3.1 Statische Sicht

- Basis der UML, auf der alle anderen Konstrukte aufbauen

3.1.1 Elemente der statischen Sicht

3.1.2 Beziehungen zwischen den Elementen der statischen Sicht

3.1.1 Elemente der statischen Sicht

- Klassen
- Schnittstellen
- Datentypen

- Modellierung von Konzepten, die für die Applikation von Bedeutung sind
- das „Vokabular“ des Systems

Klassen (Classes)

- Beschreibung einer Menge von Objekten, welche die gleichen Attribute, Operationen, Beziehungen und Semantik haben
- **Graphische Darstellung**:

Klassen - Findungsprozess

- Identifikation von Dingen, die Benutzer und Entwickler zur Beschreibung des Problems und der Lösung benutzen
- Hilfsmittel:
  - CRC-Karten
    (CRC = Class, Responsibility, Collaborations)
  - Anwendungsfall-basierte Analyse

Klassen - Findungsprozess (Forts.)

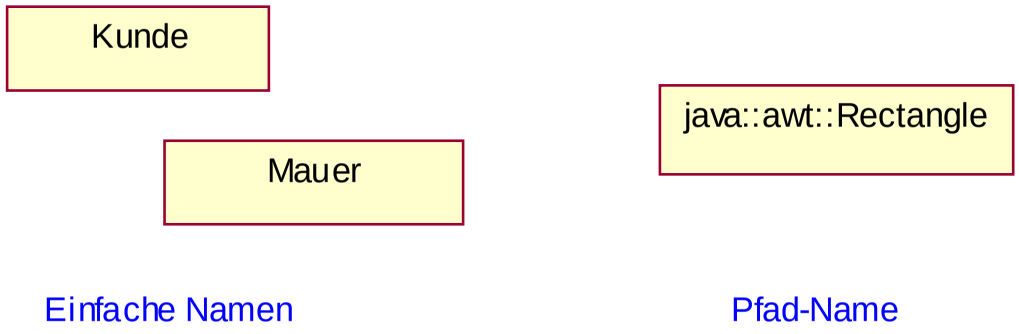
- Für jede Abstraktion eine *Verantwortlichkeit festlegen*; auf Ausgewogenheit achten
- *Festlegung* der *Attribute* und *Operationen*, die benötigt werden, um die Verantwortlichkeit zu erfüllen

Klassen - Namen

- Jede Klasse muss einen *eindeutigen* Namen haben, der sie von anderen Klassen unterscheidet

Klassen - Namen (Forts.)

- Man unterscheidet
  - *Einfache Namen* (Simple names)
  - *Pfad-Namen* (Path names)
    - Name des Paketes, in dem sich die Klasse befindet, wird vorangestellt
    - die Zeichen :: trennen Paket- und Klassennamen



Klassen - Namen (Forts.)

Konvention:

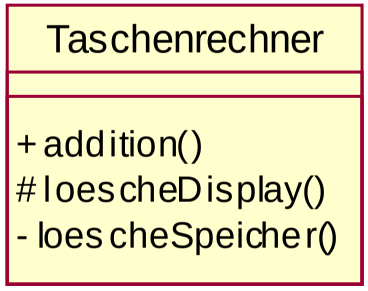
- Klassennamen sind i.a. Nomen und beginnen mit einem Großbuchstaben
  z.B. `Kunde`, `Mauer`
- Bei zusammengesetzten Namen beginnen die Teilwörter jeweils auch mit einem Großbuchstaben,
  z.B. `NetworkController`

Klassen - Attribute und Operationen - Sichtbarkeit (Visibility)

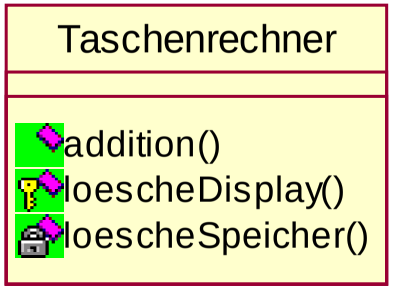
- *Sichtbarkeit* (Visibility) = Festlegung der Zugriffsrechte anderer Klassen auf die Attribute bzw. Operationen einer Klasse
  - *public* (+)
    Jede beliebige Klasse darf zugreifen
  - *protected* (#)
    Nur die definierende Klasse selbst und von dieser abgeleitete Klassen dürfen zugreifen
  - *private* (-)
    Zugriff nur durch die definierende Klasse

Klassen - Attribute und Operationen - Sichtbarkeit (Forts.)

Graphische Darstellung:



Standard



in Rational Rose

Klassen - Attribute

- *Attribut* (Attribute) = mit Namen versehene *Eigenschaft* einer Klasse
- Klasse kann beliebig viele (auch keine) Attribute besitzen
- **Konvention**:
  - Attributnamen sind i.a. Nomen und beginnen mit einem Kleinbuchstaben, z.B. `hoehe`, `breite`
  - Zusammengesetzte Namen analog zu Klassen, z.B. `anzahlSpalten`

Klassen - Attribute (Forts.)

- Verschiedene Detaillierungsgrade möglich
- **Generelle Syntax**:

```
[Sichtbarkeit] Name [[Multiplizität]][:Typ]
  [= initialer Wert][{Eigenschaft}]
```

- **Beispiele**:

`ursprung`	Name
`+ ursprung`	Sichtbarkeit und Name
`ursprung : Punkt`	Name und Typ
`zahlenListe [0 .. 10] : Integer`	Name, Multiplizität, Typ
`zaehler : Integer = 0`	Name, Typ, Initialisierung
`id : Integer = 4711 {frozen}`	Name, Typ, Eigenschaft

Klassen - Attribute (Forts.)

Drei Eigenschaften für Attribute

- `changeable`

  keine Restriktion (Standard)

- `addOnly`

  für Attribute mit Multiplizität größer als 1 können Werte hinzugefügt, jedoch nicht mehr verändert oder entfernt werden

- `frozen`

  Wert darf nicht mehr geändert werden, nachdem das Objekt initialisiert ist

Klassen - Operationen

- *Operation* = mit Namen versehenes *Verhalten* einer Klasse, das für jedes Objekt dieser Klasse angefordert werden kann
- Operationen können den Zustand eines Objektes verändern
- **Konvention**:
  - Operationennamen sind i.a. Verben und beginnen mit einem Kleinbuchstaben, z.B. `hinzufuegen()`
  - Zusammenges. Namen analog zu Klassen, `istLeer()`

Klassen - Operationen (Forts.)

- Verschiedene Detaillierungsgrade möglich
- **Generelle Syntax**:

```
[Sichtbarkeit] Name [(Parameter-Liste)]
  [:Rückgabe-Typ][{Eigenschaft}]
```

- **Beispiele**:

`display`	Name
`+ display`	Sichtbarkeit und Name
`setze(n:Name, s:String)`	Name und Parameter
`getId():Integer`	Name, Parameter, Rückgabe-Typ

Klassen - Operationen (Forts.)

- *Signatur* = `Name [(Parameter-Liste)]`
- Einzelnen Parameter der Signatur haben die folgende Syntax:

```
[Richtung] Name : Typ [= Standard-Wert]
```

Klassen - Operationen (Forts.)

- Richtungen

  `in`

  Eingabe-Parameter; darf nicht geändert werden

  `out`

  Ausgabe-Parameter; Übermittlung von Information zum Aufrufer

  `inout`

  Eingabe-Parameter, der modifiziert werden kann

Klassen - Operationen (Forts.)

- Eigenschaften:
  - `leaf`

    Operation darf nicht überschrieben werden
    (Java: `final`, C++: nicht-virtuelle Funktion)
  - `isQuery`

    Ausführung der Operation ändert den Systemzustand nicht (keine Seiteneffekte)
    (C++: `const` Funktion)
  - `sequential`, `guarded`, `concurrent`

    Behandlung später bei aktiven Klassen

Klassen - Attribute und Operationen - Geltungsbereich (Scope)

- Zwei verschiedenen *Bezugsrahmen* (Scope) für Attribute und Operationen einer Klasse
  - `instance`

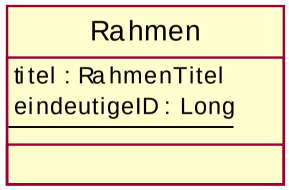
    jedes Objekt der Klasse hat ein eigenes Exemplar (Standard)
  - `classifier`

    es gibt nur ein Exemplar für alle Objekte der Klasse (C++ und Java: `static` Attribute und Operationen)

Klassen - Attribute und Operationen - Geltungsbereich (Forts.)

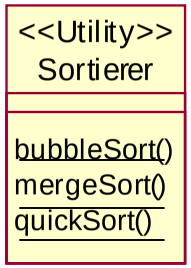
Graphische Darstellung:

– Attribute und Methoden mit Geltungsbereich `classifier` sind unterstrichen



Klassen - Attribute und Operationen - Geltungsbereich (Forts.)

- *Utility*-Klassen = Klassen, die ausschliesslich classifier Attribute und Operationen haben
- Verwendung:
  - Zusammenfassung von Hilfsoperationen, die sich keiner anderen Klasse zuordnen lassen
  - z.B. Klasse mit Sortieralgorithmen

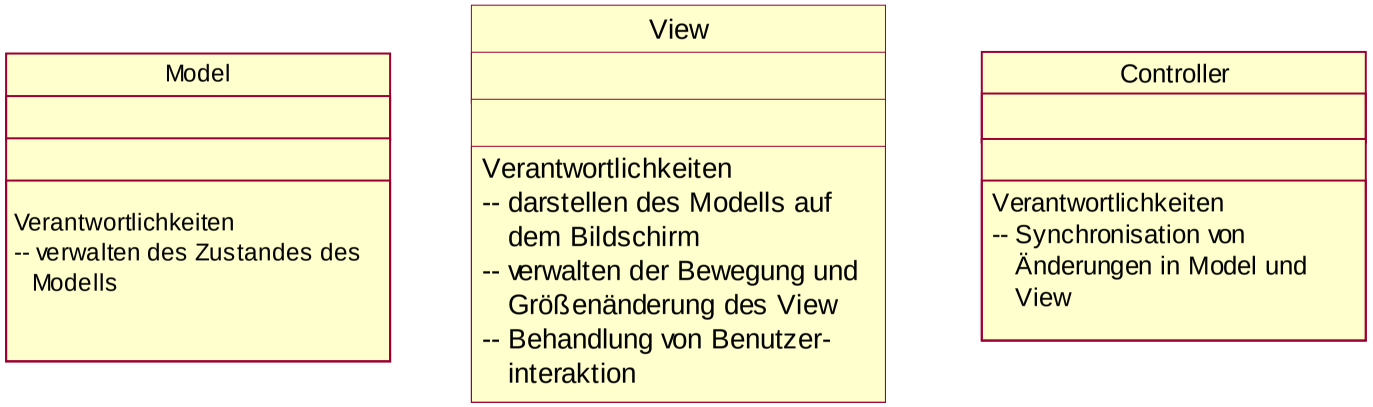


Klassen - Verantwortlichkeiten (Responsibilities)

- In der Analyse-Phase werden Klassen durch ihre *Verantwortlichkeiten* (Responsibilities) charakterisiert
- Erst in der Entwurfs- und Implementationsphase werden Attribute und Operationen festgelegt, um die speziellen Verantwortlichkeiten zu bewerkstelligen

Klassen - Verantwortlichkeiten (Forts.)

- **Graphische Darstellung**:
  - als Notiz
  - als eigener Abschnitt in einem Klassensymbol
- **Beispiel**: MVC Verantwortlichkeiten



Klassen - Objekte

- *Objekt* (Object) = Instanz einer Klasse; diskrete Einheit mit Identität und Zustand
- Zwei Perspektiven für Objekte:
  - „Schnappschuss“ (Snapshot) -Darstellung des Systems zu einem bestimmten Zeitpunkt (Objekt-Diagramm)
  - Einheit mit Identität, die im Laufe der Zeit verschiedenen Zustände annimmt (Zustandsüberg.-, Sequenz-, Kollaborationsdiagramm)

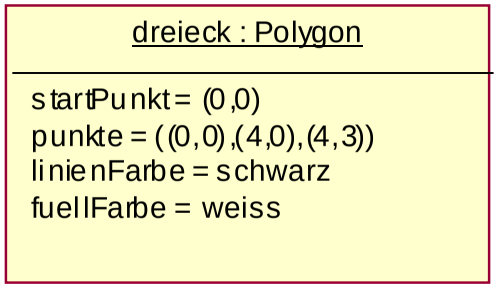
Klassen - Objekte (Forts.)

Graphische Darstellung:

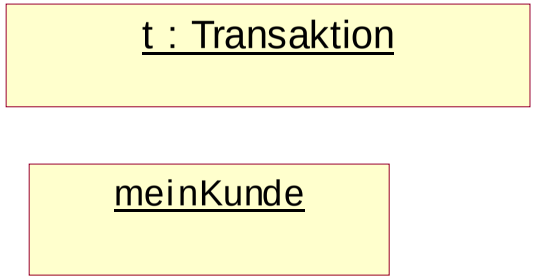
- Darstellungssymbol wie bei Klassen
- Oberstes Fach enthält
  `Objektname:Klassenname`
- nur ein Fach für Attribute und deren Werte
  `Attributname:Typ = Wert`
- kein Fach für Operationen, da diese durch zugehörige Klasse vorgegeben sind

Klassen - Objekte (Forts.)

- **Beispiel**:



- Benannte Objekte



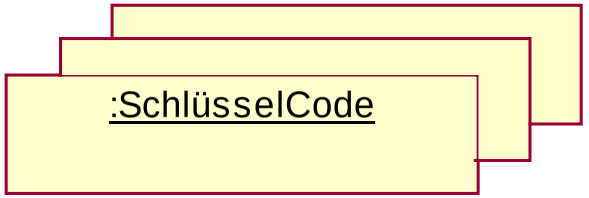
falls zugehörige Klasse klar ist

Klassen - Objekte (Forts.)

- Anonyme Objekte

z.B. bei impliziter Instanziierung von Klassen

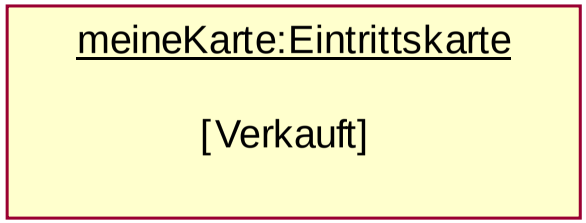
- Multiobjekte



für Darstellung großer Mengen von Objekten derselben Klasse

Klassen - Objekte (Forts.)

- Objekte mit explizitem Zustand



z.B. wenn Zustandsmaschine zur Klasse assoziiert ist

Klassen - Abstrakte Klassen und Operationen

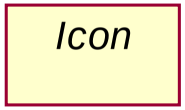
- In komplexen Klassenhierarchien werden üblicherweise gewisse Klassen als *abstrakt* (abstract) spezifiziert
- *Abstrakte Klasse* = Klasse, von der es keine direkten Objekte geben darf

  (Java: `abstract`, C++: mind. eine rein virtuelle Funktion)
- Zur Nutzung der Funktionalität von abstrakten Klassen leitet man eigene Klassen von diesen ab

Klassen - Abstrakte Klassen und Operationen (Forts.)

Graphische Darstellung:

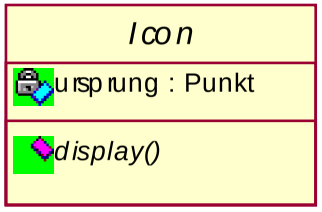
– Klassenname wird kursiv geschrieben



Klassen - Abstrakte Klassen und Operationen (Forts.)

- Ebenso lassen sich Operationen als abstrakt spezifizieren

  (Java: `abstract`, C++: rein virtuelle Funktion)
- Abstrakte Operationen müssen in abgeleiteten Klassen implementiert werden
- **Graphische Darstellung**:
  - Operationsname wird kursiv geschrieben



Klassen - Abstrakte Klassen und Operationen (Forts.)

- Für Klassen läßt sich über Einschränkungen festlegen, ob sie in einer Klassenhierarchie Eltern bzw. Kinder haben dürfen
  - `leaf`

    Klasse darf keine Kinder haben, d.h. nicht abgeleitet werden (Java: `final`)
  - `root`

    Klasse darf keine Generalisierung haben (Basisklasse)

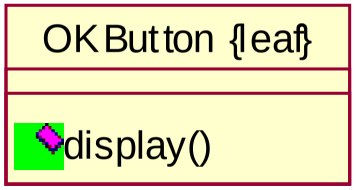
Klassen - Abstrakte Klassen und Operationen (Forts.)

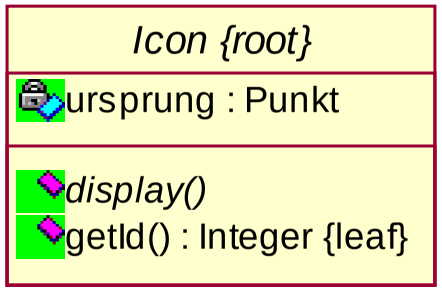
- Operationen sind typischerweise *polymorph*, d.h. sie können in abgeleiteten Klassen überschrieben werden
- Wird eine Operation auf einem Objekt aufgerufen, so wird zur Laufzeit abhängig vom Typ des Objektes die richtige Methode verwendet
- Einschränkung `leaf` verbietet das Überschreiben der Operation

  (Java: `final`, C++: nicht virtuelle Funktion)

Klassen - Abstrakte Klassen und Operationen (Forts.)

- **Beispiele**:





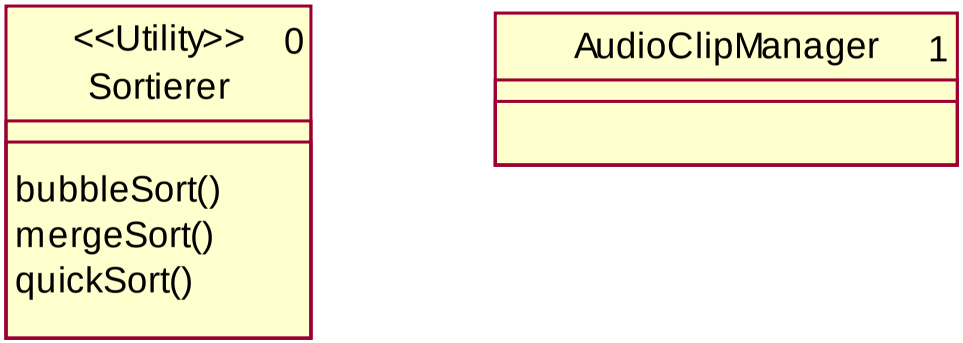
Klassen - Multiplizität (Multiplicity)

- Manchmal ist es nützlich, die Anzahl der Objekte, die von einer Klasse instanziiert werden dürfen, zu beschränken.
- Anzahl der Objekte, die Klasse haben darf, heißt ihre *Multiplizität* (Multiplicity)
- **Beispiel**:

0 Instanzen	Utility Klassen
1 Instanz	Singleton Klassen
n Instanzen	
beliebig viele Instanzen	Standard

Klassen - Multiplizität (Forts.)

- **Beispiele**:



Exkurs: Entwurfsmuster (Design Patterns)

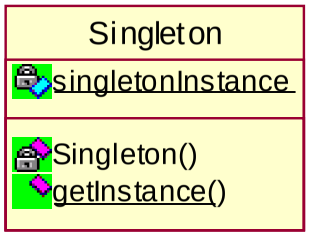
- *Entwurfsmuster* (Design Pattern) = abstrakte Lösung für ein häufig auftauchendes Problem in der OO Modellierung
- Standardwerke:
  - E. Gamma, R. Helm, R. Johnson, J. Vlissides „Design Patterns: Elements of Reusable Object-Oriented Software“ Addison Wesley, 1995
  - M. Grand „Patterns in Java“ Wiley, 1998

Entwurfsmuster - Singleton

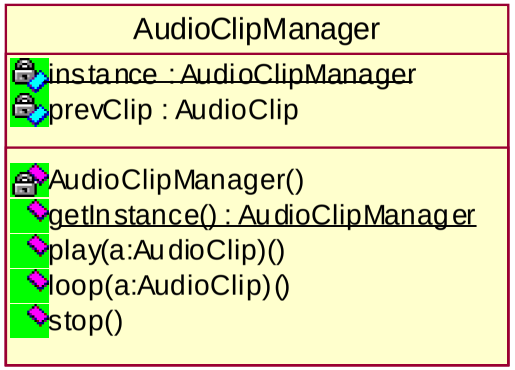
- *Singleton* Entwurfsmuster gewährleistet, dass nur eine Instanz einer Klasse kreiert werden kann
- Anwendung:
  - Klasse übernimmt die Verwaltung einer Ressource (z.B. Printer Spooler, Window Manager, ...)

Entwurfsmuster - Singleton (Forts.)

- Abstrakte Modellierung:



- Konkrete Modellierung für AudioClipManager:



Entwurfsmuster - Singleton (Forts.)

- Implementation in Java:

```
public class AudioClipManager implements AudioClip{
   private static AudioClipManager m_instance =
        new AudioClipManager();
   private AudioClip m_prevClip; // vorheriger AudioClip
   private AudioClipManager(){}
   public static AudioClipManager getInstance(){
        return m_instance;
   }
```

Entwurfsmuster - Singleton (Forts.)

- Implementation in Java (Forts.):

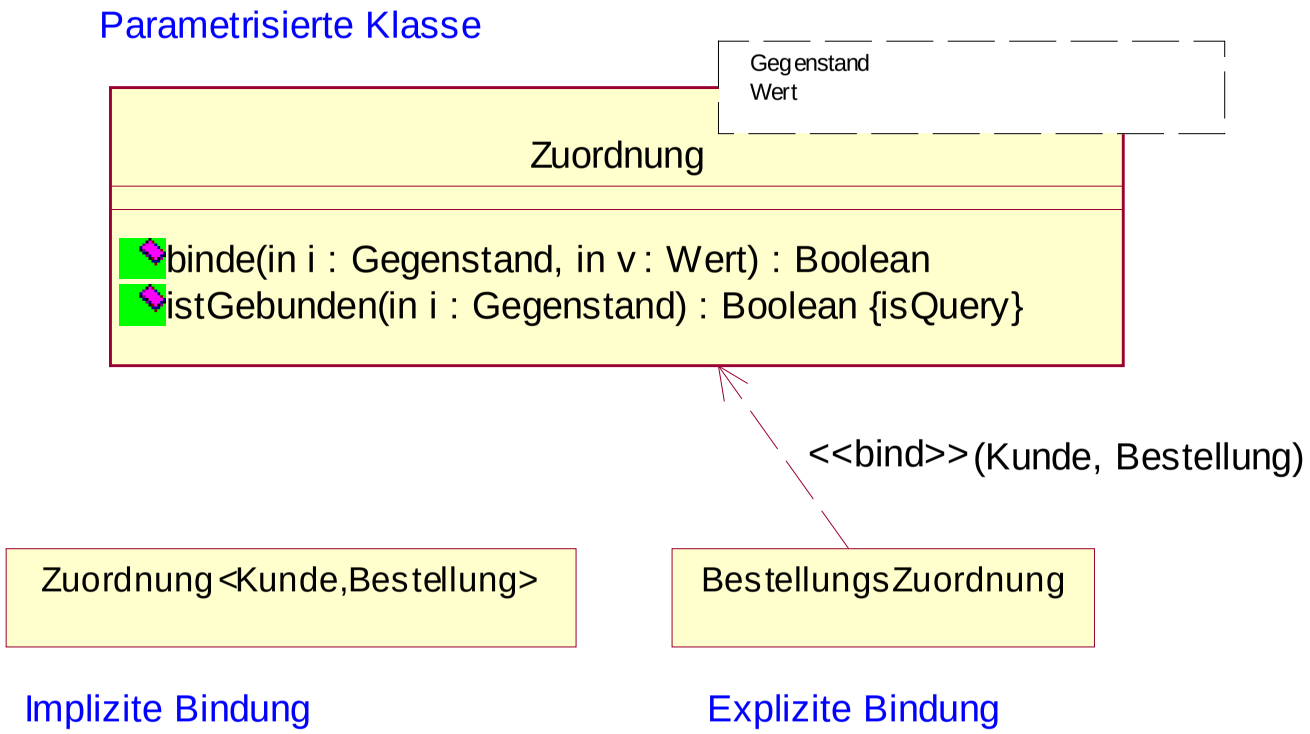
```
public void play (AudioClip clip){
    if (m_prevClip != null)
        m_prevClip.stop();
    m_prevClip = clip;
    clip.play;
}
...
} // class AudioClipManager
```

Klassen - Parametrisierte Klassen (Templates)

- *Parametrisierte Klasse* (Template) = Klasse mit einem oder mehreren formalen Parametern; Familie von Klassen
- Durch *Bindung* von Parametern an parametrisierte Klassen erhält man konkrete Klassen

Klassen - Parametrisierte Klassen (Forts.)

- **Beispiel**: Parametrisierte Klasse *Zuordnung*



Klassen - Parametrisierte Klassen (Forts.)

- **Beispiel (Forts.)**: Realisierung in C++

```
template<class Gegenstand, class Wert>
class Zuordnung {
public:
   virtual Boolean binde(const Gegenstand&, const Wert&);
   virtual Boolean istGebunden(const Gegenstand&) const;
   ...
};

bestellungsZuordnung = Zuordnung<Kunde, Bestellung>;
```

Klassen - Aktive Klassen, Aktive Objekte

- Moderne Betriebssysteme ermöglichen *Multitasking*, d.h. nebenläufige Ausführung verschiedener Kontrollflüsse
- Man unterscheidet
  - *Prozess* = Kontrollfluss mit eigenem Adressraum, der nebenläufig zu anderen Prozessen abläuft
  - *Thread* = Kontrollfluss innerhalb eines Prozesses, der nebenläufig zu anderen Threads desselben Prozesses abläuft (gemeinsamer Adressraum)

Klassen - Aktive Klassen, Aktive Objekte (Forts.)

- Modellierung realer Systeme erfordert oftmals eine *Prozeßsicht* von Klassen und Objekten
- *Aktives Objekt* = Objekt, dessen Kontrollfluss in einem eigenen Prozess oder Thread ausgeführt wird
- *Aktive Klasse* = Klasse, deren Instanzen aktive Objekte sind
- „Normale“ Klassen werden als *passiv* bezeichnet

Klassen - Aktive Klassen, Aktive Objekte (Forts.)

Graphische Darstellung:

- wie Objekt bzw. Klasse
- mit fetter Umrandung

- Stereotypen für aktive Klassen
  - `process`
  - `thread`

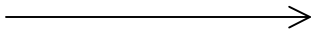
Klassen - Aktive Klassen, Aktive Objekte - Kommunikation

- Objekte kommunizieren untereinander durch Austausch von Nachrichten
- 4 Möglichkeiten für System mit aktiven und passiven Objekten:
  - Passiv → Passiv
    entspricht einem normalen Aufruf einer Operation eines Objektes

Klassen - Aktive Klassen, Aktive Objekte - Kommunikation (Forts.)

– Aktiv → Aktiv

Interprozesskommunikation mit zwei verschiedenen Aufrufmöglichkeiten:

- *synchron* (Darstellung: )
  Aufrufer wartet auf Ausführung der Operation (Rendezvous-Semantik)
- *asynchron* (Darstellung: )
  Aufrufer tätigt Aufruf oder setzt Signal ab und fährt in seinem Kontrollfluss fort (Mailbox-Semantik)

Klassen - Aktive Klassen, Aktive Objekte - Kommunikation (Forts.)

- Aktiv → Passiv

  Normaler Aufruf. Problematisch, wenn mehrere aktive Objekte dasselbe Zielobjekt benutzen
  ⇒ Synchronisationsmechanismen verwenden

- Passiv → Aktiv

  passives Objekt ist letztlich verwurzelt in aktivem Objekt ⇒ 2. Fall

Klassen - Aktive Klassen, Aktive Objekte - Synchronisation

Problem:

- Mehrere aktive Objekte greifen zur selben Zeit auf dasselbe Zielobjekt zu
  $\Rightarrow$ Zustand des Zielobjektes wird (wahrscheinlich) inkonsistent

Lösung:

- Realisierung von *wechselseitigem Ausschluss* (Mutual Exclusion), d.h. zu jedem Zeitpunkt darf nur ein aktives Objekt auf das Zielobjekt zugreifen

Klassen - Aktive Klassen, Aktive Objekte - Synchronisation (Forts.)

- Modellierung in UML durch Zuweisung von Synchronisationseigenschaften an Operationen
  - `sequential`

    keine direkte Synchronisation; Koordination muss außerhalb erfolgen (Standard)
  - `guarded`

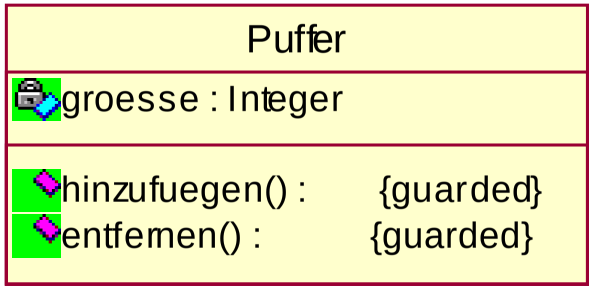
    Sequentialisierung der Aufrufe, d.h. zu einem Zeitpunkt kann nur exakt eine Operation auf dem Objekt aufgerufen werden (Java: `synchronized`)

Klassen - Aktive Klassen, Aktive Objekte - Synchronisation (Forts.)

- `concurrent`

  jede mit concurrent versehene Operation kann nebenläufig ausgeführt werden

- **Beispiel**: Zugriff auf einen gemeinsam genutzten Puffer



Schnittstellen (Interfaces)

- *Schnittstelle* (Interface) = eine Menge von Operationen, die ein bestimmtes Verhalten (unabhängig von einer Implementierung) vorschreiben
- Sorgfältige und frühzeitige Festlegung von Schnittstellen erlaubt, dass Systembestandteile (relativ) *unabhängig* voneinander entwickelt werden können

Schnittstellen (Forts.)

- Schnittstellen werden von Klassen oder Komponenten *realisiert*
- Realisierung heißt, dass alle spezifizierten Operationen ordnungsgemäß implementiert werden müssen

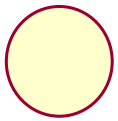
Schnittstellen (Forts.)

- Schnittstellen lassen sich durch ein *Client/Server-Verhalten* charakterisieren
  - *Client-Seite*
    benutzt Operationen, ohne sich um die Implementation zu kümmern
  - *Server-Seite*
    implementiert Operationen;
    Modifikation, Austausch oder sogar Kauf der Implementation möglich, solange Schnittstellen-Verhalten realisiert wird

Schnittstellen (Forts.)

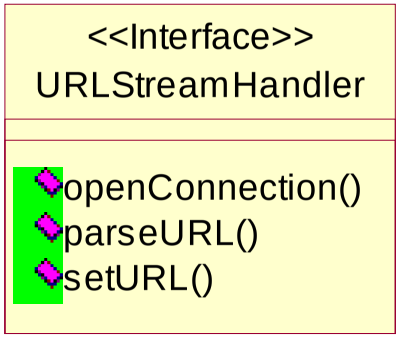
Graphische Darstellung:

- „Lolipop“-Notation oder stereotypisierte Klassennotation
- Namen und Operationen wie bei Klassen, jedoch keine Attribute



URLStreamHandler

Lolipop-Notation



Stereotypisierte Klassennotation

Schnittstellen (Forts.)

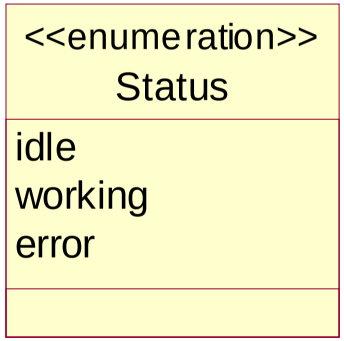
- Konzept der Schnittstellen wird in Java unterstützt (`interface`)
- Die Funktionsweise von CORBA basiert auf der Schnittstellen-basierten Sprache *CORBA-IDL*
  - CORBA = Common Object Request Broker Architecture
  - IDL = Interface Definition Language

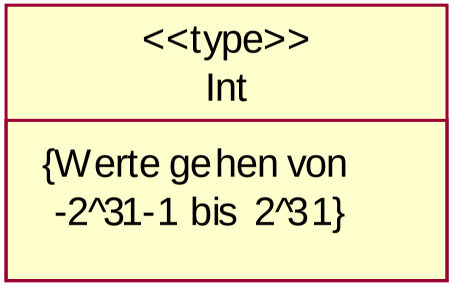
Datentyp (Data type)

- *Datentyp* (Data Type) = Beschreibung einer Menge von Werten, die jedoch keine Identität haben
  - Einfache vordefinierte Typen
    z.B. `Integer`, `String`, `Boolean`
  - Benutzerdefinierte Typen
    Aufzählungen (Enumerations)
- Basis, auf der Klassen und Schnittstellen definiert werden

Datentyp (Forts.)

Graphische Darstellung:



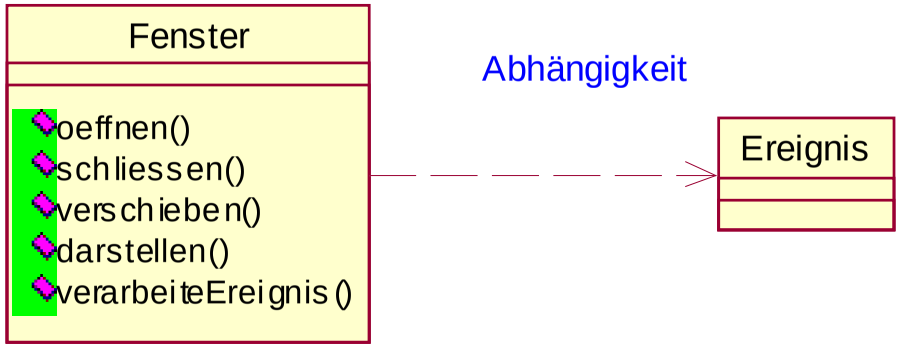


3.1.2 Beziehungen zwischen den Elementen der statischen Sicht

- Es gibt vier Arten von Beziehungen in UML
  - Abhängigkeit (Dependency)
  - Generalisierung (Generalization)
  - Assoziation (Association)
  - Realisierung (Realization)

Abhängigkeit (Dependency)

- *Abhängigkeit* (Dependency) = „benutzt“-Beziehung zwischen Elementen
- **Graphische Darstellung**:
  - gestrichelte Linie mit Pfeilspitze, die vom abhängigen zum unabhängigen Element zeigt



Abhängigkeit - Erweiterungen

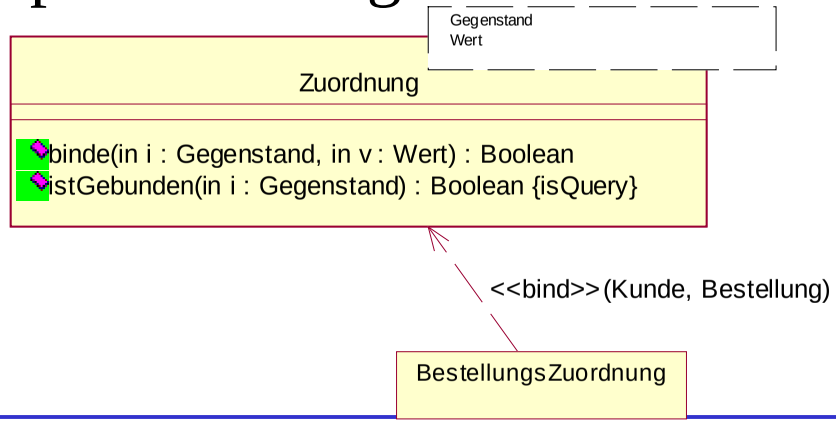
- In der Regel reicht einfache Anwendung von Abhängigkeiten
- Bei fortschreitendem Entwicklungsprozess jedoch genauere Spezifikation der Abhängigkeiten über Stereotypen möglich
  - aussagekräftigeres Modell
  - besser geeignet für Forward-Engineering
- In UML 17 Standard Stereotypen für Abhängigkeiten (6 Gruppen)

Abhängigkeit - Erweiterungen (Forts.)

- Erste Gruppe: 7 Stereotypen für Abhängigkeiten zwischen Klassen und Objekten
- Andere Gruppen später in anderen Sichten

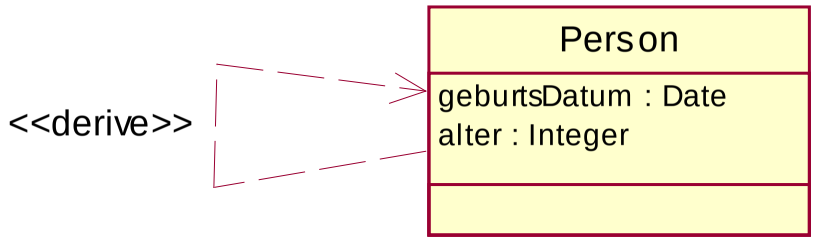
Abhängigkeit - Erweiterungen (Forts.)

- `bind`
  - Abhängigkeit zwischen parametrisierter Klasse und instanziierter Klasse
  - enthält Liste aktueller Parameter, die auf formale Parameter der p. Klasse abgebildet werden



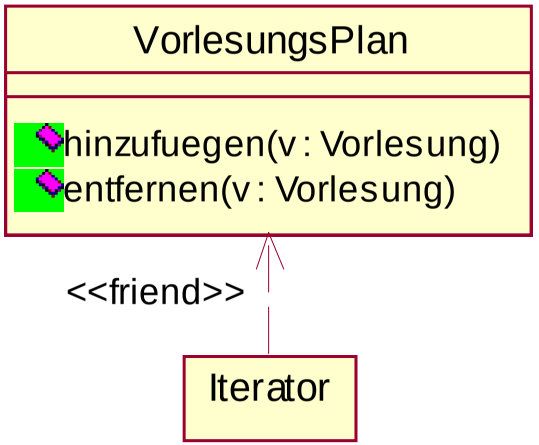
Abhängigkeit - Erweiterungen (Forts.)

- `derive`
  - Abhängigkeit zwischen zwei Attributen (oder Assoziationen), von denen das eine konkret und das anderer herleitbar ist



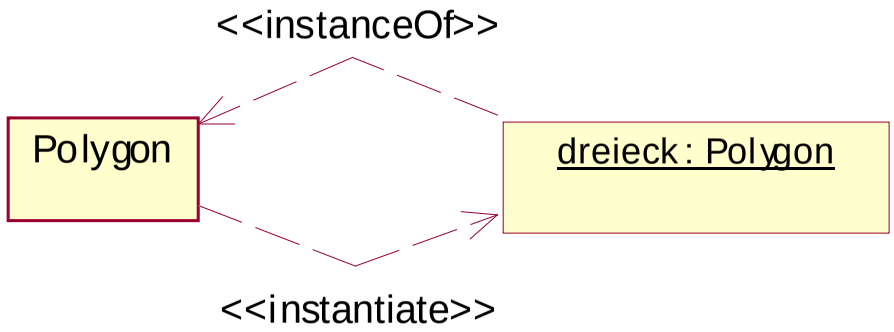
Abhängigkeit - Erweiterungen (Forts.)

- `friend`
  - Quellelement wird eine spezielle Sichtbarkeit auf das Zielelement eingeräumt
  - Modellierung des `friend` Mechanismus in C++



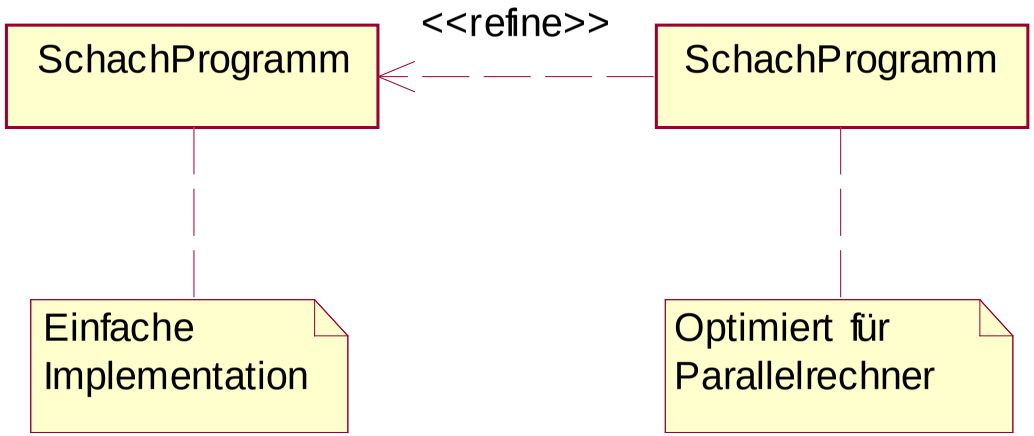
Abhängigkeit - Erweiterungen (Forts.)

- `instanceOf`
  - Quellelement ist eine Instanz des Zielelements
- `instantiate`
  - Quellelement kreiert Instanzen des Zielelements



Abhängigkeit - Erweiterungen (Forts.)

- `refine`
  - gibt an, dass das Quellelement genauer ausgearbeitet ist als das Zielelement



Abhängigkeit - Erweiterungen (Forts.)

- use
  - explizites Modellieren der „benutzt“-Beziehung
  - Quellelement ist abhängig vom sichtbaren Teil des Zielelements

Generalisierung (Generalization)

- *Generalisierung* (Generalization) = Beziehung zwischen einem allgemeinen Element (hier *Oberklasse* (Superclass)) und einem spezielleren Element (hier *Unterklasse* (Subclass))
- Durch Generalisierung *erbt* (inherits) die Unterklasse die Struktur und das Verhalten der Oberklasse

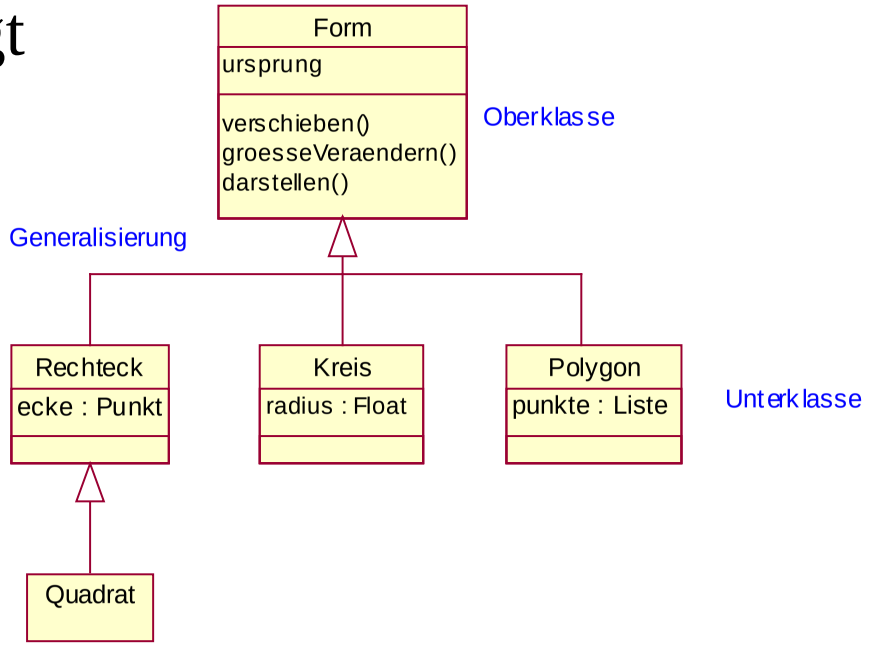
Generalisierung (Forts.)

- Unterklasse erweitert oder modifiziert i.a. Funktionalität der Oberklasse
- Vorzüge der Generalisierung:
  - Reduzierung des Implementationsaufwandes
  - Oberklasse ist durch Unterklasse ersetzbar
  - Nutzung von Polymorphie

Generalisierung (Forts.)

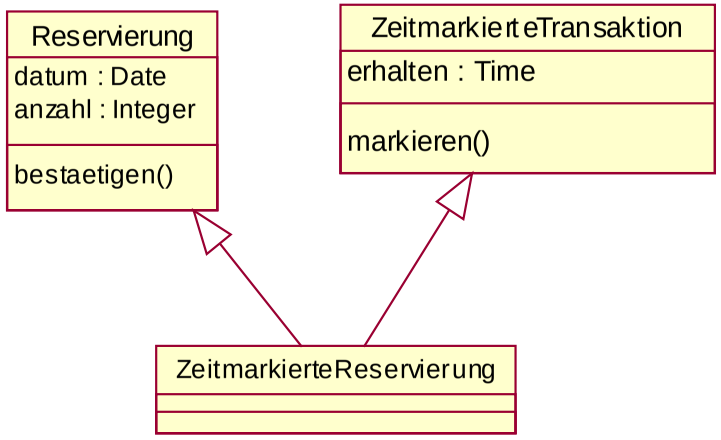
Graphische Darstellung:

– Linie mit Dreieck an dem Ende, welches auf die Oberklasse zeigt



Generalisierung (Forts.)

- *Einfache Vererbung* (Single Inheritance)
  - Klasse hat genau eine Oberklasse
- *Mehrfache Vererbung* (Multiple Inheritance)
  - Klasse hat mehr als eine Oberklasse



Generalisierung - Erweiterungen

- Stereotyp `implementation`
  - Unterklasse erbt Implementation der Oberklasse, aber stellt diese nicht zur Verfügung
    $\Rightarrow$ Oberklasse nicht ersetzbar durch Unterklasse
  - Modellierung der `private` Ableitung in C++

Generalisierung - Erweiterungen (Forts.)

- 4 Standard Einschränkungen
  - `complete`

    alle möglichen Unterklassen sind im Modell spezifiziert worden; zusätzliche Unterklassen sind nicht erlaubt
  - `incomplete`

    es können weitere Unterklassen gebildet werden

Generalisierung - Erweiterungen (Forts.)

- `disjoint`

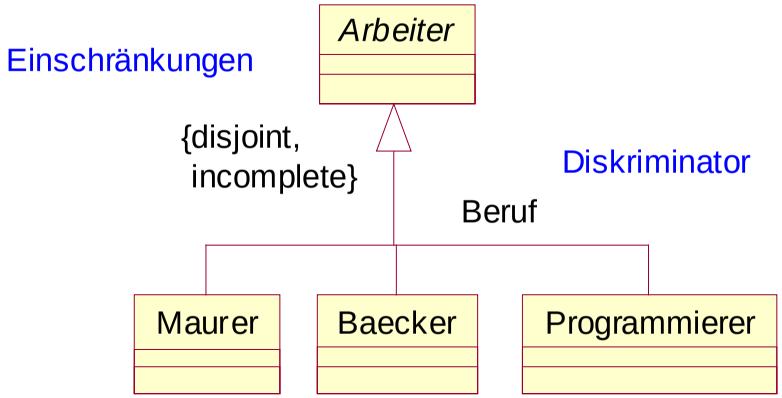
  spezifiziert, dass ein Objekt der Oberklasse nicht mehr als eine Unterklasse als Typ haben kann

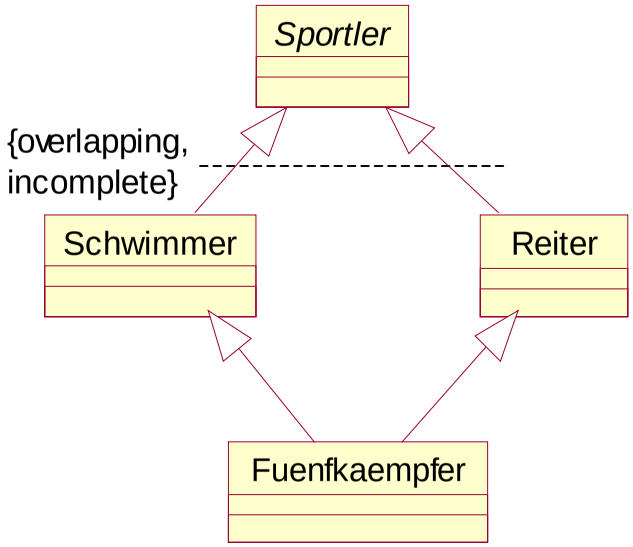
- `overlapping`

  gibt an, dass ein Objekt der Oberklasse mehr als eine Unterklasse als Typ haben kann

Generalisierung - Erweiterungen (Forts.)

- **Beispiel**:





Assoziation (Association)

- *Assoziation* (Association) = strukturelle Beziehung zwischen Elementen
- Mittels Assoziation modelliert man die direkte Zugriffsmöglichkeit der beteiligten Elemente aufeinander
- i.a. hat man binäre Assoziationen (*n*-äre sind jedoch möglich)

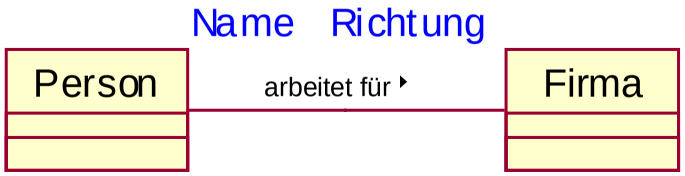
Assoziation (Forts.)

Graphische Darstellung:

– Linie zwischen beteiligten Elementen

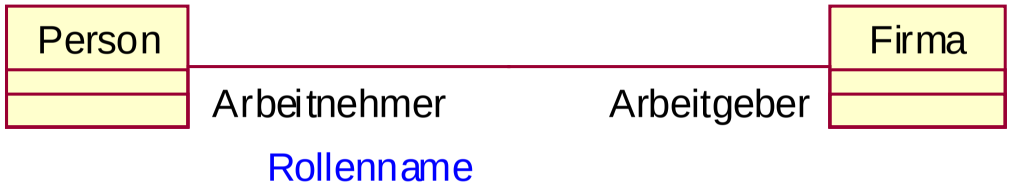
Assoziation - Namen

- Assoziation kann mit einem *Namen* versehen werden, um die Beziehung zu verdeutlichen
- Optional: Angabe der Richtung, in welcher der Name gelesen wird



Assoziation - Rolle

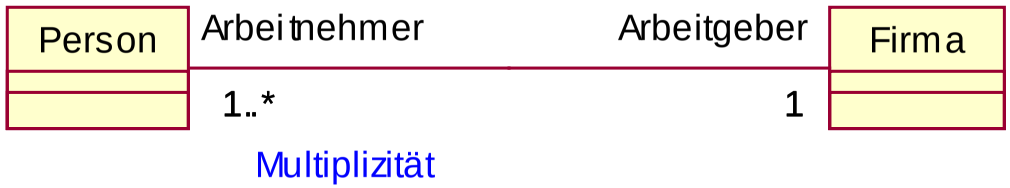
- Klassen, die an einer Assoziation beteiligt sind, spielen in diesem Kontext eine gewisse *Rolle*



- Dieselbe Klasse kann verschiedene Rollen in anderen Assoziationen spielen

Assoziation - Multiplizität

- *Multiplizität* einer Rolle = Anzahl der Verbindungen, die es für eine Rolle geben kann



Assoziation - Navigation

- Klasse *A* und Klasse *B* durch Assoziation verbunden $\Rightarrow$ von Objekt der Klasse *A* kann man zu Objekt der Klasse *B* *navigieren* und umgekehrt
- Einschränkung der Navigierbarkeit durch explizite Spezifikation der *Richtung*

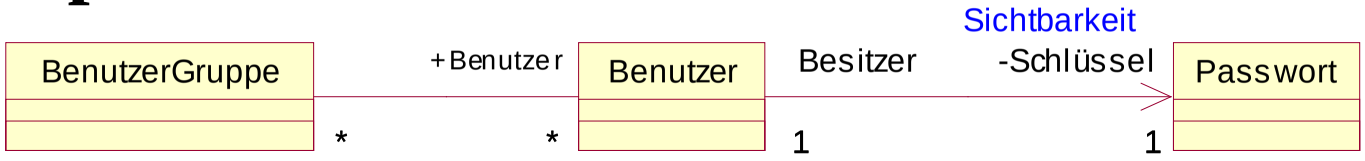
Assoziation - Navigation (Forts.)

- **Beispiel**:

- Einschränkung der Navigierbarkeit bedeutet *nicht*, dass überhaupt nicht auf das andere Objekt zugegriffen werden kann
- Im Vordergrund steht direkter, einfacher Zugriff (i.a. durch Speichern einer Referenz)

Assoziation - Sichtbarkeit

- Für Assoziation lässt sich - wie bei Attributen und Operationen - *Sichtbarkeit* festlegen (Kennzeichnung am Rollennamen)
- **Beispiel**:



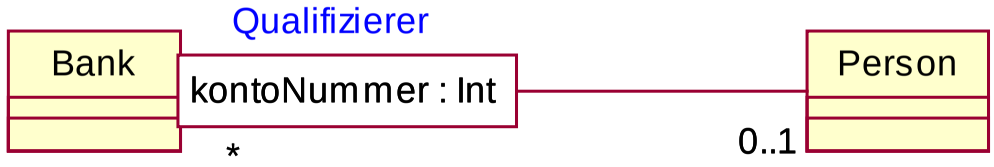
  - Nur von Objekt der Klasse `Benutzer` selbst darf auf das `Passwort` Objekt zugegriffen werden

Assoziation - Qualifizierung

- Bei Assoziation zweier Elemente ergibt sich oft das Problem, dass nur ein bestimmtes Element (oder Teilmenge) von Interesse ist
- Assoziation lässt sich mit einem Attribut, einem sog. *Qualifizierer* (Qualifier), versehen, welches das gewünschte Element genauer spezifiziert

Assoziation - Qualifizierung (Forts.)

- **Beispiel**:

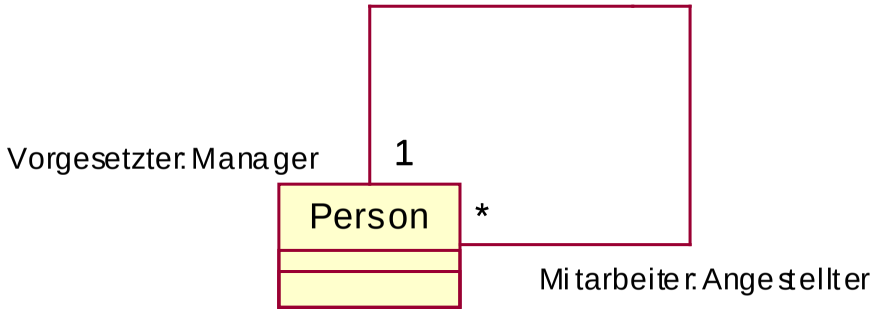


Assoziation - Schnittstellen-Spezifikator

- Klasse kann beliebig viele Schnittstellen realisieren
- Im Kontext einer Assoziation zu einer anderen Klasse ist oft nur eine Schnittstelle von Relevanz
- Kennzeichnung durch *Schnittstellen-Spezifikator* (Interface Specifier) der Form
  `Rolle:Schnittstelle`
  als Rollenname

Assoziation - Schnittstellen-Spezifikator (Forts.)

- **Beispiel**: Personen in der Arbeitswelt
  - versch. Schnittstellen `Manager`, `Leitender Angestellter`, `Angestellter` für Person



Assoziation - Aggregation

- *Aggregation* =
  Modellierung einer „Ganzes-Teile“-Beziehung
- **Graphische Darstellung**:
  - Raute am Ende des „Ganze-Elements“ der Assoziation

- Aggregation ist rein konzeptuell und ändert nichts an der Eigenschaft einer Assoziation

Assoziation - Komposition

- *Komposition* = Spezialform der Aggregation, bei der das Ganze für Konstruktion und Destruktion seiner Teile verantwortlich ist
- Eigenschaften:
  - Weitere Teile können dem Ganzen hinzugefügt werden, nachdem es kreiert wurde. Ganzes zerstört jedoch alle seine Teile bei dessen Destruktion
  - Teil einer Komposition darf nur zu einem einzigen Ganzen gehören (im Gegensatz zur Aggregation)

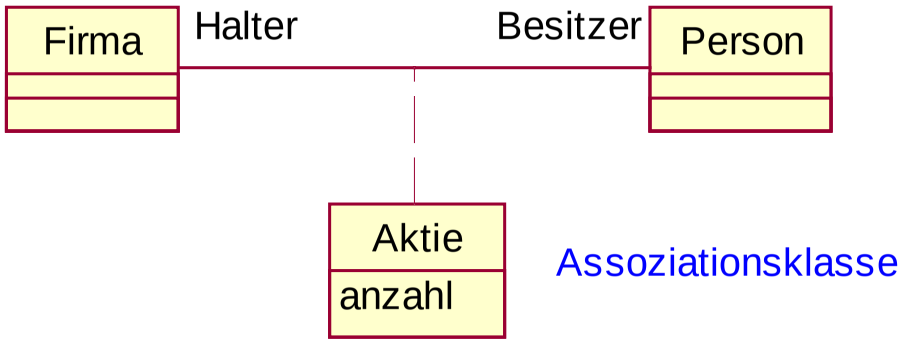
Assoziation - Komposition (Forts.)

Graphische Darstellung:

– wie bei Aggregation, jedoch ist die Raute gefüllt

Assoziation - Assoziationsklasse

- *Assoziationsklasse* (Association class) = Assoziation mit Eigenschaften einer Klasse
- **Graphische Darstellung**:
  - Assoziationsklasse und Assoziation durch gestrichelte Linie verbunden



Assoziation - Assoziationsklasse (Forts.)

- Implizite Einschränkung:
  - es darf nur *eine* instanziierte Assoziationsklasse zwischen zwei Objekten der beteiligten Klassen geben
  - kann es mehrere Assoziationen zwischen zwei Objekten geben, muss dies über eine herkömmliche Klasse modelliert werden

Assoziation - Einschränkungen

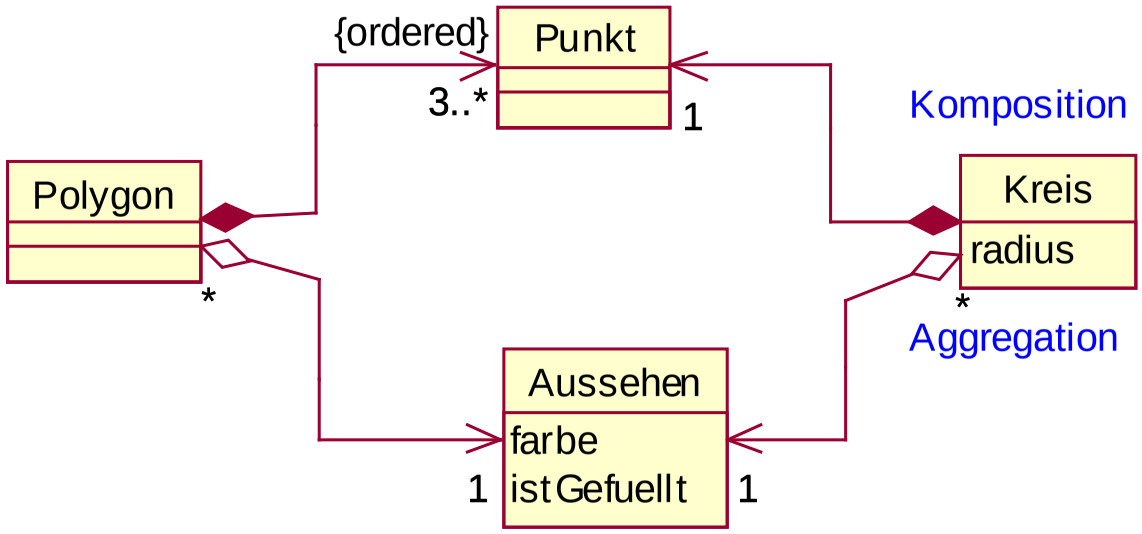
- Einschränkungen für Assoziationen:
  - `implicit`

    Beziehung ist nicht offensichtlich, sondern konzeptuell, z.B. zwei Basisklassen sind assoziiert
    $\Rightarrow$ Unterklassen haben implizite Assoziation
  - `ordered`

    Objekte an einem Ende der Assoziation sind geordnet
    (Voraussetzung: Multiplizität > 1)

Assoziation - Einschränkungen (Forts.)

- **Beispiel**: `ordered`, Komposition, Aggregation



Assoziation - Einschränkungen (Forts.)

- `changeable`

  Verbindungen zwischen Objekten können beliebig modifiziert werden (Standard)
- `addOnly`

  Nur Hinzufügen von Verbindungen erlaubt
- `frozen`

  eine etablierte Verbindung darf nicht mehr entfernt werden

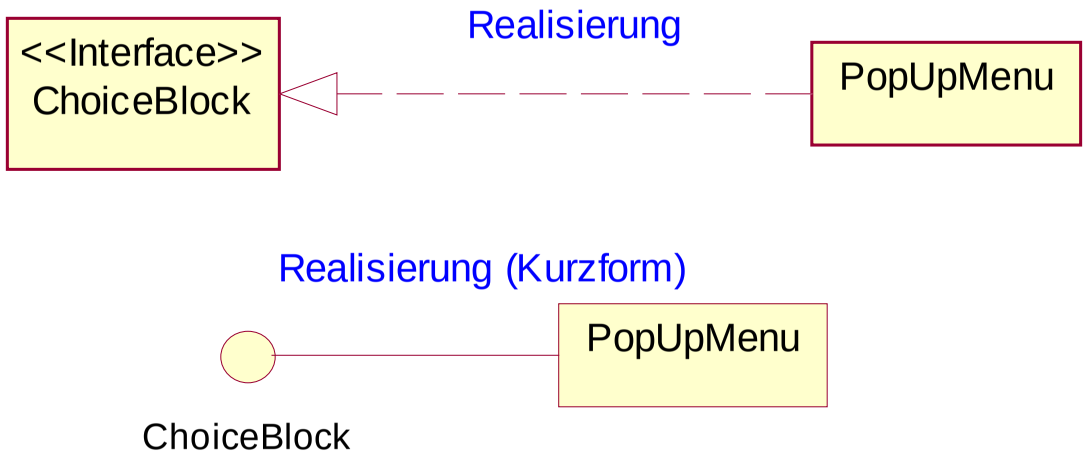
Realisierung (Realization)

- *Realisierung* (Realization) = Beziehung, bei der die eine Seite eine spezielle Anforderung stellt, welche die andere Seite bewerkstelligt
- Verwendung im Zusammenhang mit Schnittstellen und Kollaborationen
- Schnittstelle wird durch Klasse oder Komponente realisiert

Realisierung (Forts.)

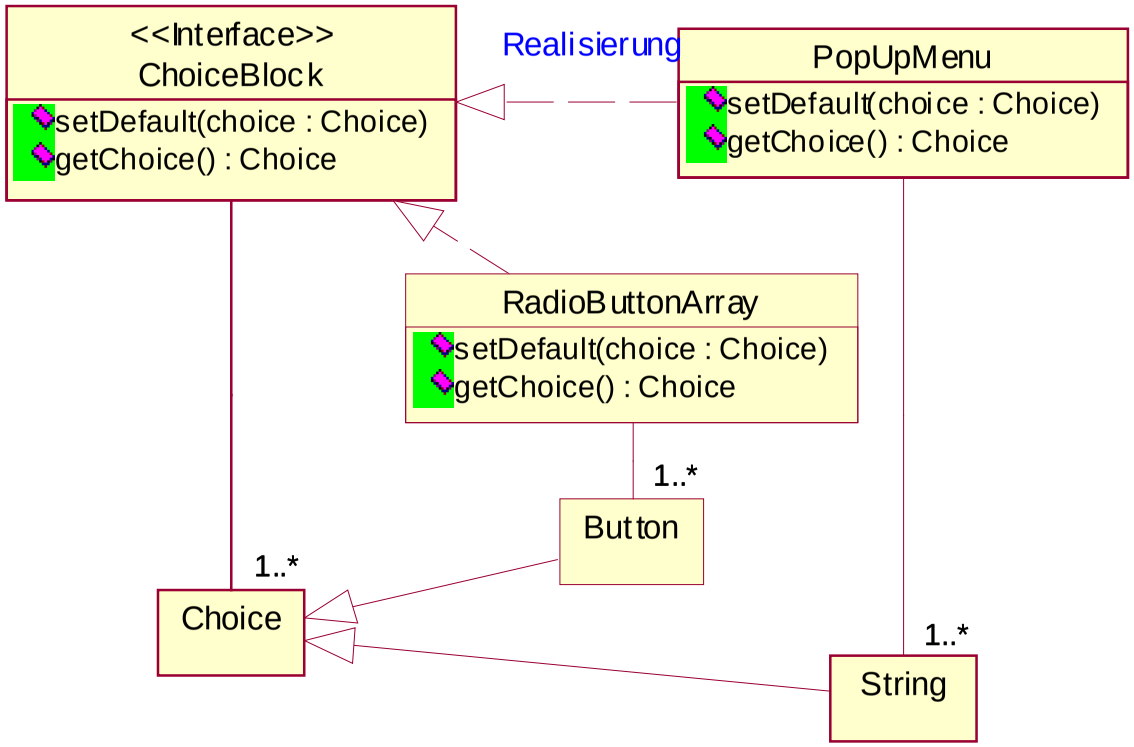
Graphische Darstellung:

– Mischform aus Abhängigkeits- und Generalisierungsbeziehung



Realisierung (Forts.)

- **Beispiel**:

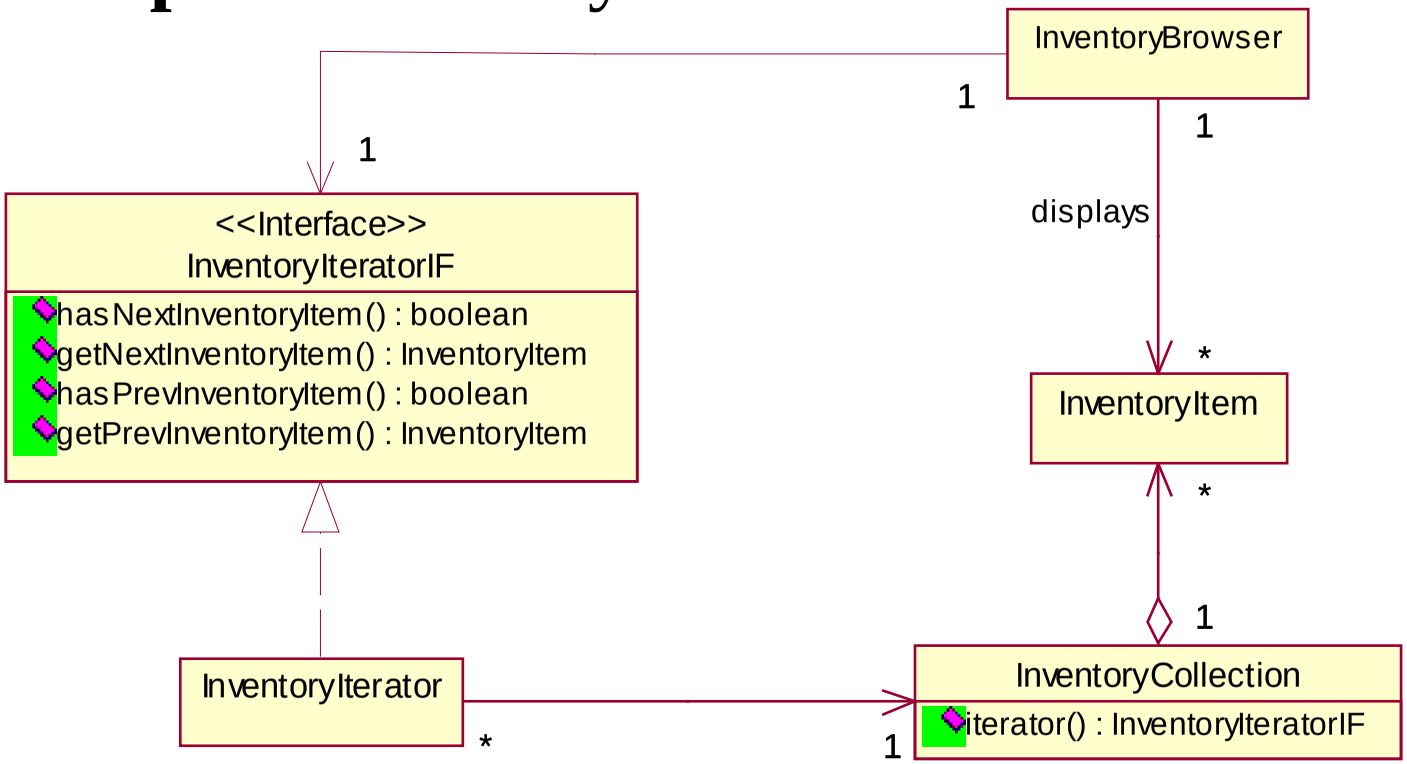


Entwurfsmuster - Iterator

- *Iterator* Entwurfsmuster definiert Zugriffs-möglichkeit auf eine Sammlung von Daten, ohne dass die darunterliegende Datenstruktur offengelegt wird (*Kapselung*)
- Anwendung ermöglicht:
  - einheitlichen Zugriff auf verschiedene Datenstrkt.
  - verschiedene Zugriffsreihenfolgen
  - mehrere Zugriffe zur gleichen Zeit

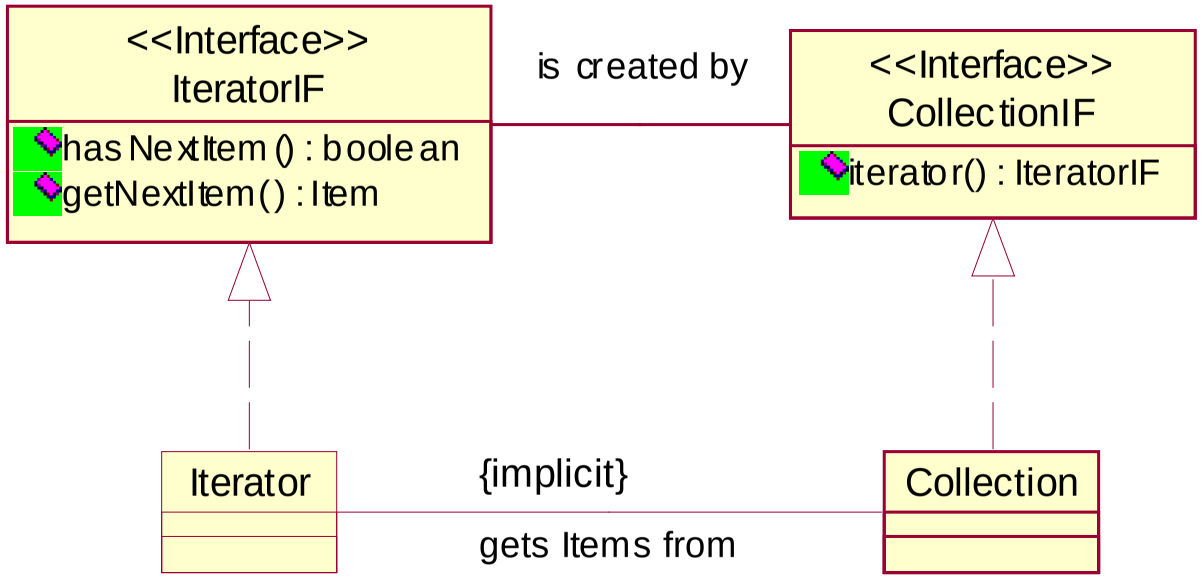
Entwurfsmuster - Iterator (Forts.)

- **Beispiel**: InventoryBrowser



Entwurfsmuster - Iterator (Forts.)

- Abstrakte Modellierung



Entwurfsmuster - Iterator (Forts.)

- Implementationsskizze in Java:

```
public interface InventoryIteratorIF{
   public boolean hasNextInventoryItem();
   public InventoryItem getNextInventoryItem();
   public boolean hasPrevInventoryItem();
   public InventoryItem getPrevInventoryItem();
} // interface InventoryIteratorIF
public class InventoryCollection{
   ...
   public InventoryIteratorIF iterator(){
       return new InventoryIterator();
   }
```

Entwurfsmuster - Iterator (Forts.)

```
    private class InventoryIterator implements
                InventoryIteratorIF{
        public boolean hasNextInventoryItem(){
                ...
        }
        public InventoryItem getNextInventoryItem(){
                ...
        }
        ...
    } // class InventoryIterator
} // class InventoryCollection
```

Entwurfsmuster - Iterator (Forts.)

- In Java gibt es vordefinierte Schnittstellen
  - `java.util.Collection`
  - `java.util.Iterator` bzw. `java.util.Enumeration`

Statische Sicht - Zusammenfassung

- Modellierung der Konzepte einer Applikation in Form von
  - Klassen, Schnittstellen und Datentypen
- sowie der Beziehungen untereinander in Form von
  - Abhängigkeit, Generalisierung, Assoziation und Realisierung

⇒ Bausteine, aus denen das System zusammengesetzt wird