produkt((preis) -> preis*0.8);
    // Kein Rabatt auf Tiernahrung
    Produkt tiernahrung = new Produkt((p) -> p);
    // Bohrmaschinen über 100€ für 50% billiger
    Produkt bohrmaschine = new Produkt((p) -> p>=100.0 ? p*0.5 : p);
```

Entwurfsmuster – Strategie-Muster

Lambda-Funktionen

```
// Bohrmaschinen über 100€ für 50% billiger
Produkt bohrmaschine = new Produkt((p) -> p>=100.0 ? p*0.5 : p);
```

Ist äquivalent zu:

```
public class BohrmaschineRabattStrategie implements RabattStrategie {
    public double berechneRabatt(double p) {
        return p>=100.0 ? p*0.5 : p;
    }
}

...
Produkt bohrmaschine = new Produkt(new BohrmaschineRabattStrategie());
```

Da bei einer Lambda-Funktion keine Klasse benannt wird, werden Lambda-Funktionen auch **anonyme Funktionen** genannt.

Entwurfsmuster – Strategie-Muster

Lambda-Funktionen

Beispiel: Java Comparator-Interface für Sortierfunktionen

```
ArrayList<Produkt> produktliste = new ArrayList<Produkt>();
...

// Sortierstrategie nach Preis
produktliste.sort((p1, p2) -> p1.getPreis() - p2.getPreis());

// Sortierstrategie nach Name
produktliste.sort((p1, p2) -> p1.getName().compareTo(p2.getName()));
```

RUHR-UNIVERSITÄT BOCHUM
Lehrstuhl für Informatik im Bauwesen

RUB

Programmierung und Programmiersprachen

Sommersemester 2023

Iterator-Muster

Entwurfsmuster – Iterator-Muster

Iteratoren in Java

Erinnerung:

- Vereinfachte `for` Schleife

```
for (Point2D p : points) {
      System.out.println(p);
}
```

- Normale `while` Schleife

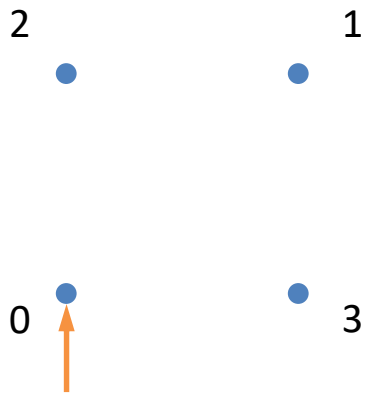
```
Iterator<Point2D> iter = points.iterator();
while (iter.hasNext()) {
      Point2D p = iter.next();
      System.out.println(p);
}
```

- Java stellt die Interfaces **`Iterable`** und **`Iterator`** zur Verfügung
- Das Iterable (der Aggregator) kann mithilfe eines Iterators durchlaufen werden

Entwurfsmuster – Iterator-Muster

Iteratoren in Java

Malen nach Zahlen



Iterable:

```
0: Point2D(0,0)
1: Point2D(1,1)
2: Point2D(0,1)
3: Point2D(1,0)
```

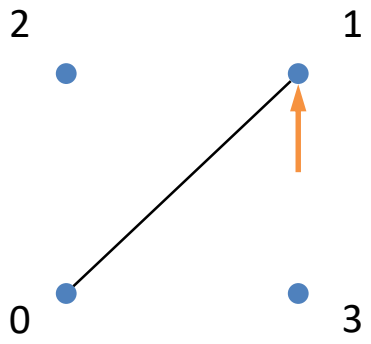
Iterator:

```
index = 0

next(): Point2D(0,0)
hasNext(): true
```

Entwurfsmuster – Iterator-Muster

Iteratoren in Java

Malen nach Zahlen



Iterable:

```
0: Point2D(0,0)
1: Point2D(1,1)
2: Point2D(0,1)
3: Point2D(1,0)
```

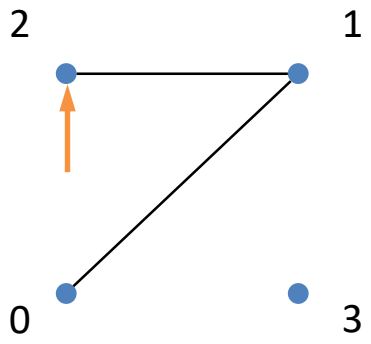
Iterator:

```
index = 1

next(): Point2D(1,1)
hasNext(): true
```

Entwurfsmuster – Iterator-Muster

Iteratoren in Java

Malen nach Zahlen



Iterable:

```
0: Point2D(0,0)
1: Point2D(1,1)
2: Point2D(0,1)
3: Point2D(1,0)
```

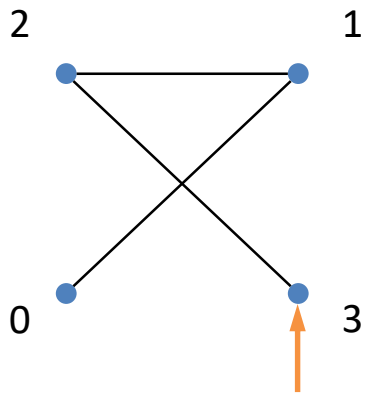
Iterator:

```
index = 2

next(): Point2D(0,1)
hasNext(): true
```

Entwurfsmuster – Iterator-Muster

Iteratoren in Java

Malen nach Zahlen



Iterable:

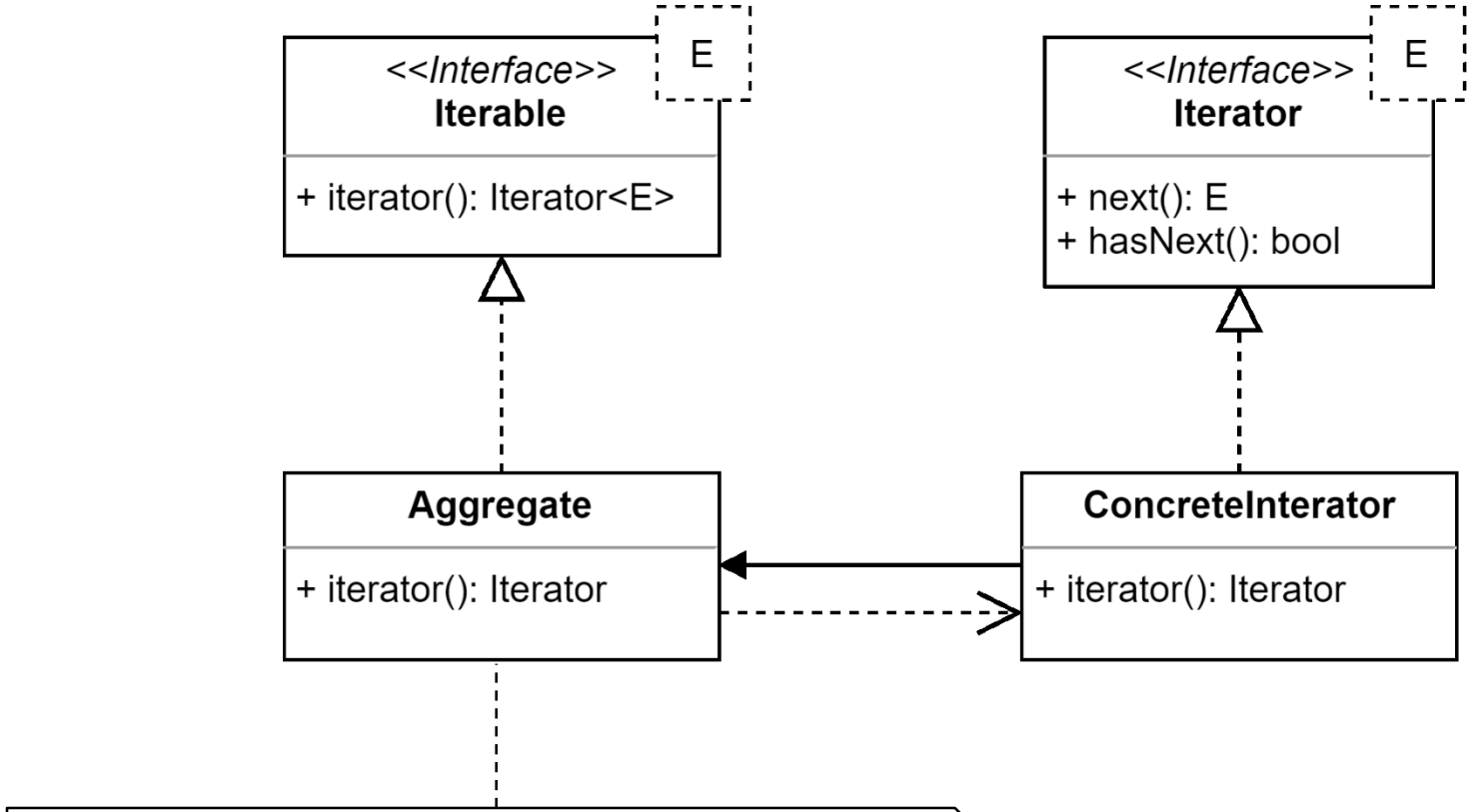
```
0: Point2D(0,0)
1: Point2D(1,1)
2: Point2D(0,1)
3: Point2D(1,0)
```

Iterator:

```
index = 3

next(): Point2D(1,0)
hasNext(): false
```

Entwurfsmuster – Iterator-Muster

Das Iteratormuster




```
public Iterator iterator() {
  return new ConcreteIterator(this);
}
```

Entwurfsmuster – Iterator-Muster

Das Iteratormuster

Komponenten

- Das Interface **`Iterator`** mit den Funktionen **`next()`** und **`hasNext()`**
  - Weitere mögliche Funktionen: **`peek()`**, **`first()`**, **`previous()`**, **`skip()`**
- Das Interface **`Iterable`** mit der Funktion **`iterator()`**
  - Kann auch weitere Iteratoren vorschreiben (z.B. **`reverse()`**)
- Der Aggregator
  - Datenspeicher (z.B. Listen, Warteschlangen, etc.)
  - Berechnungsvorgabe (z.B. mathematische Reihen)
- Der realisierte Iterator
  - Kapselt den Zustand der Iteration
  - Hat meist privilegierten Zugriff auf das Aggregat (z.B. durch interne Klassen oder package-private Attribute)

Entwurfsmuster – Iterator-Muster

Das Iteratormuster

```
public class MalenNachZahlen implements Iterable<Point2D> {
    private Point2D[] punkte;

    public Iterator<Point2D> iterator() {
        return new MalenNachZahlenIterator(this);
    }

    // Konstruktor & Getter
    // ...
}
```

Entwurfsmuster – Iterator-Muster

```
public class MalenNachZahlenIterator implements Iterator<Point2D> {
    private MalenNachZahlen aggregate;
    private int index = 0;  // Zustand des Iterators

    public Point2D next() {
        Point2D p = aggregate.getPunkte()[index];
        index++;  // Zustand wird verändert
        return p;
    }
    public boolean hasNext() {
        return index < aggregate.getPunkte().length;
    }
    // Konstruktor
    // ...
}
```

Entwurfsmuster – Iterator-Muster

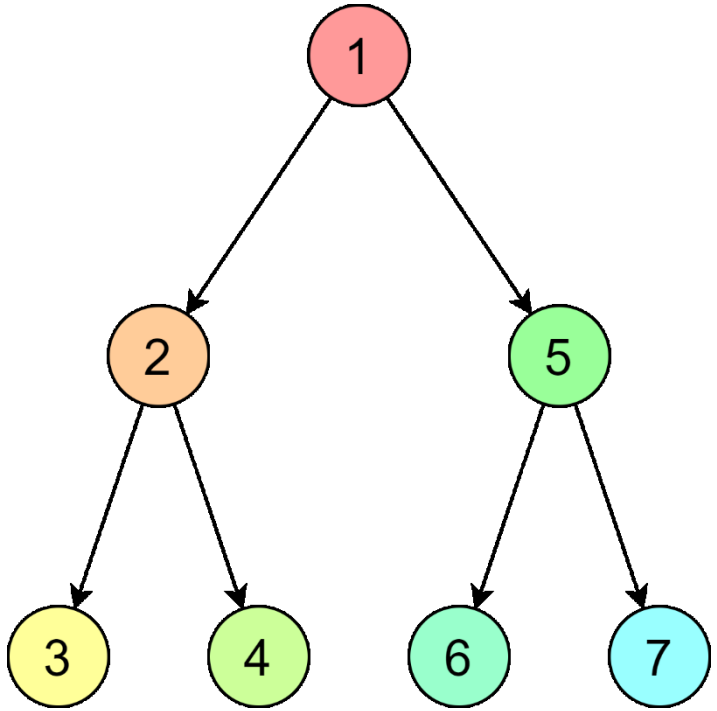
Das Iteratormuster

```
public class Test {
    public static void main(String[] args) {
        MalenNachZahlen malen = new MalenNachZahlen();
        for(Point2D p: malen) {
            p.draw();
        }
    }
}
```

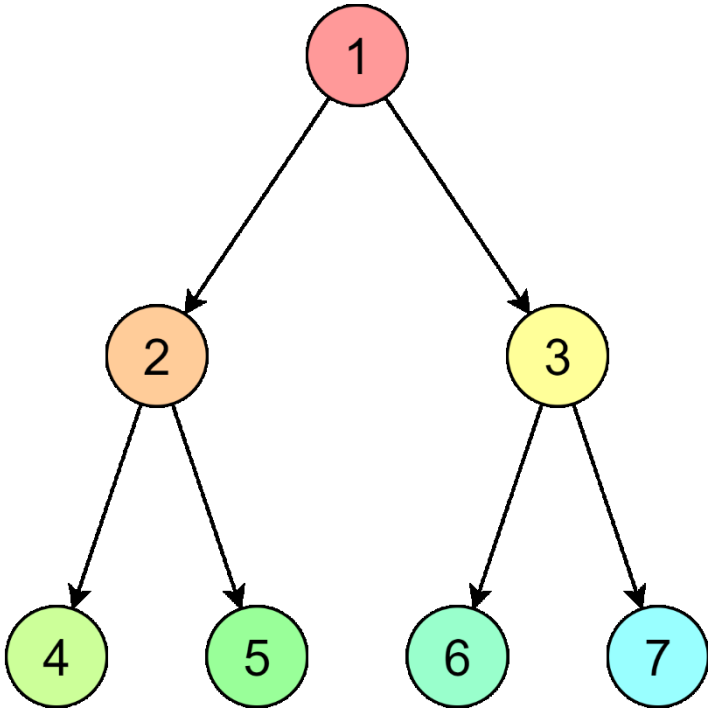
Entwurfsmuster – Iterator-Muster

Das Iteratormuster

Beispiel: Traversieren eines Binärbaums



Tiefensuche

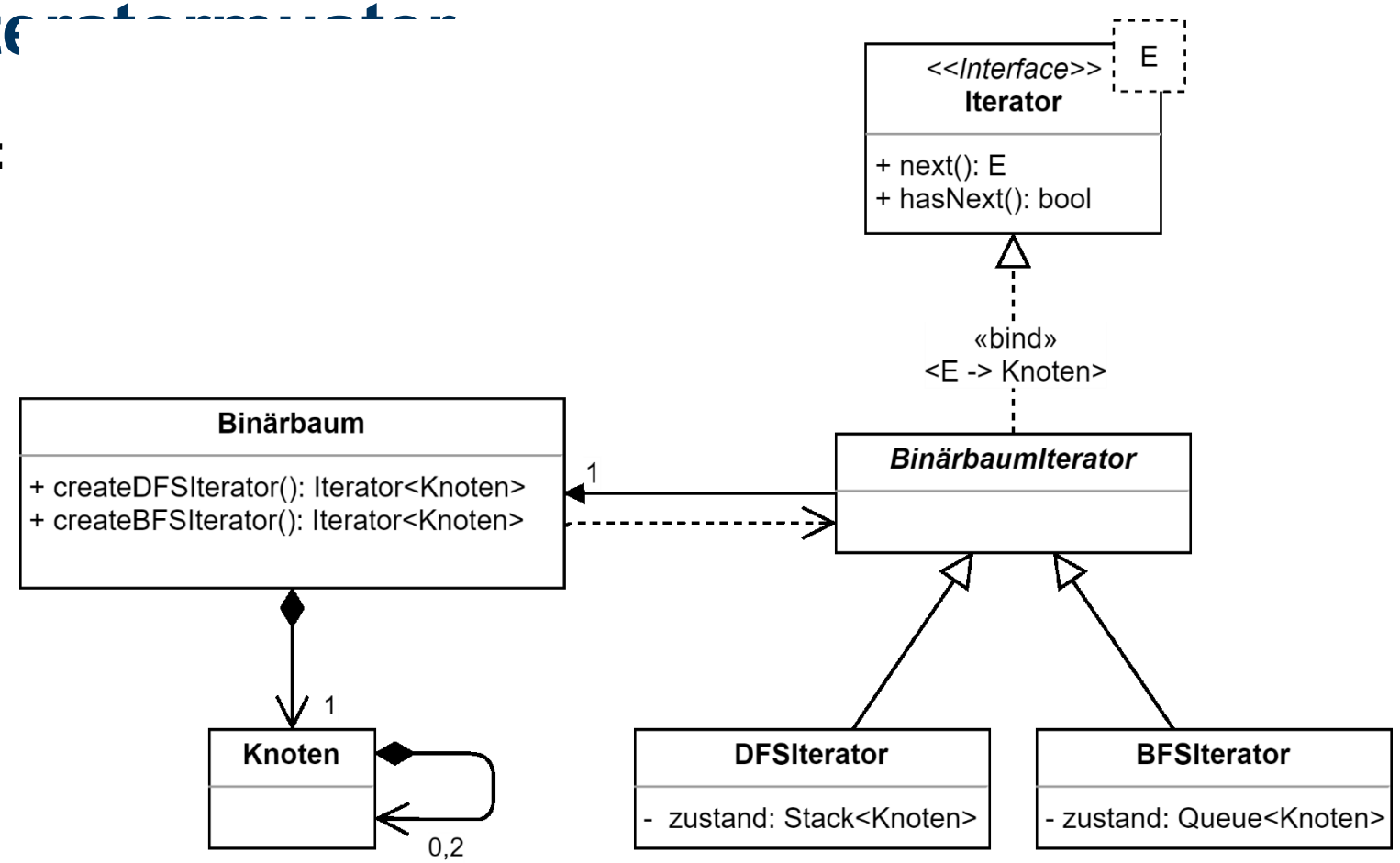


Breitensuche

Entwurfsmuster – Iterator-Muster

Das Iteratormuster

Beispiel:



Implementierung in der Übung!

Entwurfsmuster – Iterator-Muster

Das Iteratormuster

Vorteile des Iteratormusters

- Der Zustand einer Iteration wird in einer Klasse gekapselt
  - $\Rightarrow$ Mehrere parallele Iterationen eines Objektes
  - $\Rightarrow$ Verschiedene Iterationsvarianten für das selbe Aggregat
- Generalisierter Zugriff von außen (z.B. durch for-each-Syntax)
- Zugriff auf das Aggregat kann kontrolliert werden

Entwurfsmuster – Iterator-Muster

Das Iteratormuster

Variationen des Iteratormusters

- Robuster Iterator
  - Verändert sich der Aggregator (z.B. durch entfernen/hinzufügen) muss auch der Iterator geändert werden
- Cursor-Objekte
  - Die Parameter der Traversierung werden vom Client gesetzt (z.B. Suchparameter oder Filter)
  - Häufig benutzt für Datenbankabfragen (SQL, etc.)