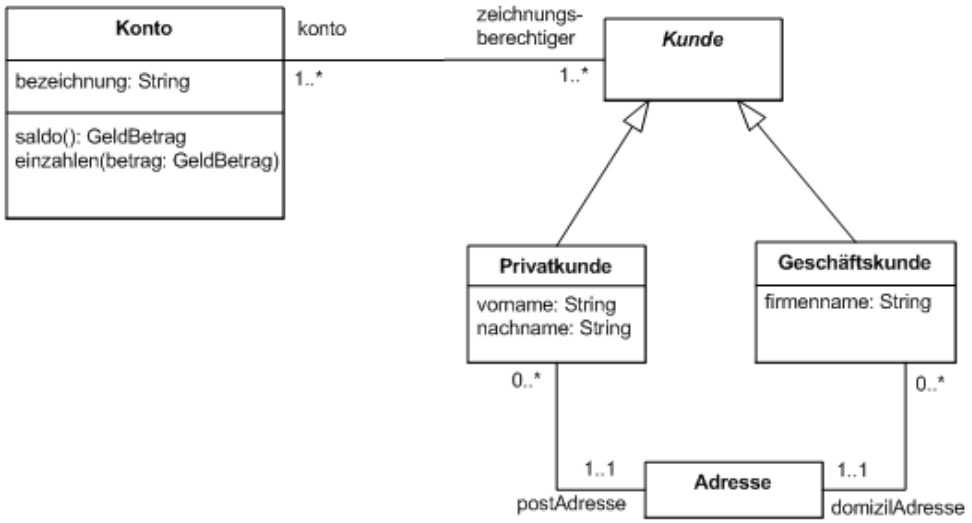
RUHR-UNIVERSITÄT BOCHUM
Lehrstuhl für Informatik im Bauwesen

RUB

Programmierung und Programmiersprachen

Sommersemester 2023

UML Klassendiagramme



Objektorientierte Modellierung - UML

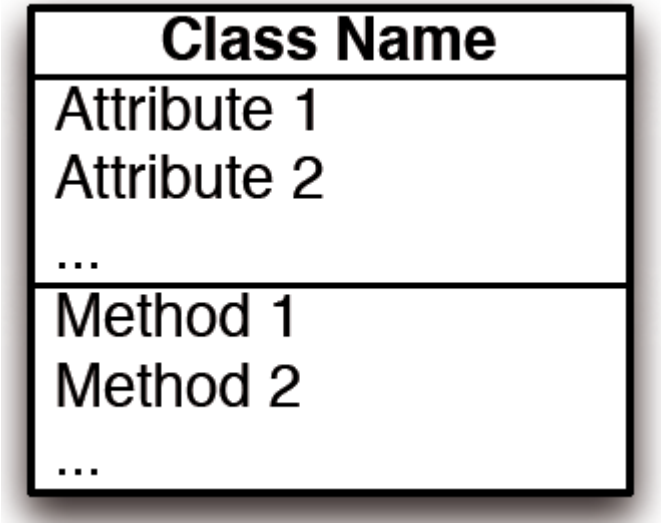
Unified Modeling Language UML

Grafische Beschreibung des Software Designs

- Sprachunabhängige Beschreibung
- Statische Klassenstrukturen
- Dynamische Zustandsänderungen
- Wichtig für die Arbeit in interdisziplinären Teams

UML Klassendiagramme

- Klasse(n) mit Attributen und Methoden
- Beziehungen zwischen Klassen



Objektorientierte Modellierung - UML

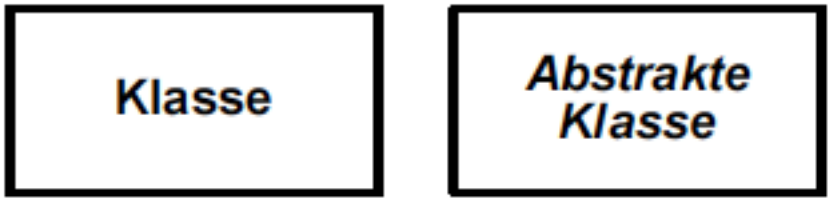
UML Diagramm für Rechtecke

Rectangle
- x: double - y: double - w: double - h: double
+ Rectangle(x: double, y: double, w: double , h: double) ...

- Klassenname
- Attribute mit Datentypen und Zugriffsrechten (hier `private`)
- Methoden – Konstruktor zum Erstellen von Objekten
- Konstruktoren besitzen den gleichen Namen wie die Klasse

Objektorientierte Modellierung - UML

UML Syntax und Sichtbarkeiten



Syntax für Attribute:
Sichtbarkeit Attributname : Paket::Typ [Multiplizität Ordnung] = Initialwert {Eigenschaftswerte}
Eigenschaftswerte: {readOnly}, {ordered}, {composite}

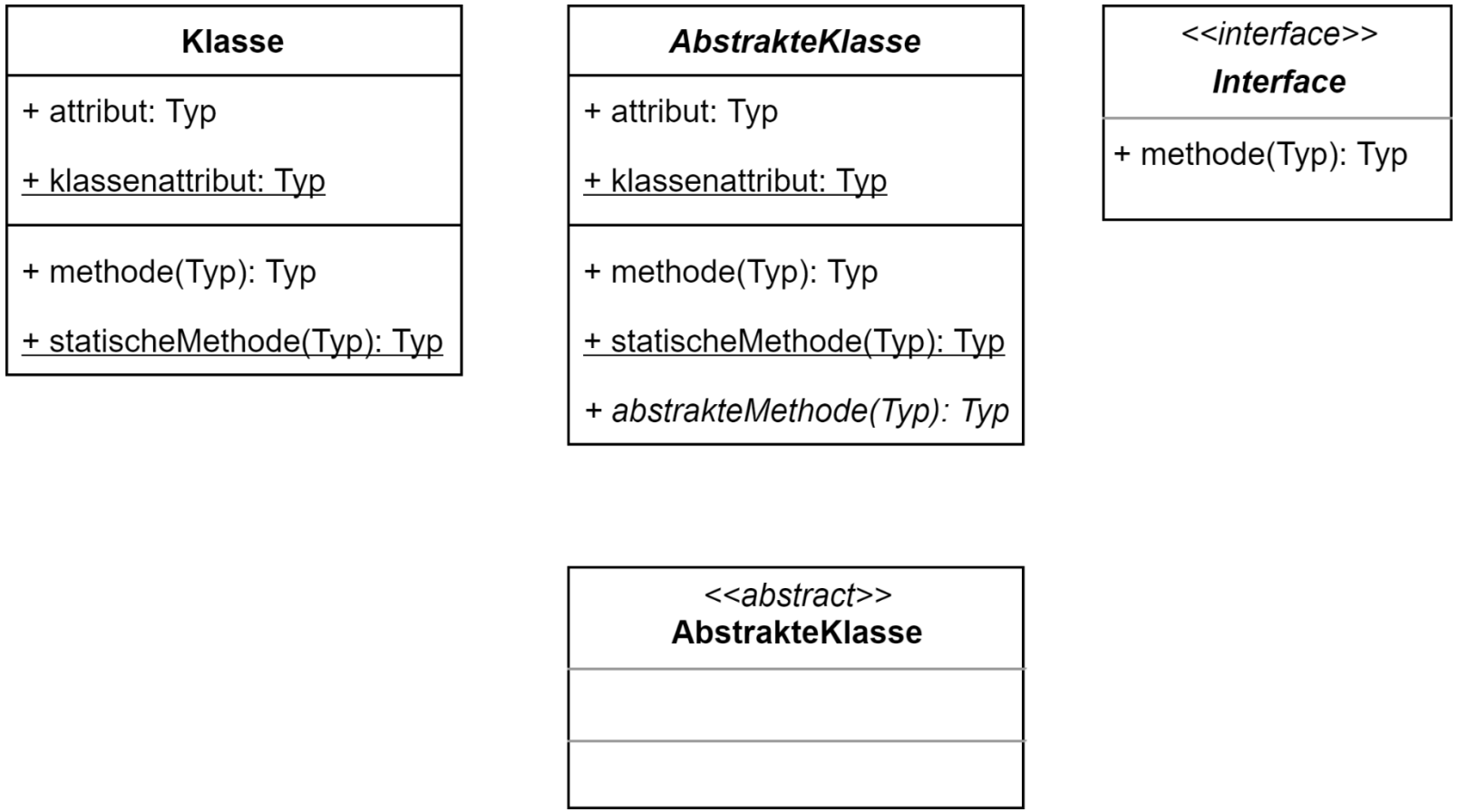
Syntax für Operationen:
Sichtbarkeit Operationsname (Parameterliste):Rückgabetyp {Eigenschaftswerte}

Sichtbarkeit:
+ public element
# protected element
– private element
~ package element

Parameterliste: Richtung Name : Typ = Standardwert
Eigenschaftswerte: {query}
Richtung: in, out, inout

Objektorientierte Modellierung - UML

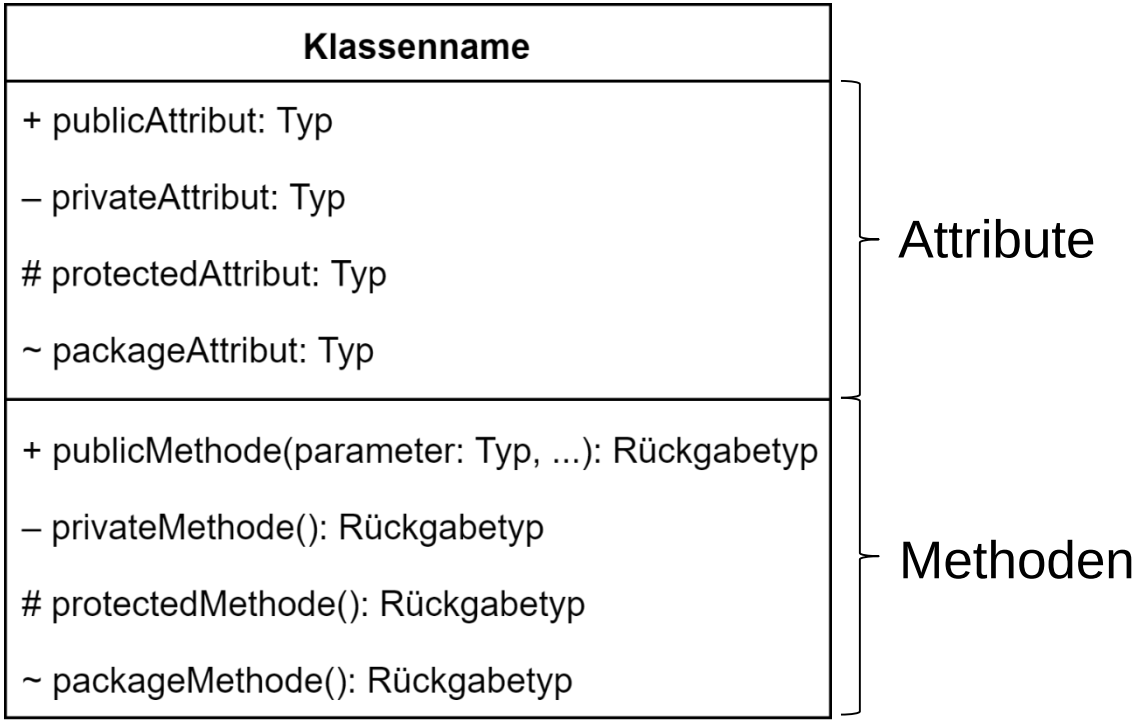
UML Klassentypen



Objektorientierte Modellierung - UML

UML Syntax und Sichtbarkeiten

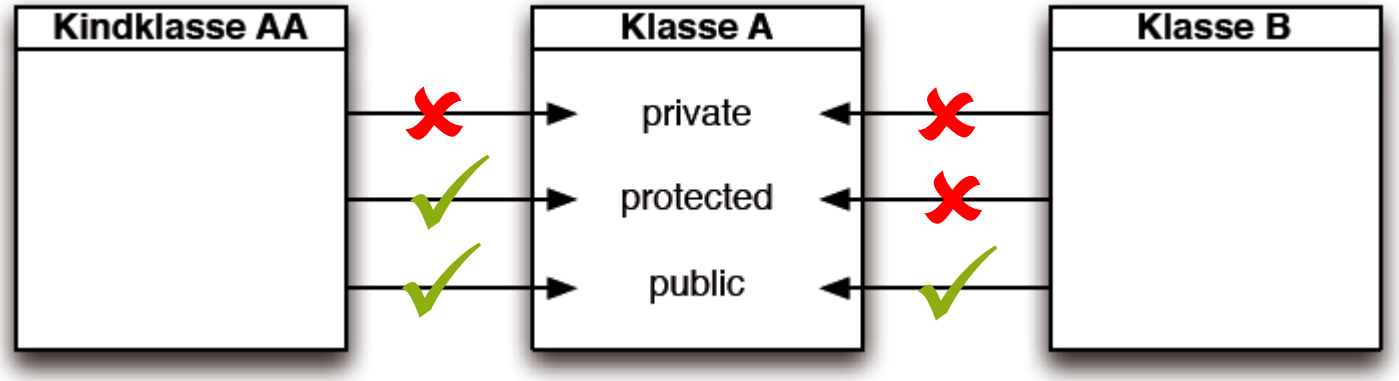
Symbol	Java-Keyword
+	`public`
–	`private`
#	`protected`
~	Keins/Standard



Objektorientierte Modellierung - UML

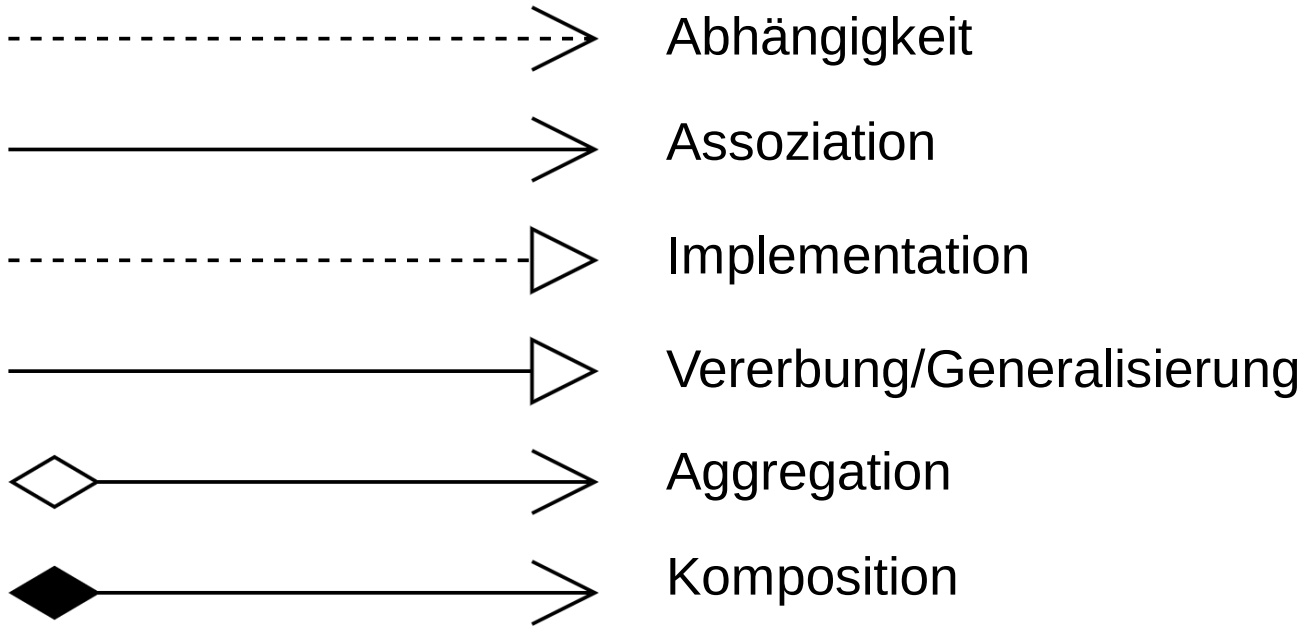
Zugriffsrechte

- Um den Zugriff auf Attribute und Methoden der Elternklasse zu ermöglichen, dürfen diese nicht als `private` gekennzeichnet sein
- Um einen eingeschränkten Zugriff nur für die Kindklassen zu ermöglichen, kann das Zugriffsrecht `protected` verwendet werden



Objektorientierte Modellierung - UML

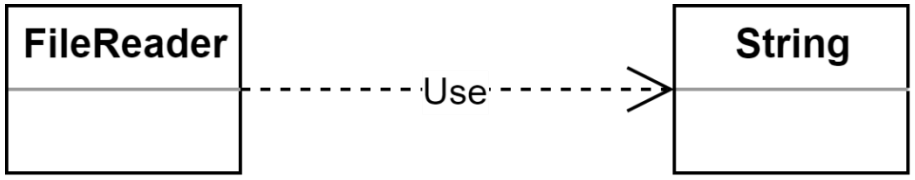
Relationen zwischen Klassen



Objektorientierte Modellierung - UML

Abhängigkeit

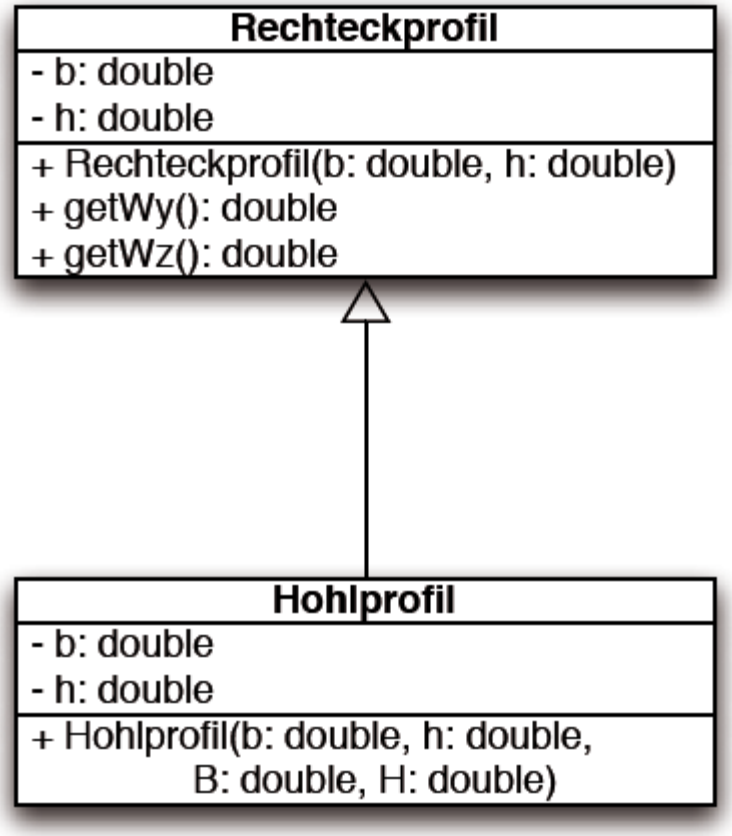
- Eine Klasse benutzt eine andere Klasse (in einem Funktionskörper oder als Parameter)
- Immer unidirektional
- Schwächste der Assoziationen
- Ändert sich etwas an der äußeren Struktur der Zielklasse, so muss eventuell auch die abhängige Klasse verändert werden



Objektorientierte Modellierung - UML

Vererbung/Generalisierung

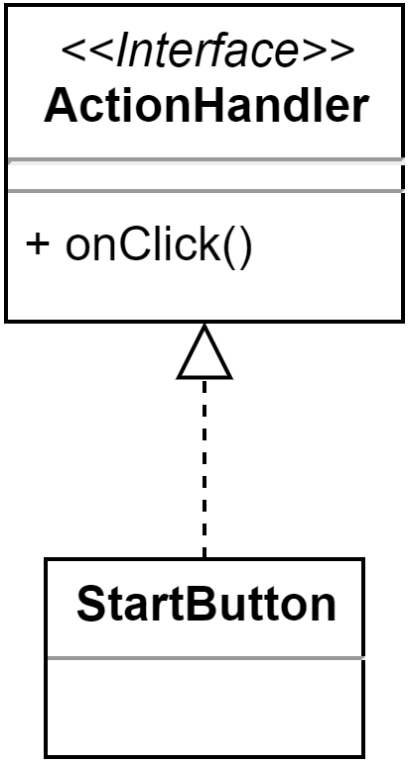
- Die Klasse `Hohlprofil` soll alle Attribute und Methoden der Klasse `Rechteckprofil` erben
- Die vererbten Attribute und Methoden werden nicht noch einmal aufgenommen
- Zirkuläre Vererbungshierarchien sind nicht erlaubt
- Java-Schlüsselwort `extends`



Objektorientierte Modellierung - UML

Implementation/Realisierung

- Nur zwischen Klasse und Interface möglich
- Implementierende Klassen müssen die Funktionen des Interfaces nicht nochmal auflisten
- Java-Schlüsselwort `implements`

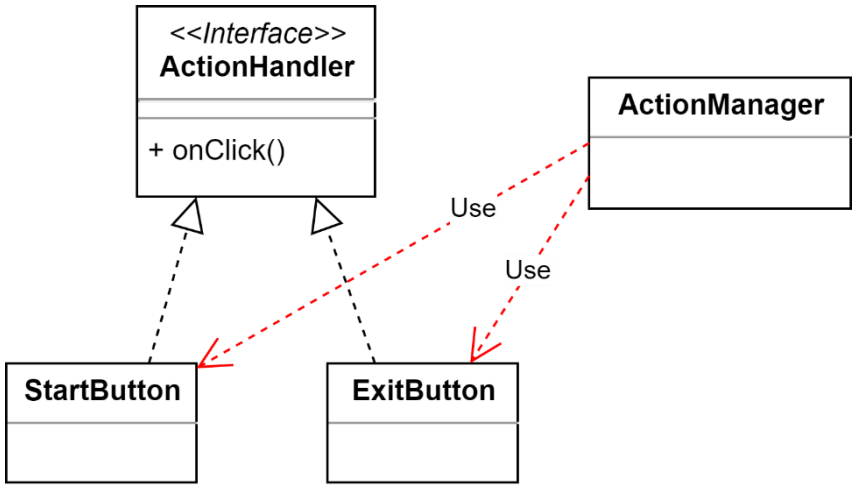


Objektorientierte Modellierung - UML

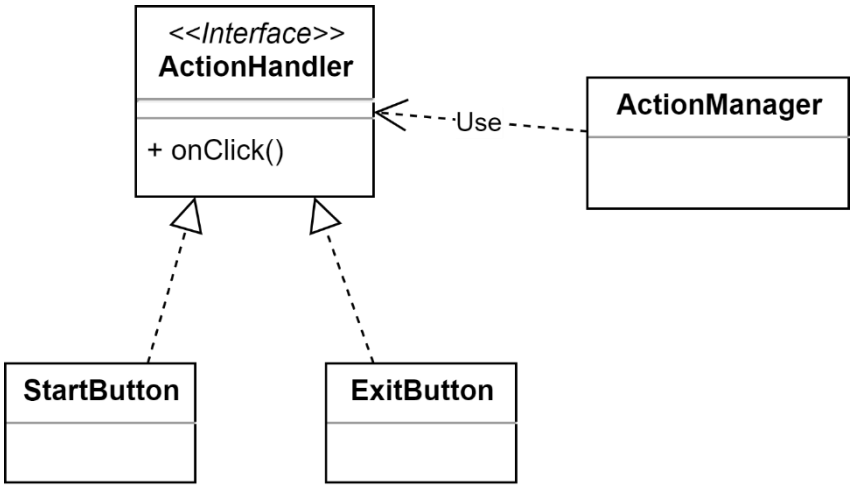
Implementation/Realisierung

Auf Subtyping achten!

Unsauber



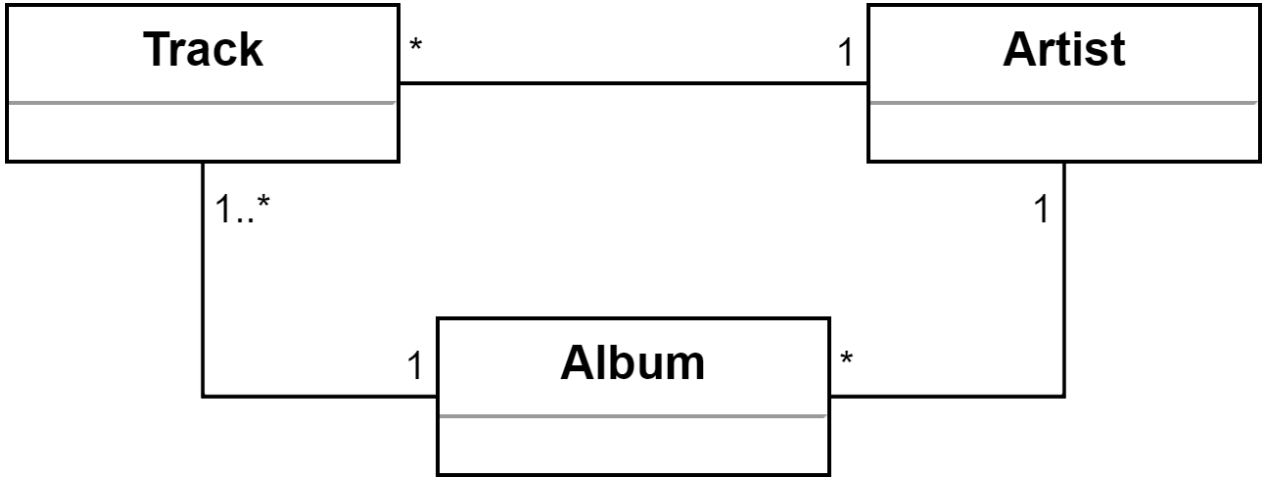
Besser



Objektorientierte Modellierung - UML

Assoziation

- Längerfristige Beziehung zweier Klassen (meist durch Attribute)
- Bestimmt Multiplizität zwischen Objekten

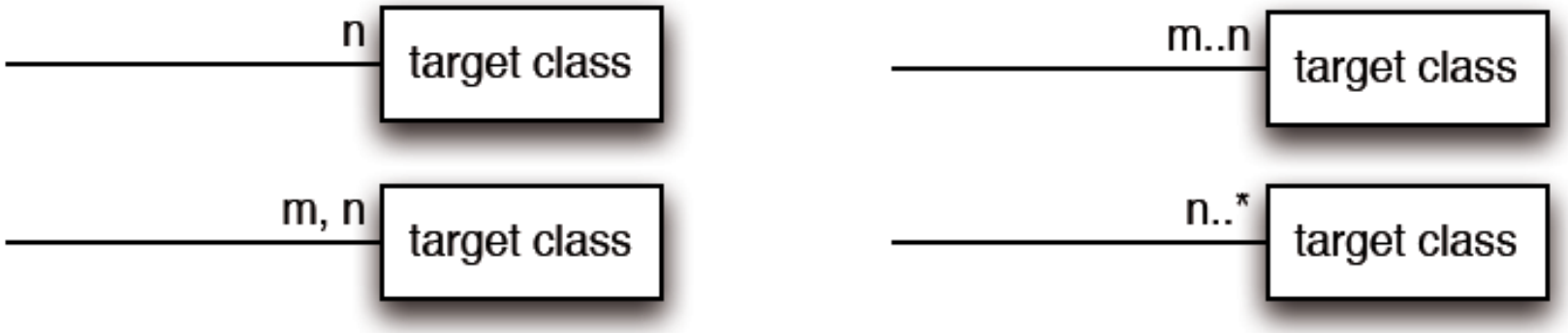


- Ein Track hat genau einen Artist
- Ein Artist kann beliebig viele Tracks haben
- Ein Album besteht aus mindestens einem Track
- …

Objektorientierte Modellierung - UML

Multiplizitäten

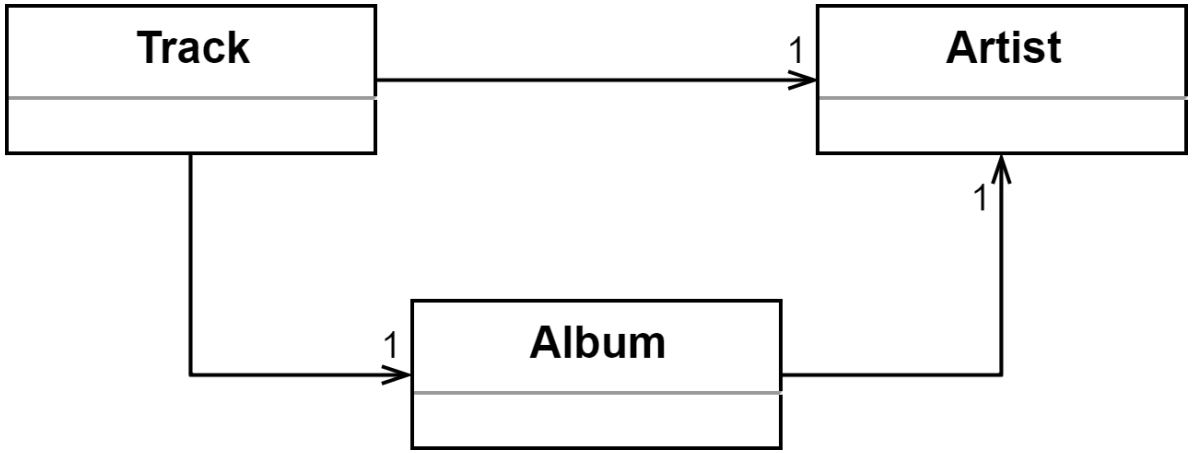
Mit Hilfe von Multiplizitäten wird angegeben, wie viele Objekte einer Klasse gleichzeitig an einer Beziehung teilnehmen



Objektorientierte Modellierung - UML

Assoziation

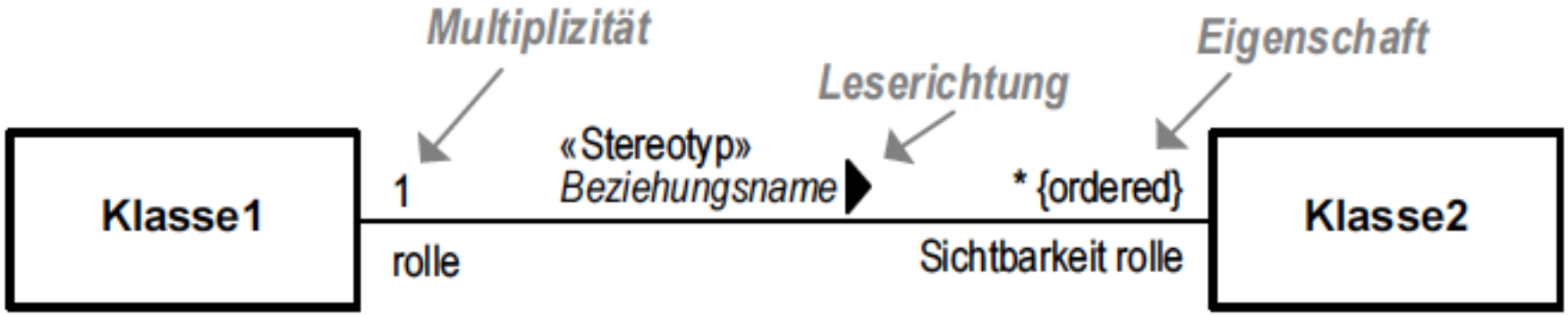
- Direktionale Assoziationen geben die Richtung der Bekanntheit an




```
class Artist {

}

class Track {
    Artist artist;
    Album album;
}

class Album {
    Artist artist;
}
```

Objektorientierte Modellierung - UML

Assoziation

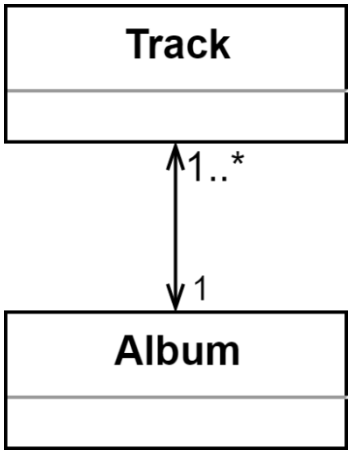


Objektorientierte Modellierung - UML

Assoziation

- UML-Notation gibt nicht vor wie die Implementierung aussieht!
- Eine Möglichkeit: Prüfen im Konstruktor

```
class Album {
    ArrayList<Track> tracks;

    public Album(List<Track> tracks) {
        if(tracks.isEmpty()) {
            throw new java.lang.IllegalArgumentException(
                    "The Album requires at least one track!"
                );
        }
        this.tracks = new ArrayList<>(tracks);
    }
}
```



- Alternative: Factory-Muster

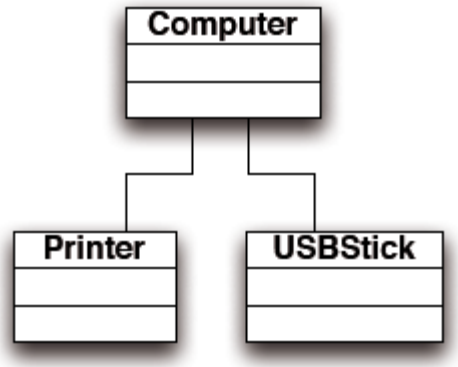
Objektorientierte Modellierung - UML

Assoziation: Laufzeit

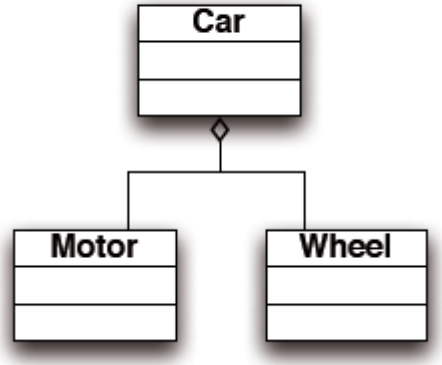
Eine normale Assoziation hat eine unbestimmte Laufzeit.

Aggregation und **Komposition** sind zwei Assoziationstypen bei denen die Laufzeit bestimmt ist.

Association



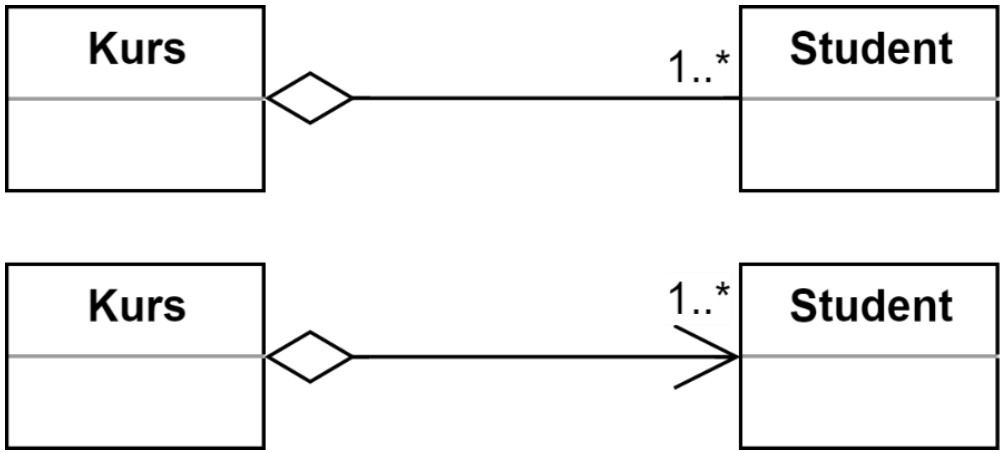
Aggregation



Objektorientierte Modellierung - UML

Aggregation

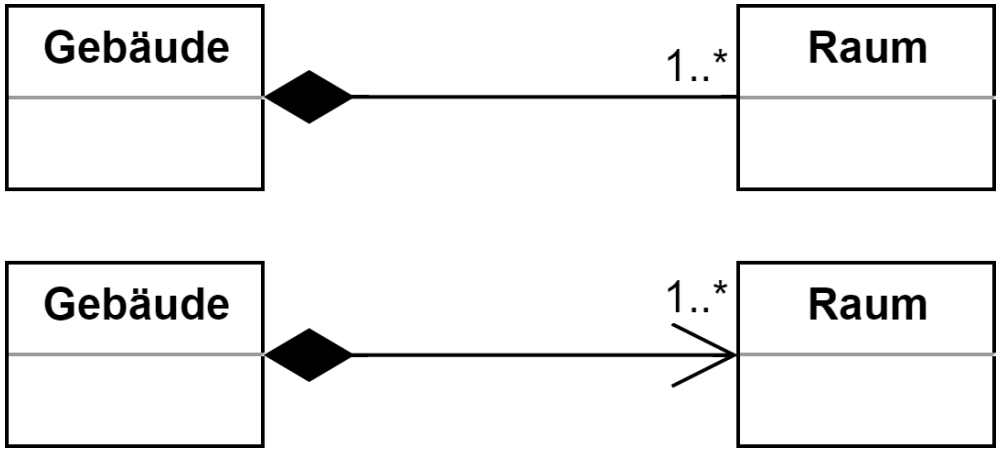
Bei einer **Aggregation** kann ein Unterobjekt unabhängig vom Oberobjekt existieren.



Objektorientierte Modellierung - UML

Komposition

Bei einer **Komposition** existieren die Unterobjekte nur im Zusammenhang mit dem Oberobjekt. Die Laufzeit des Unterobjektes sollte maximal so lange sein wie die des Oberobjektes.

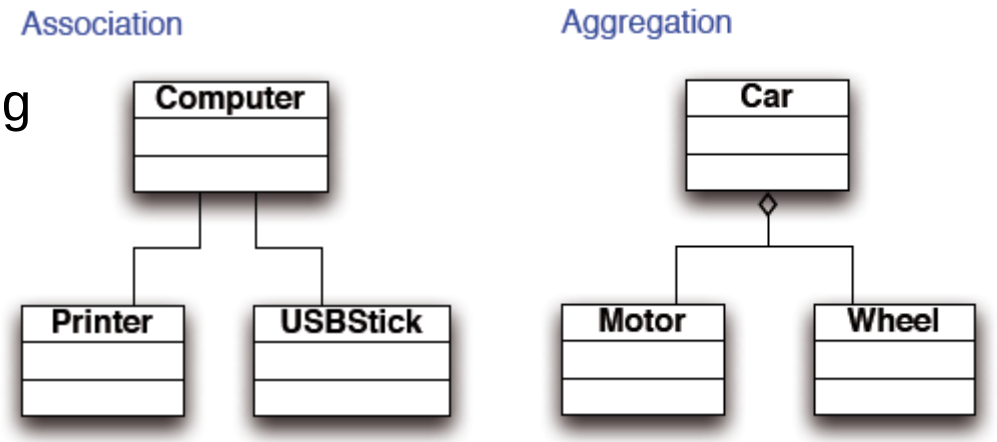


Objektorientierte Modellierung - UML

Assoziationen zwischen Objekten

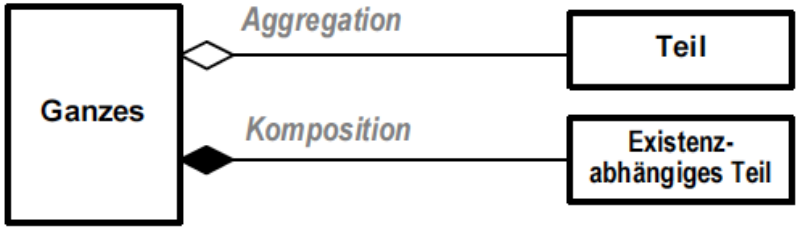
Beziehungen zwischen Objekten

- haben unterschiedliche Bedeutung
- können in der Anzahl variieren
- werden zur Laufzeit aufgebaut
- sind eine bestimmte Zeit gültig



Elementare Beziehungstypen

- Einfache Beziehung (verwendet)
- Aggregation (besteht aus)
- Komposition (Existenz-abhängig)



Speichern von Beziehungen

Beziehungen können auch für bestimmte Zeit gespeichert werden

- zwingend bei Aggregationen
- häufig bei Zuordnungen

In Java werden Attribute bzw. Variablen zur Speicherung von Beziehungen zu anderen Objekten verwendet

- Beziehung zu maximal einem Objekt: als einfaches Attribut
- Feste Anzahl von Beziehungen: als Array
- Variable Anzahl von Beziehungen: dynamische Datenstruktur (z.B. `HashSet` oder `ArrayList`)

Objektorientierte Modellierung - UML

Was gehört in eine UML-Klasse?

Eine Klasse im Code muss nicht deckungsgleich sein mit der UML-Klasse!

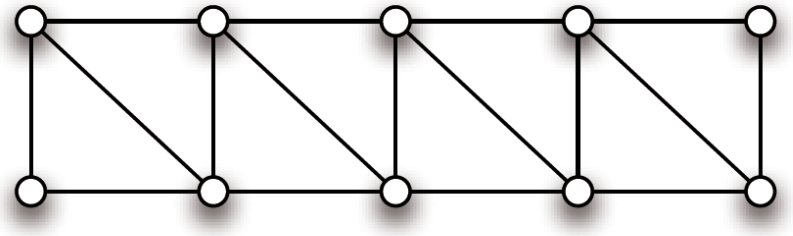
- Rein interne Attribute können oft im UML-Klassendiagramm ausgelassen werden wenn diese nicht für andere Klassen relevant sind.
- Manche Diagramme lassen Getter- und Setter-Methoden aus wenn angenommen wird dass alle Attribute Getter und Setter-Methoden haben.
- Vererbte Attribute und Methoden müssen nicht erneut in der Kindklasse dargestellt werden (dies kann aber zur Verdeutlichung gemacht werden).

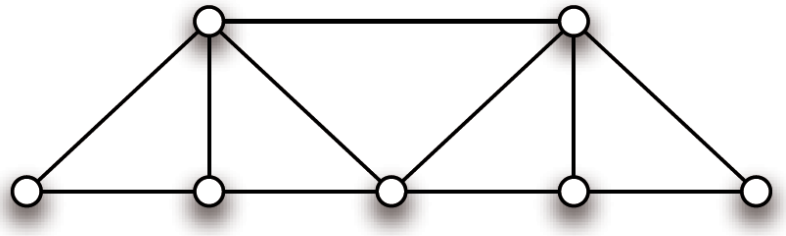
Objektorientierte Modellierung - UML

Beispiel Fachwerkträger

Eine Fachwerkträger besteht aus einer Menge von Elementen. Jedes Element wird durch zwei Knoten definiert

- Welche Klassen sind sinnvoll?
- Welche Beziehungen bestehen zwischen den Klassen?





Objektorientierte Modellierung - UML

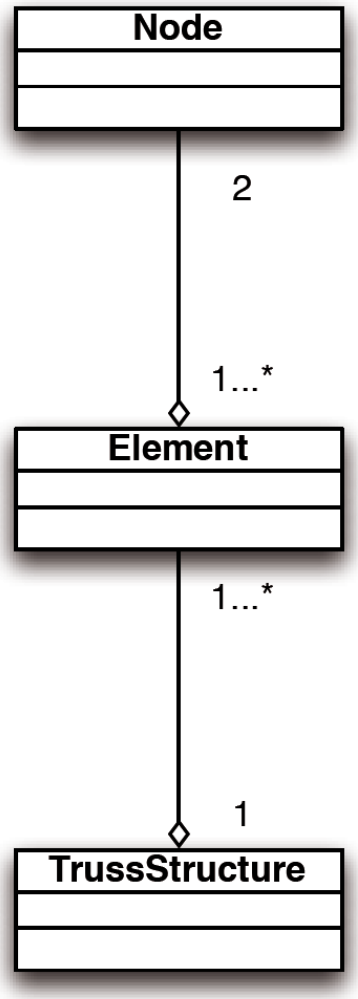
Gerichtete Beziehungen

Mit Hilfe einer Beziehung wird der Zugriff auf andere Objekte ermöglicht

- **Unidirektionale Beziehung**
  Zugriff auf ein Objekt ist nur in eine Richtung möglich

  z.B. nur die Elemente können auf die Knoten zugreifen
- **Bidirektionale Beziehung**
  Beide Objekte habe Zugriff auf das jeweils andere Objekt

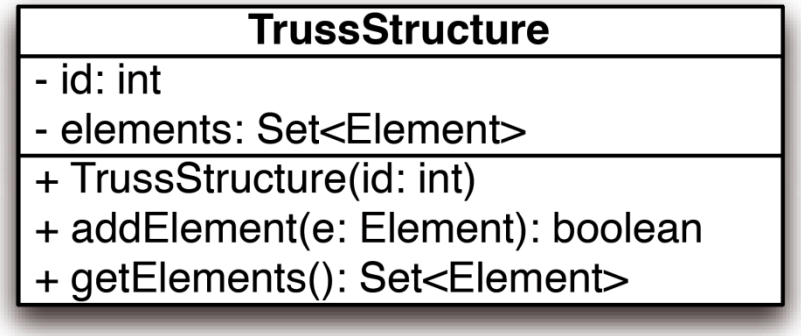
  z.B. der Fachwerkträger kann auf die Elemente und jedes Element kann auf den Fachwerkträger zugreifen



Objektorientierte Modellierung - UML

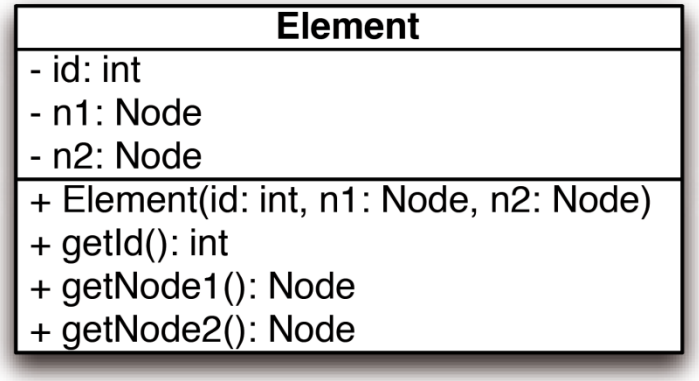
Klasse für Fachwerkträger

```
public class TrussStructure {
     // Eindeutiger Bezeichner
     private int id;
     // Menge von Elementen
     private Set<Element> elements;
     // Konstruktor
     public TrussStructure(int id) {
          this.id = id;
          this.elements = new HashSet<Element>();
     }
     // Element hinzufügen
     public boolean addElement(Element e) {
          return this.elements.add(e);
     }
     // Alle Elemente erfragen
     public Set<Element> getElements() {
          return this.elements;
     }
}
```



Objektorientierte Modellierung - UML

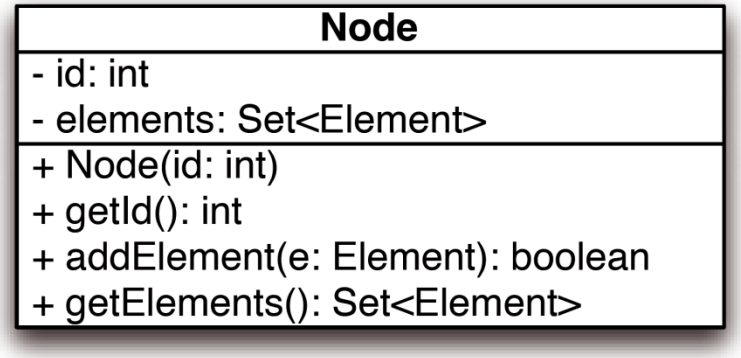
Klasse für Stabelemente




```
public class Element {
     // Eindeutiger Bezeichner
     private int id;
     // Erster Knoten
     private Node n1;
     // Zweiter Knoten
     private Node n2;
     // Konstruktor
     public Element(int id, Node n1, Node n2) {
          this.id = id; this.n1 = n1; this.n2 = n2;
     }
     // Bezeichner erfragen
     public int getId()      { return this.id; }
     // Ersten Knoten erfragen
     public Node getNode1() { return this.n1; }
     // Zweiten Knoten erfragen
     public Node getNode2() { return this.n2; }
}
```

Objektorientierte Modellierung - UML

Klasse für Knoten




```
public class Node {
    private int id;
    // Zugehörige Elemente
    private Set<Element> elements;
    // Konstruktor
    public Node(int id) {
        this.id = id;
        this.elements = new HashSet<Element>();
    }
    // Bezeichner erfragen
    public int getId() { return this.id; }
    // Element hinzufügen
    public boolean addElement(Element e) {
        return this.elements.add(e);
    }
    // Elemente erfragen
    public Set<Element> getElements() {
        return this.elements;
    }
}
```

Objektorientierte Modellierung - UML

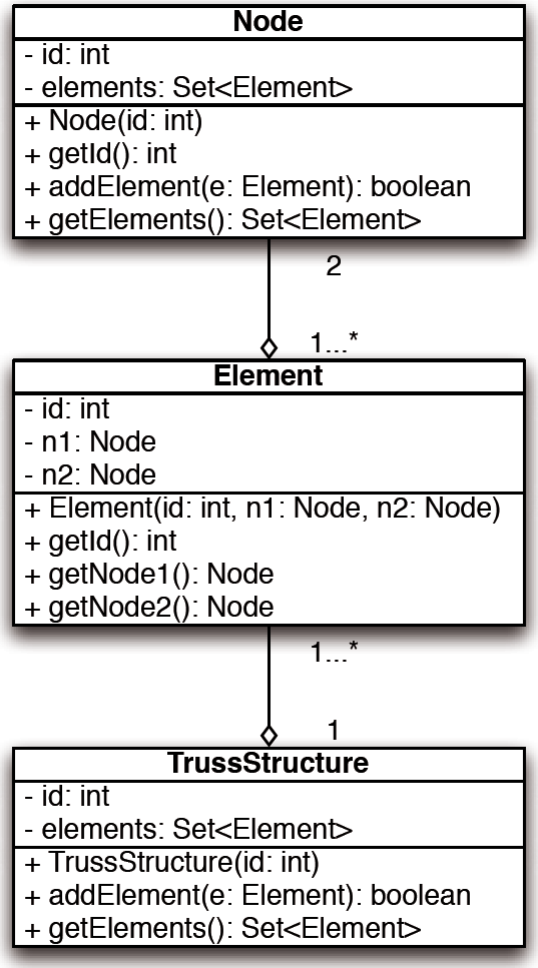
Demoprogramm

```
public class TrussStructureProgram {
    public static void main(String[] args) {

        Node n1 = new Node(1);
        Node n2 = new Node(2);
        Node n3 = new Node(3);

        Element e1 = new Element(10, n1, n2);
        Element e2 = new Element(11, n2, n3);

        TrussStructure truss = new TrussStructure(100);
        truss.addElement(e1);
        truss.addElement(e2);

        for (Element e : truss.getElements()) {
            System.out.println(e.getId());
        }
    }
}
```

Objektorientierte Modellierung - UML

Erweitertes Demoprogramm

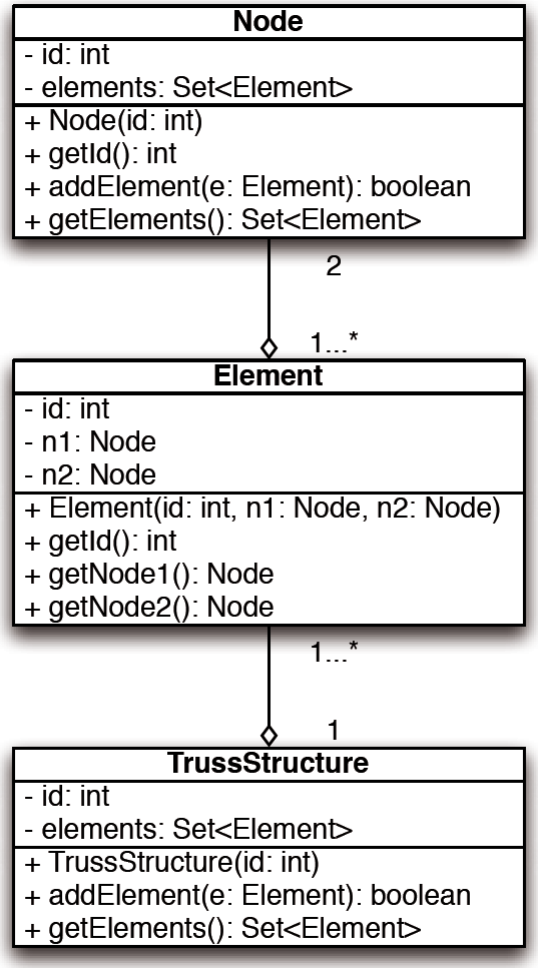
```
// ...
Node n1 = new Node(1);
Node n2 = new Node(2);
Node n3 = new Node(3);

Element e1 = new Element(10, n1, n2);
Element e2 = new Element(11, n2, n3);

n1.addElement(e1);
n2.addElement(e1);
n2.addElement(e2);
n3.addElement(e2);

System.out.println(e1.getNode1());

for (Element e : n2.getElements()) {
    System.out.println(e.getId());
}
```



Objektorientierte Modellierung - UML

Erweitertes Demoprogramm

```
// ...
Node n1 = new Node(1);
Node n2 = new Node(2);
Node n3 = new Node(3);

Element e1 = new Element(10, n1, n2);
Element e2 = new Element(11, n2, n3);
/*
n1.addElement(e1);
n2.addElement(e1);
n2.addElement(e2);
n3.addElement(e2);
*/
System.out.println(e1.getNode1());

for (Element e : n2.getElements()) {
    System.out.println(e.getId());
}
```



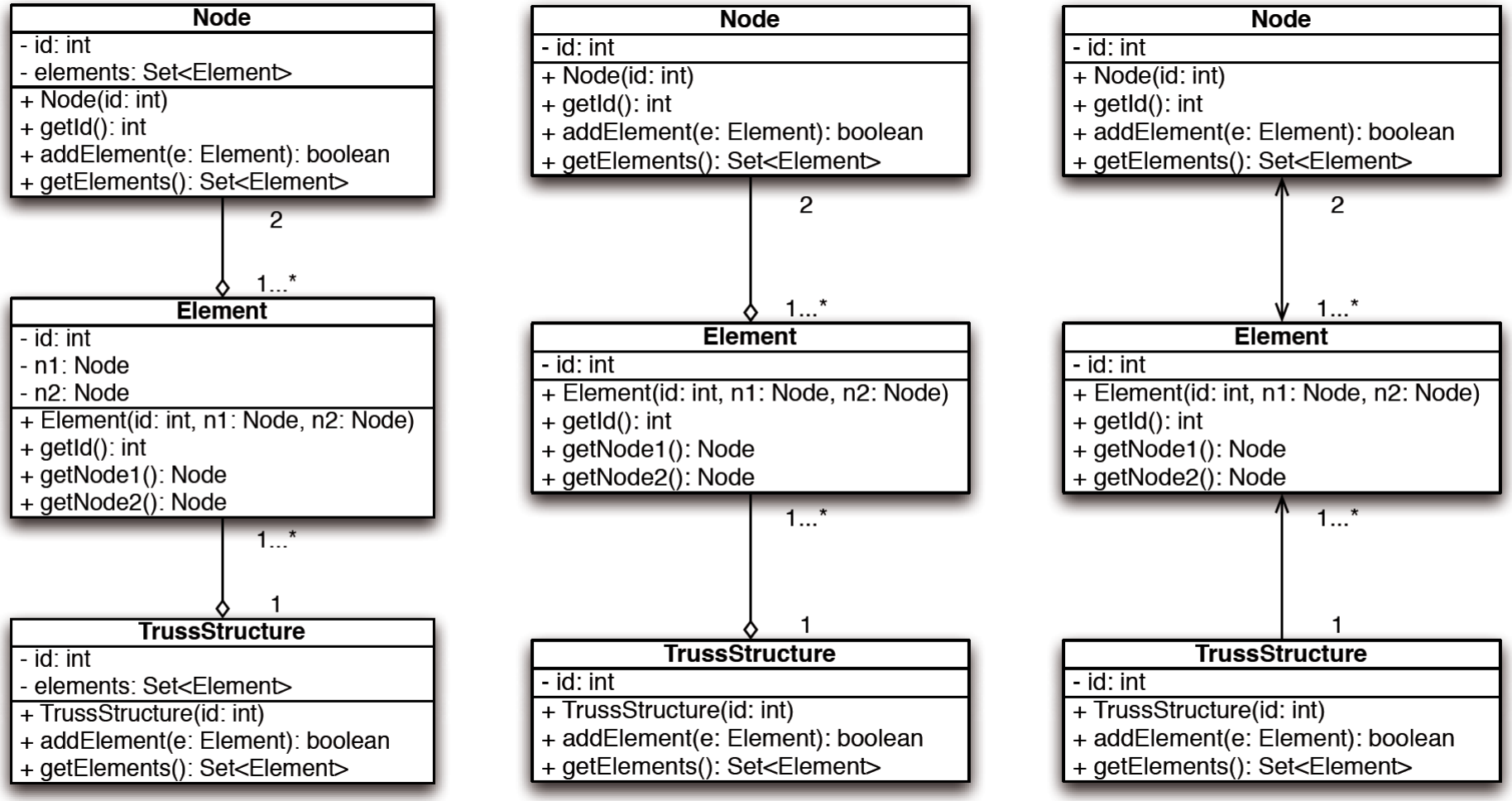
Objektorientierte Modellierung - UML

Erweiterung Klasse für Stabelemente

```
public class Element {

     private int id;
     private Node n1;
     private Node n2;

     // Konstruktor
     public Element(int id, Node n1, Node n2) {
          this.id = id;
          this.n1 = n1;
          this.n2 = n2;

          // Element anmelden
          this.n1.addElement(this);
          this.n2.addElement(this);
     }
     // ..
}
```

Objektorientierte Modellierung - UML

Vereinfachte UML Darstellung



Programmierung und Programmiersprachen

Sommersemester 2023

Prinzipien der Objektorientierte Modellierung

Objektorientierte Modellierung - Prinzipien

Kopplung & Bindung

- Architekturen werden häufig nach den Kriterien von **Kopplung** und **Bindung** bewertet
- Kopplung (*Coupling*) beschreibt die Abhängigkeit von Komponenten
  - Geringe Kopplung = Gut!
- Bindung (*Cohesion*) beschreibt den Fokus auf die Funktion einer Komponente
  - Hohe Bindung = Gut!
- Beide Kriterien lassen sich auf Methoden, Klassen, Pakete, und ganze Programme anwenden

Objektorientierte Modellierung - Prinzipien

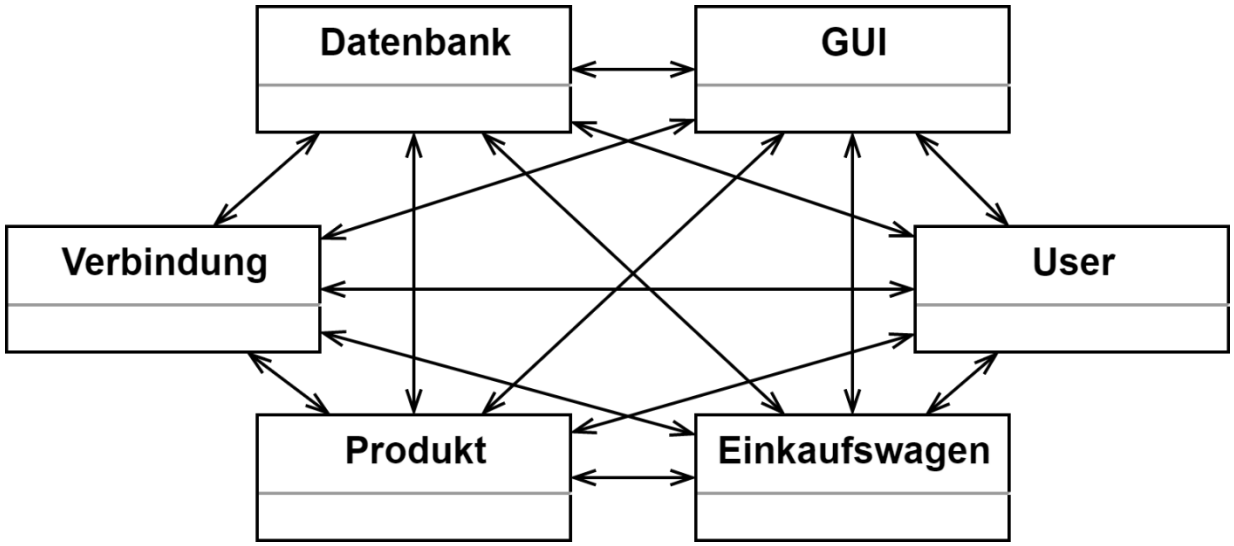
Kopplung *(Coupling)*

“Das Maß der Stärke der durch Verbindungen erzeugten Assoziationen von einem Modul zu einem anderen.” – Stevens et al., 1974

Objektorientierte Modellierung - Prinzipien

Kopplung *(Coupling)*

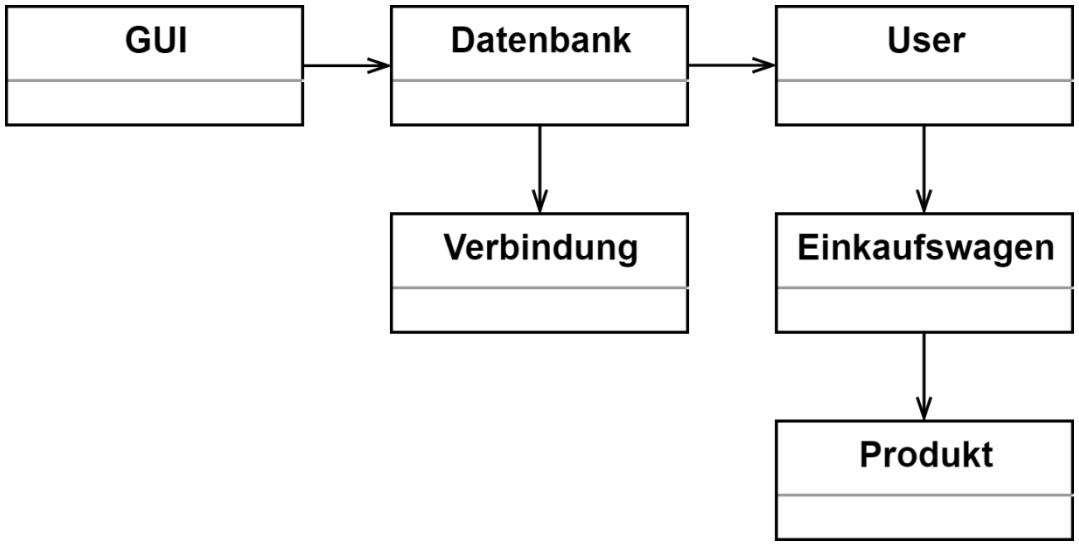
- Hohe Kopplung = viele Abhängigkeiten
  - Eine Klasse kann nicht einfach ersetzt werden
  - Veränderungen können Fehler an unerwarteten Stellen hinzufügen



Objektorientierte Modellierung - Prinzipien

Kopplung *(Coupling)*

- Niedrige Kopplung = wenige Abhängigkeiten
  - Wird eine Klasse verändert, müssen nur abhängige Klassen geprüft werden
  - Neue Elemente können einfach hinzugefügt werden



Objektorientierte Modellierung - Prinzipien

Kopplung *(Coupling)*

Wie erreicht man eine niedrige Kopplung?

- Vermeiden von globalen Variablen
- Datenkapselung
  - Attribute und Methoden wenn möglich `private` stellen (Geheimnisprinzip, *data hiding*)
  - Attribute nur über Getter- und Setter-Funktionen zugänglich machen
- Sinnvolle Schnittstellen/APIs planen (Modularität)

Objektorientierte Modellierung - Prinzipien

Bindung *(Cohesion)*

“Verbundheitsgrad zwischen Elementen eines einzelnen Moduls”

– Stevens et al., 1974

Objektorientierte Modellierung - Prinzipien

Bindung *(Cohesion)*

- Niedrige Bindung = eine Komponente hat viele unabhängige Funktionen
  - Führt zu Codeverdopplung
  - Schlechte Lesbarkeit
  - Besser: Methoden/Klassen/Module/… aufteilen

Beispiel: Niedrige Methodenbindung

```
public String berechnePreis(String produktID, boolean ermaessigt) {
    Produkt p = Datenbank.getInstance().getProdukt(produktID);
    double grundpreis = p.getPreis();
    double preisMitMwst = grundpreis * 1.19;
    double preisFinal = preisMitMwst * (ermaessigt ? 0.7 : 1.0);
    String preisString = preisFinal + "€";
    return preisString;
}
```

Objektorientierte Modellierung - Prinzipien

Bindung *(Cohesion)*

- Hohe Bindung = eine Komponente hat eine Funktion
  - Bessere Wiederverwendbarkeit und Lesbarkeit
  - *„Do One Thing and Do It Well“ – UNIX Philosophy*
  - Hohe Bindung korreliert mit niedriger Kopplung

```
public Produkt getProdukt(String produktID) {
    return Datenbank.getInstance().getProdukt(produktID);
}
public double berechneMwst(double grundwert) {
    return grundpreis * MWST_PROD;
}
public double berechneErmaessigt(double grundpreis) {
    return grundpreis * this.ermaessigung;
}
public String getPreisString(double preis) {
    return preisFinal + "€";
}
public double getProduktPreis(String produktID, boolean ermaessigt) {
    double grundpreis = getProdukt(produktID).getPreis();
    return berechneErmaessigt(berechneMwst(grundpreis), ermaessigt);
}
```

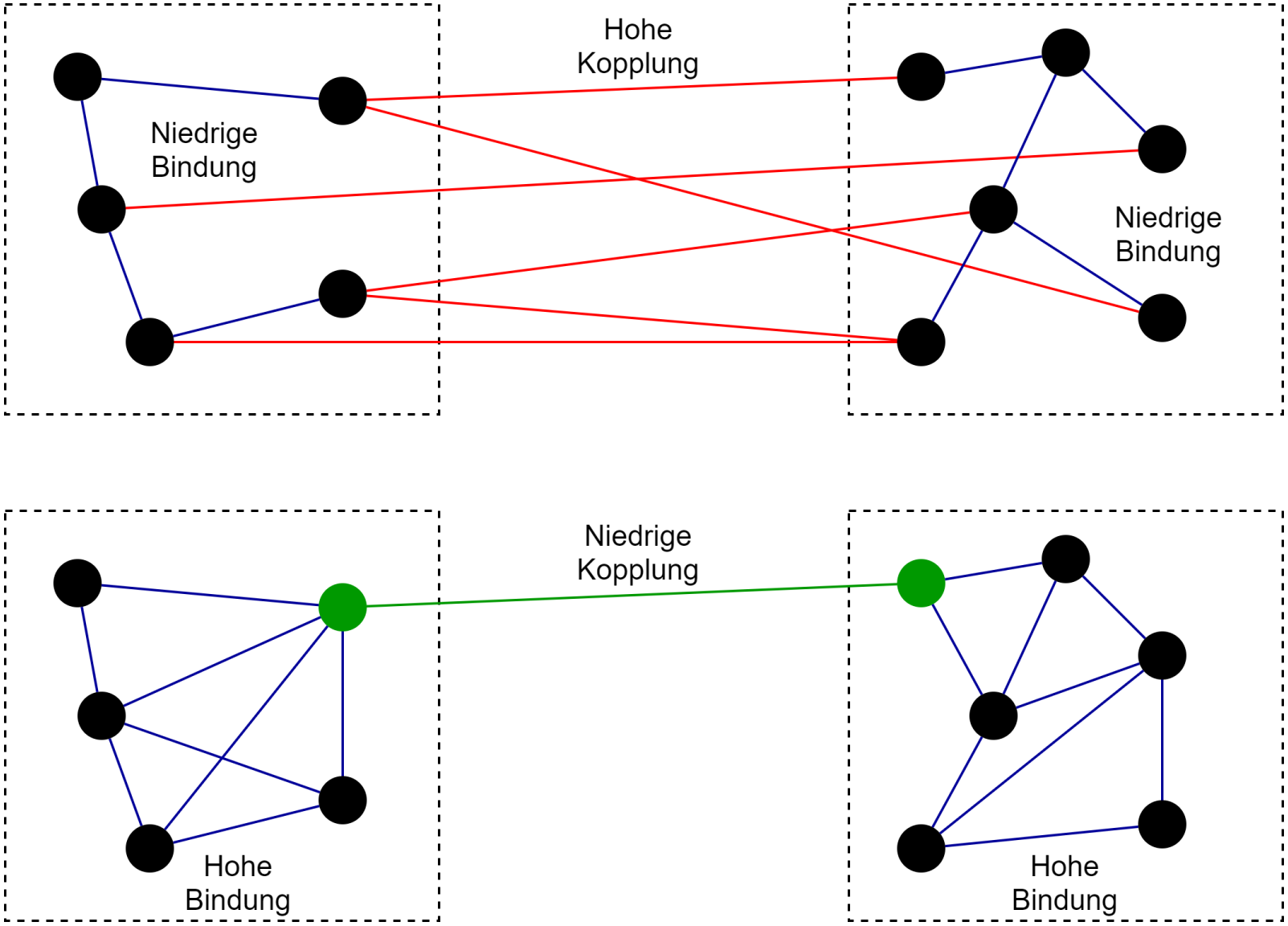
Objektorientierte Modellierung - Prinzipien

Bindung *(Cohesion)*

Wie erreicht man eine hohe Bindung?

- Methoden, die mehr als eine Funktion ausführen, aufteilen
- Bestehende Methoden wiederverwenden (*Code reuse*)
- Funktional ähnliche Klassen abstrahieren und zu Modulen/Paketen zusammenfassen

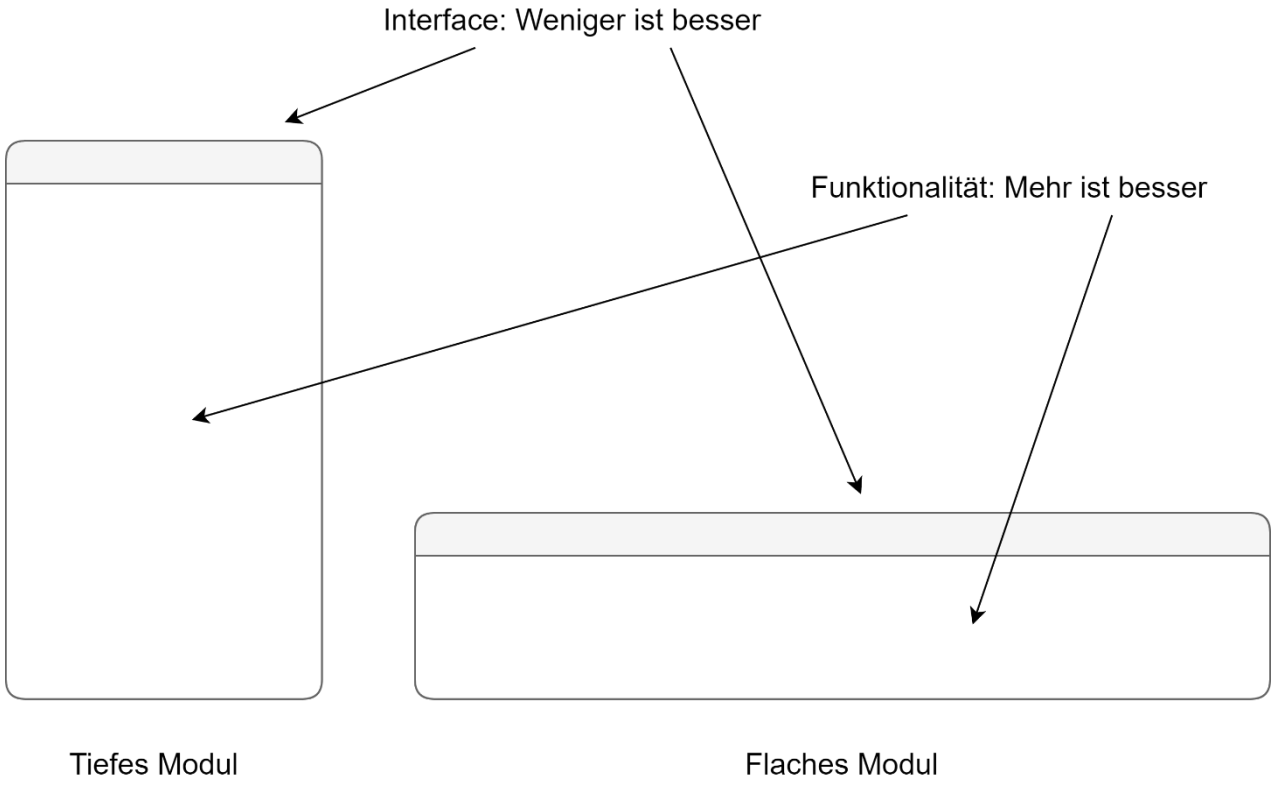
Objektorientierte Modellierung - Prinzipien



Objektorientierte Modellierung - Prinzipien

Modultiefe

Module sollten hohe Funktionalität bieten mit kleinen Interfaces (tiefes Modul)



Aus „A Philosophy of Software Design“, J. Ousterhout (2018)

Objektorientierte Modellierung - Prinzipien

Codewartung

- Die meisten Projekte sind keine Einzelarbeit
- Einzelne Personen haben nie den Überblick über das gesamte Projekt
  - Schnittstellen verständlich halten
  - Geheimnisprinzip anwenden
  - Eine Komponente sollte verständlich sein ohne die innere Arbeitsweise zu kennen

Objektorientierte Modellierung - Prinzipien

Codewartung

- Sinnvolle, aussagekräftige Bezeichner verwenden
  - Zu lange Namen sind auch unübersichtlich!
- *Sinnvolle* Kommentare
- Projekt- und Sprachkonventionen einhalten
- Kopplung und Bindung beachten

Merke: Eine Codezeile wird häufiger gelesen als geschrieben

Objektorientierte Modellierung - Prinzipien

Literatur

- Freeman, Eric (2005) “Entwurfsmuster von Kopf bis Fuß”, *O'Reilly*
- Gamma et al. (1994) “Design Patterns. Elements of Reusable Object-Oriented Software.”, *Prentice Hall*
- Ousterhout, John (2018), “A Philosophy of Software Design”, *Yaknyam Press*
- Balzert, Helmut (2011) “Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb”, *Springer*
- Martin, Robert C. (2009) “Clean Code: Refactoring, Patterns, Testen und Techniken für sauberen Code”, *mitp*