

14. Programmieren mit Löchern

Die Basismittel für objektorientierte Frameworks

Prof. Dr. rer. nat. Uwe Aßmann
Lehrstuhl Softwaretechnologie
Fakultät für Informatik
TU Dresden
Version 22-0.1, 30.04.22

- 1) Abstrakte Klassen und Schnittstellen
- 2) Generische Klassen
- 3) Kern- und Mixinklassen

- ▶ ST für Einsteiger Kap. 4
 - Störrle, Kap. 5.2.6, 5.6
 - Zuser Kap 7, Anhang A
- ▶ Java
 - <http://docs.oracle.com/javase/tutorial/java/index.html> is the official Oracle tutorial on Java classes
 - Generics tutorial
<https://docs.oracle.com/javase/tutorial/java/generics/index.html>
 - Balzert LE 9-10
 - Boles Kap. 7, 9, 11, 12

Vorlesungssprechstunde Mo 11:10
<https://bbb.tu-dresden.de/b/uwe-7dz-bps-p4q>

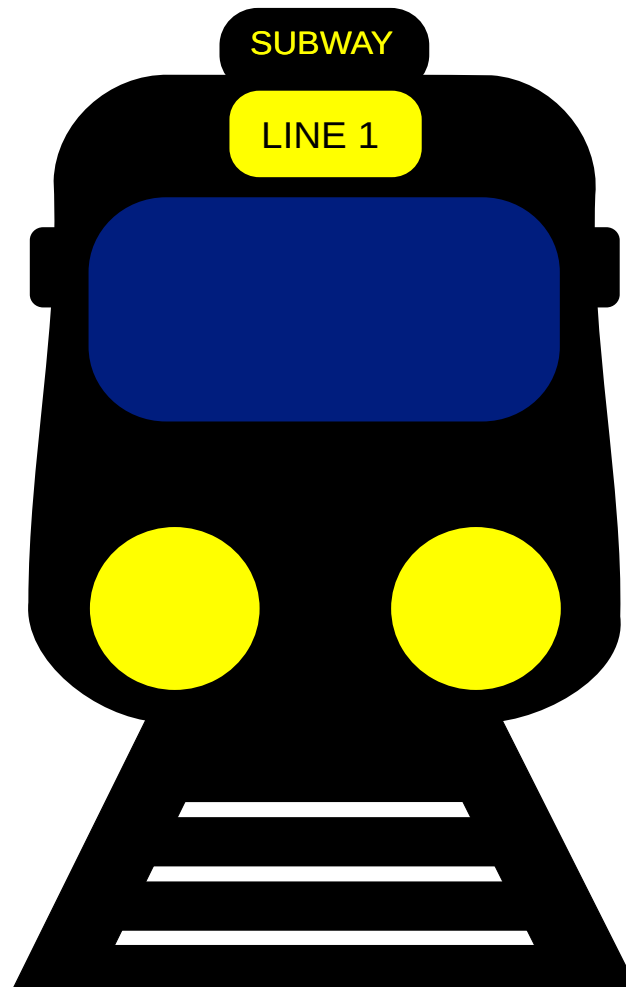
Ziele

- ▶ “Programmieren mit Löchern” nutzt Abstrakte Klassen, Generische Parameter, und Mixins zum Erstellen von Code, der wiederverwendet werden kann
- ▶ Abstrakte Klassen und Schnittstellen verstehen
- ▶ Generische Typen zur Vermeidung von Fehlern (Nachbartypschränken)

Q0.1 Java Herunterladen

4 Softwaretechnologie (ST)

- ▶ Das Java Development Kit (JDK)
- ▶ <http://adaptopenjdk.java.net/>



Def.: Ein Framework bildet eine besondere Art von wiederverwendbarer Komponente, denn es ist mit Löchern programmiert, an denen man es erweitern kann.

- ▶ Ein Framework hat immer einen essentiellen Kern-Ring und einen Erweiterungs-Ring (die Löcher). Es gibt 3 Sorten von Löchern, die wir betrachten werden:
 - fehlende Implementierungen ("Methodenlöcher") ("method hooks")
 - fehlende Typen ("Typlöcher") in generischen Klassen ("type hooks")
 - fehlende oder optionale Mixins ("Mixin-Löcher") in Großobjekten ("mixin hooks")
- ▶ Die Löcher machen Vorgaben für Erweiterungen, z.B. geben sie Typen oder Typschränken für Erweiterungen vor
 - Damit kann der Framework-Konstrukteur dem Anwendungs-Konstrukteur (Plugin-Konstrukteur) Vorgaben machen, um die Funktionsfähigkeit des Frameworks in der Anwendung zu garantieren
- ▶ Komponenten und Frameworks bilden die Basis aller objekt-orientierter Wiederverwendung in Firmen.

14.1 Schnittstellen und Abstrakte Klassen für das Programmieren von Löchern

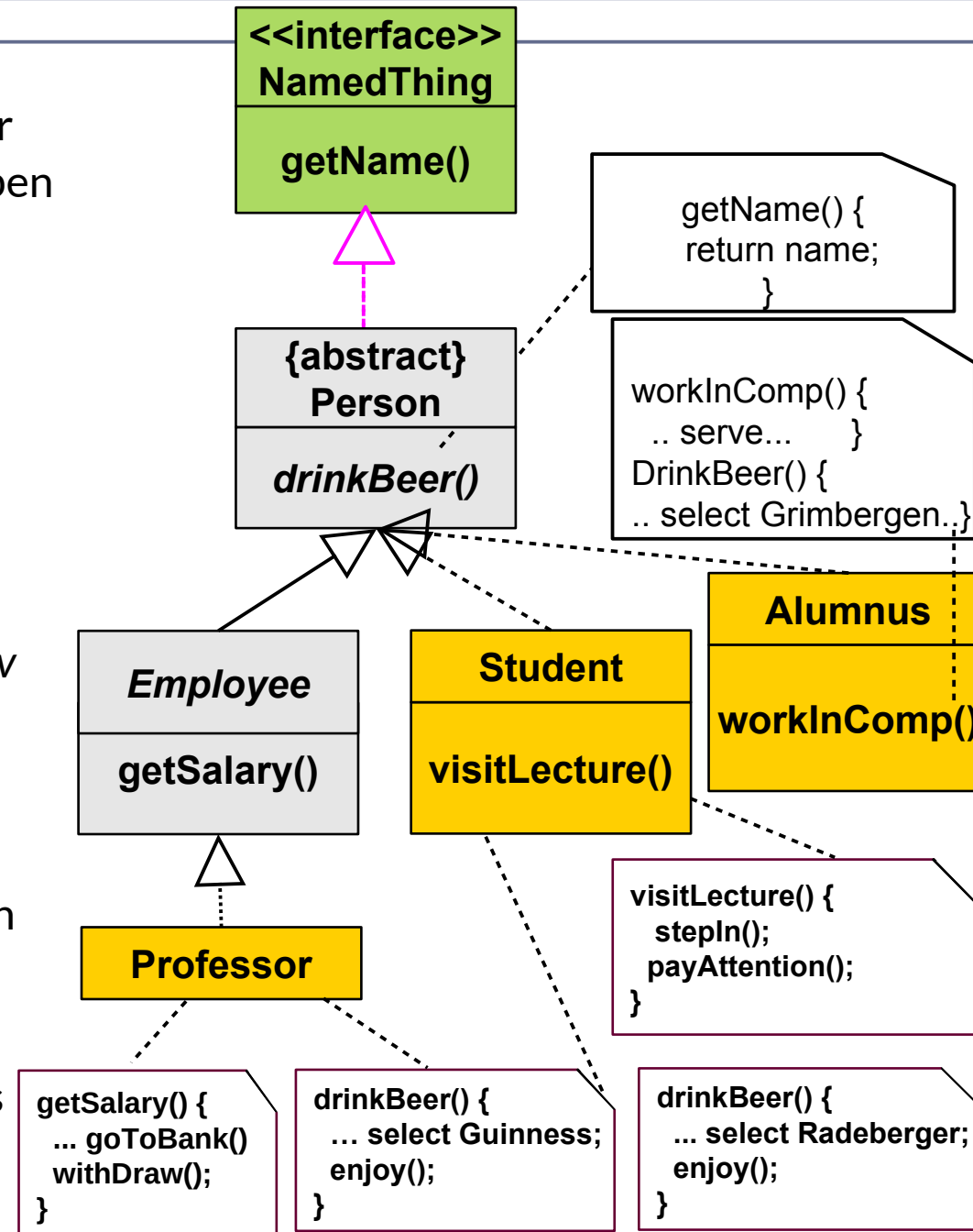
Typen können verschiedene Formen annehmen. Eine partiell spezifizierte Klasse (Schnittstelle, abstrakte Klasse, generische Klasse) macht Vorgaben für Anwendungsentwickler

Schnittstellen und Abstrakte Klassen bilden “Haken”, an die man Klassenimplementierungen anhängen kann

7

Softwaretechnologie (ST)

- ▶ Beim Entwurf von Bibliotheken soll für Anwendungen eine Struktur vorgegeben werden, an die sich alle Anwendungsprogrammierer halten müssen
- ▶ **Vorsehen von “Haken” (hooks)** in der Vererbungshierarchie:
- ▶ **Abstrakte Klassen** werden mit einem speziellen Markierer (tagged value) gekennzeichnet ({abstract}) oder *kursiv* gemalt
- ▶ **Schnittstellen** beschreiben einen Teil der Funktionalität eines Objekts
 - In UML werden sog. Stereotypen vergeben, um Schnittstellen zu kennzeichnen (<<interface>>)
 - Sie dienen dazu, Verhalten eines Objektes in einem bestimmten Kontext festzulegen



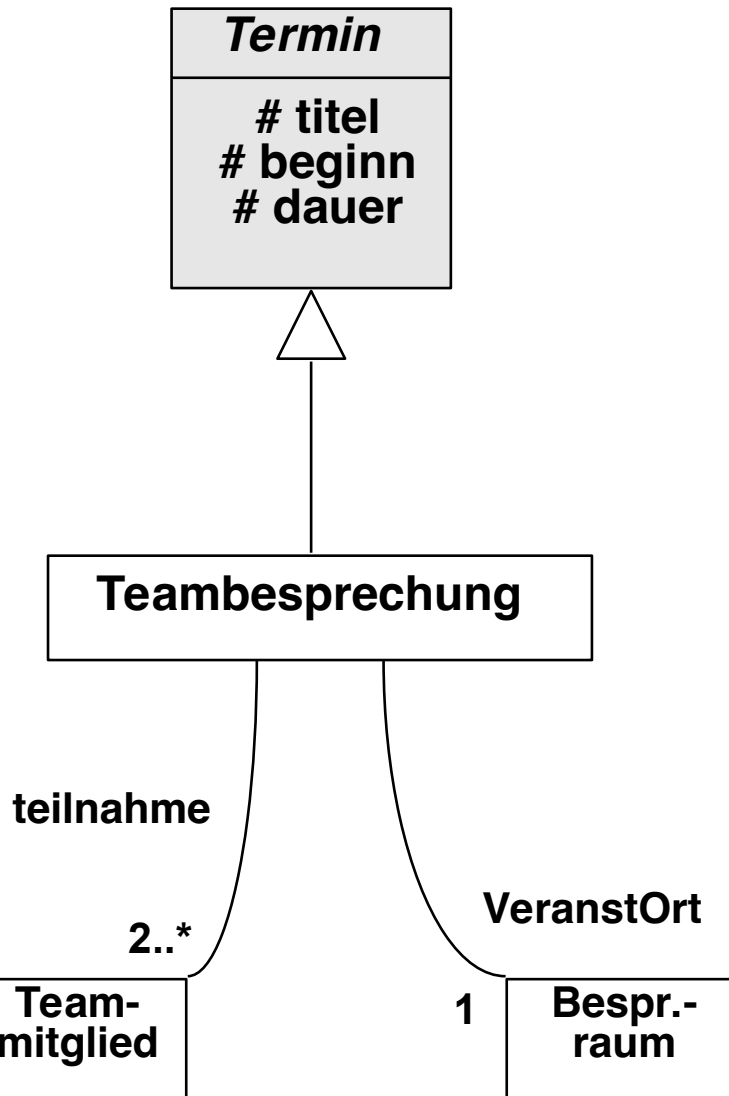
Schnittstellen und Klassen in Java geben “Hooks” vor (“abstract”)

```
interface NamedThing {
    String getName(); // no implementation
}
abstract class Person implements NamedThing {
    String name;
    String getName() { return name; } // implementation exists
    abstract void drinkBeer(); // no implementation
}
abstract class Employee extends Person {
    abstract void getSalary(); // no implementation
}
class Professor extends Employee { // concrete class
    void getSalary() { goToBank(); withdraw(); }
    void drinkBeer() { .. select Guinness(); enjoy(); }
}
class Student extends Person { // concrete class
    void visitLecture() { stepIn(); payAttention(); }
    void drinkBeer() { .. select Radeberger(); enjoy(); }
}
```


Schnittstellen und Klassen in Java geben "Hooks" (Löcher, Haken) vor

```
interface NamedThing {
    String getName(); // no implementation
}
abstract class Person implements NamedThing {
    String name;
    String getName() { return name; } // implementation exists
    abstract void drinkBeer(); // no implementation
}
abstract class Employee extends Person {
    abstract void getSalary(); // no implementation
}
class Professor extends Employee { // concrete class
    void getSalary() { goToBank(); withdraw(); }
    void drinkBeer() { .. select Guinness(); enjoy(); }
}
class Student extends Person { // concrete class
    void visitLecture() { stepIn(); payAttention(); }
    void drinkBeer() { .. select Radeberger(); enjoy(); }
}
class Alumnus extends Person {
    // new concrete class must fit to Person and NamedThing
    void workInComp() { .. serve... }
    void drinkBeer() { ...select Wine... }
}
```

Laufendes Beispiel Terminverwaltung



```
abstract class Termin {
    ...
    protected String titel;
    protected Hour beginn;
    protected int dauer;
    ...
};
```

```
class Teambesprechung
    extends Termin {
    private Teammitglied[] teilnahme;
    private BesprRaum veranstOrt;
    ...
};
```



Beispiel (2): Abstrakte Klassen und Abstrakte Operationen

11

Softwaretechnologie (ST)

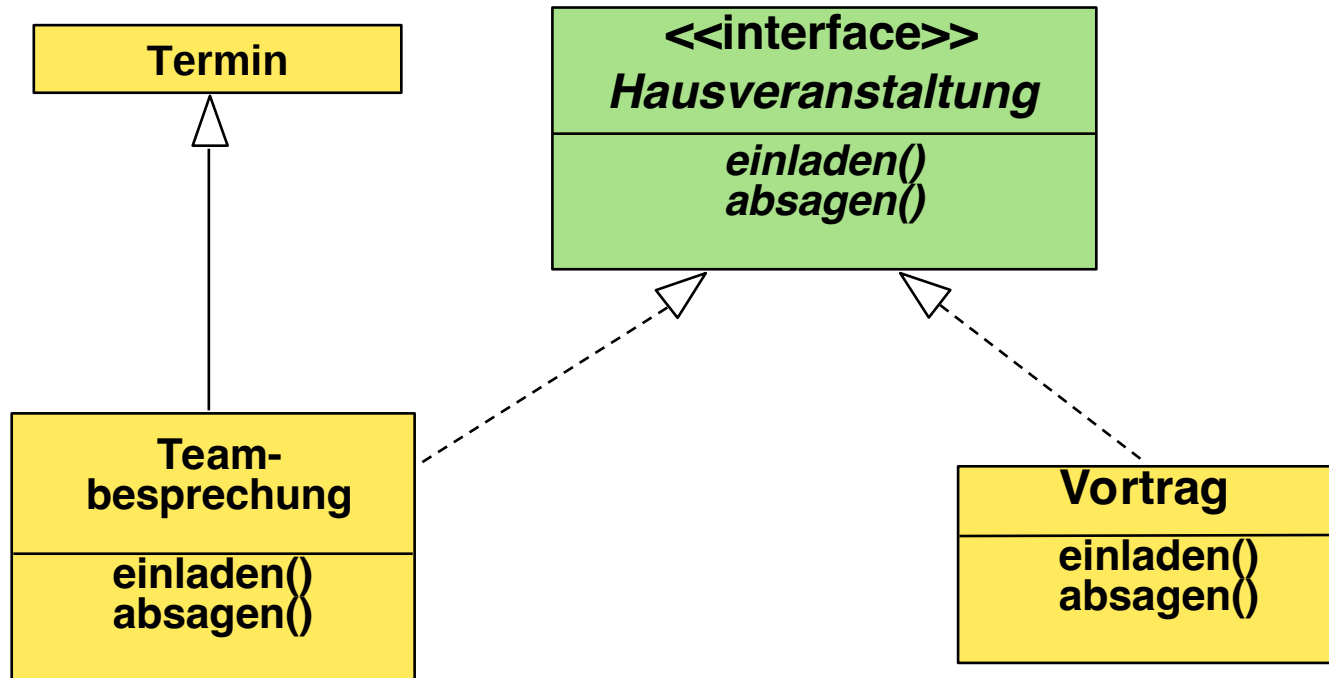
```
abstract class Termin {  
    ...  
    protected String titel;  
    protected Hour beginn;  
    protected int dauer;  
    ...  
    abstract boolean verschieben (Hour neu);  
};
```

Jede abstrakt deklarierte Methode muß in einer Unterklasse realisiert werden - sonst können keine Objekte der Unterklasse erzeugt werden!

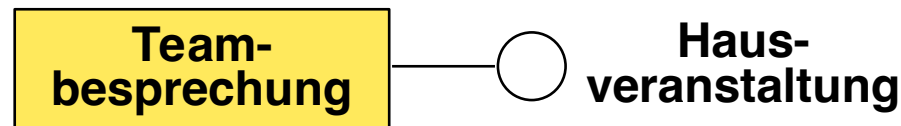
```
class Teambesprechung  
    extends Termin {  
    ...  
    boolean verschieben (Hour neu) {  
        boolean ok =  
            abstimmen(neu, dauer);  
        if (ok) {  
            beginn = neu;  
            raumFestlegen();  
        };  
        return ok;  
    };  
};
```

```
class Betriebsversammlung  
    extends Termin {  
    ...  
    boolean verschieben (Hour neu) {  
        beginn = neu;  
        raumFestlegen();  
        return ok;  
    };  
};
```

Einfache Vererbung von Typen durch Schnittstellen



Hinweis: “Lutscher”-Notation (*lollipop*) für Schnittstellen „Klasse bietet Schnittstelle an“:



Vergleich von Schnittstellen und abstrakte Klassen

Abstrakte Klasse

**Enthält Attribute
und Operationen**

**Kann Default-Verhalten
festlegen**

**Wiederverwendung von
Schnittstellen und Code,
aber keine Instanzbildung**

**Default-Verhalten kann in
Unterklassen überdefiniert
werden**

**Java: Unterklasse kann nur
von einer Klasse erben**

Schnittstelle (voll abstrakt)

**Enthält nur Operationen
(und ggf. Konstante)**

**Kann kein Default-Verhalten
festlegen**

Redefinition unsinnig

**Java und UML: Eine Klasse kann
mehrere Schnittstellen
implementieren**

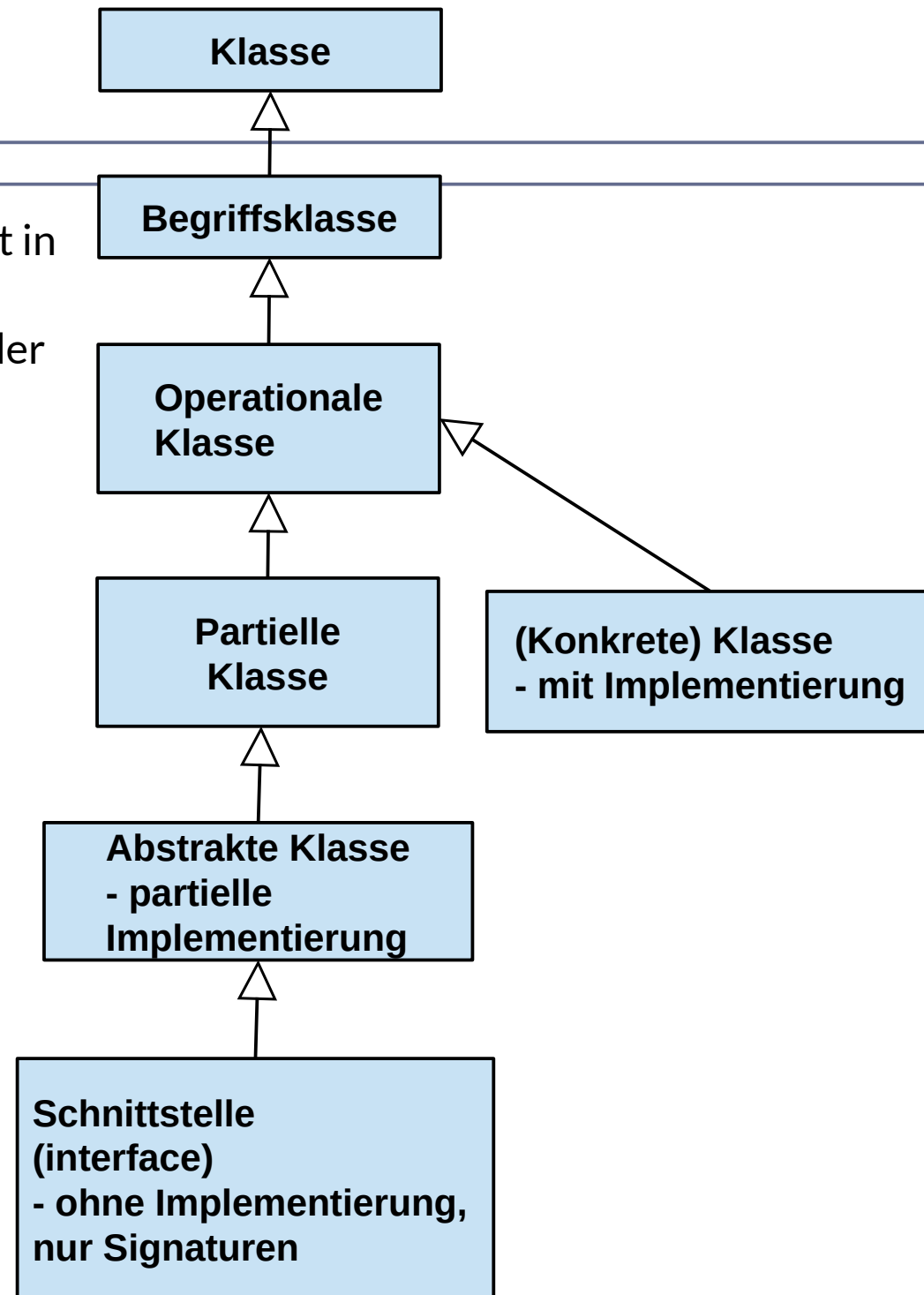
**Schnittstelle ist eine spezielle
Sicht auf eine Klasse**

Q2: Begriffshierarchie von Klassen (Erw.)

14

Softwaretechnologie (ST)

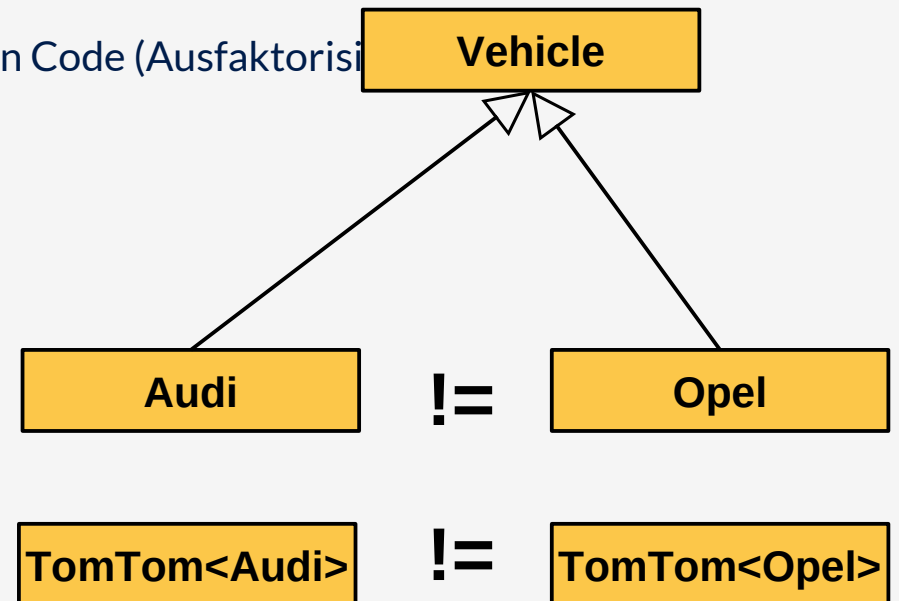
- ▶ *Operationale Klassen* werden unterteilt in Klassen mit, ohne, und mit Implementierung einer Untermenge der Operationen
- ▶ *Schnittstellen* und *Abstrakte Klassen* dienen dem Teilen von Typen und partiellem Klassen-Code



14.2. Generische Klassen (Klassenschablonen, Template-Klassen, Parametrische Klassen)

... bieten eine weitere Art, mit Löchern zu programmieren, um Vorgaben zu machen

- Generische Klassen lassen den Typ von einigen Attributen und Referenzen offen ("generisch")
- Sie ermöglichen typisierte Wiederverwendung von Code (Ausfaktorisierung von Gemeinsamkeiten)
- Sie helfen, Nachbarn zu spezialisieren



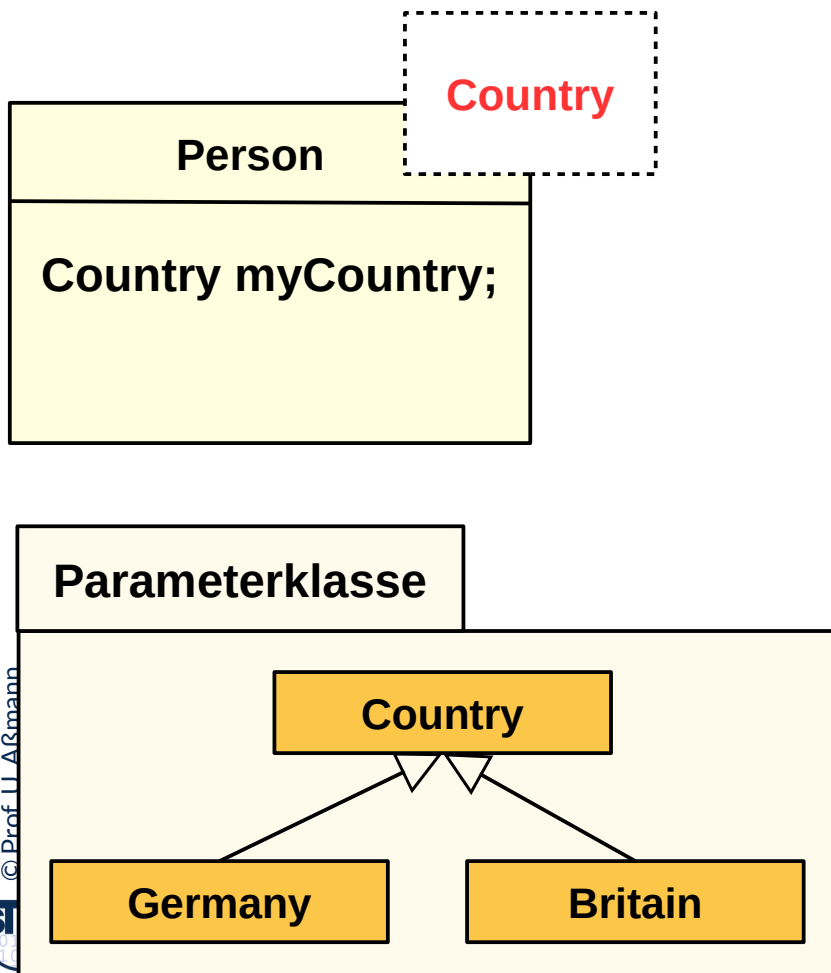
Generische Klassen

16

Softwaretechnologie (ST)

Def.: Eine *generische (parametrische, Template-) Klasse* ist eine Klassenschablone, die mit einem oder mehreren Typparametern (für Attribute, Referenzen, Methodenparameter) versehen ist.

► In UML



► In Java

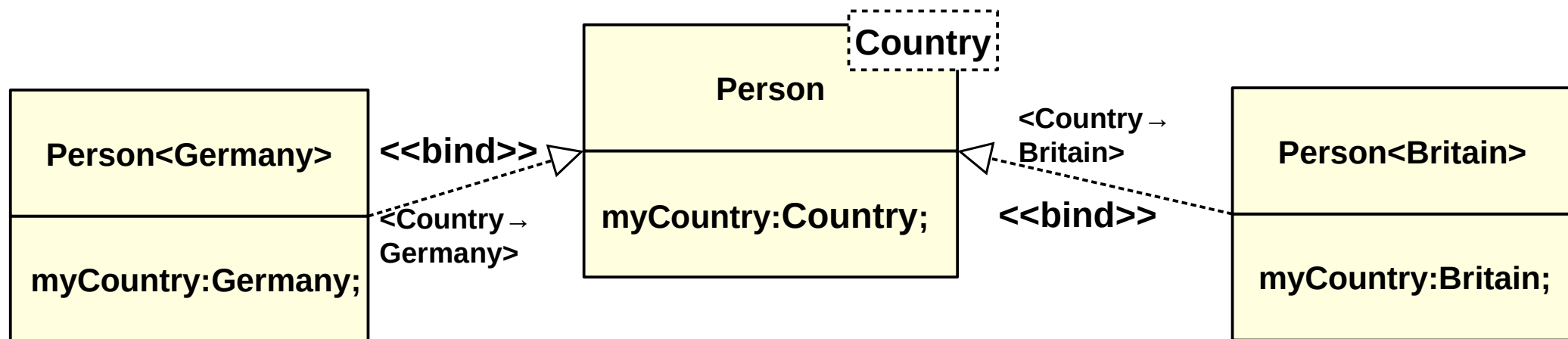
- Sprachregelung: "Person of Country"

```
// Definition of a generic class  
class Person<Country> {  
    Country myCountry;  
}
```

```
/* Type definition using a generic type */  
Person<Germany> egon;  
Person<Britain> john;
```


Feinere statische Typüberprüfung für Attribute

- ▶ Zwei Attributtypen, die durch Parameterisierung aus einer generischen Klassenschablone entstanden sind, sind nicht miteinander kompatibel
- ▶ Der Übersetzer entdeckt den Fehler (**statische Typprüfung**)
- ▶ Die generische Klasse beschreibt das Gemeinsame (Generalisierung); der Parameter die Verschiedenheiten; die ausgeprägte Klasse die Spezialisierung



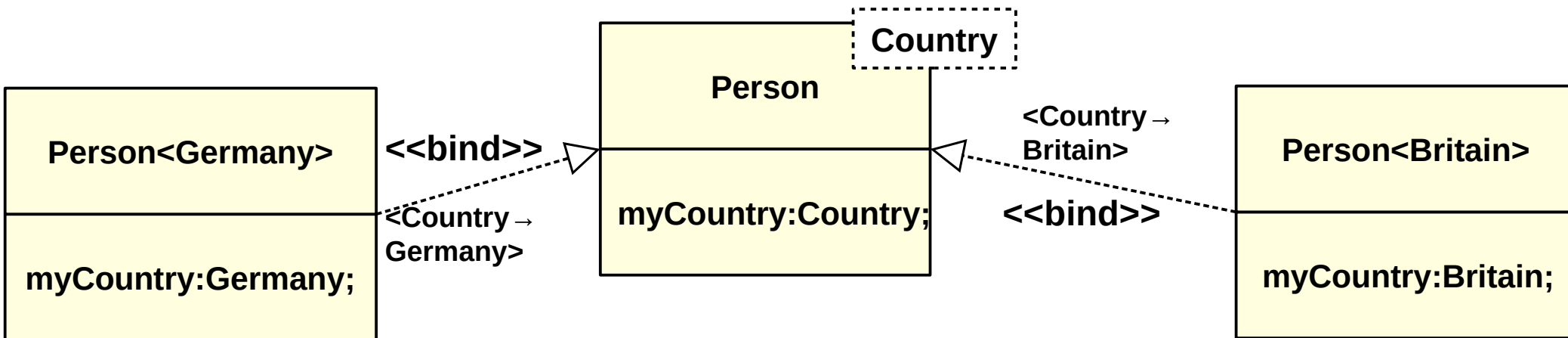
```
/* Type definition and initialization with object */
Person<Germany> egon = new Person<Germany>;
Person<Britain> john = new Person<Britain>;

/* Checks of assignments can use the improved typing */
john = egon;
```



Einsatzzweck: Typen von Nachbarn vorgeben

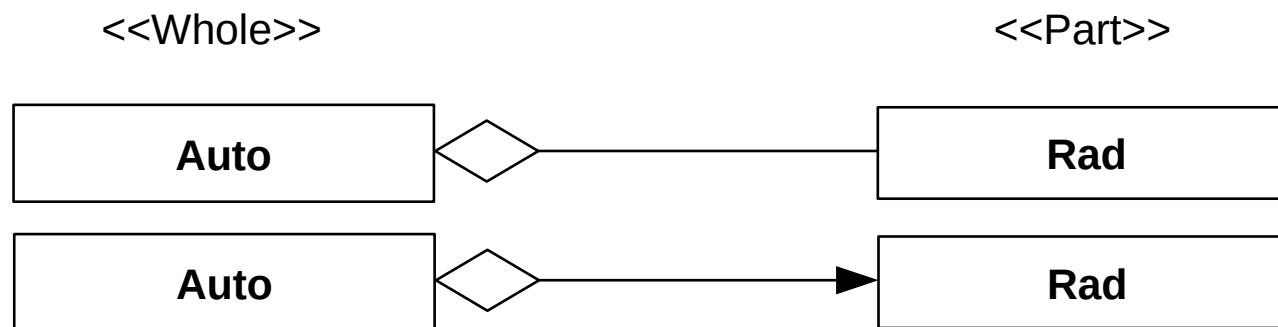
- ▶ **Def.:** Wenn ein formaler Typparameter einer generischen Klasse einen Nachbarn beschreibt, nennen wir ihn **Nachbartypschränke**



- ▶ **Bsp.:**
 - C gibt für die generische Klasse den Typ des Landes Country vor (Nachbar)
 - C ist eine Nachbartypschränke

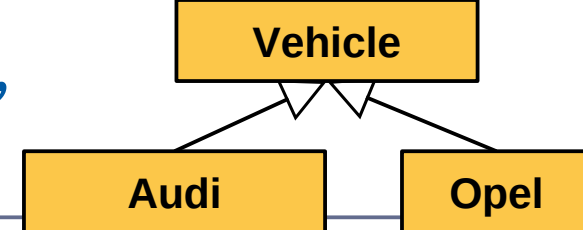
Einsatzzweck: Typen von Nachbarn vorgeben, Typsichere Aggregation (has-a)

- ▶ **Def.:** Wenn eine Assoziation den Namen „hat-ein“ oder „besteht-aus“ tragen könnte, handelt es sich um eine **Aggregation** zwischen einem *Aggregat*, dem *Ganzen*, und seinen *Teilen* (Ganzes/Teile-Relation, whole-part relationship).
 - Die auftretenden Aggregationen bilden auf den Objekten immer eine transitive, antisymmetrische Relation (einen gerichteten zyklensfreien Graphen, *dag*).
 - Ein Teil kann zu mehreren Ganzen gehören (*shared*), zu einem Ganzen (*owns-a*) und exklusiv zu einem Ganzen (*exclusively-owns-a*)
 - Exklusiv zugeeignete Objekte heißen **Unterojekte**

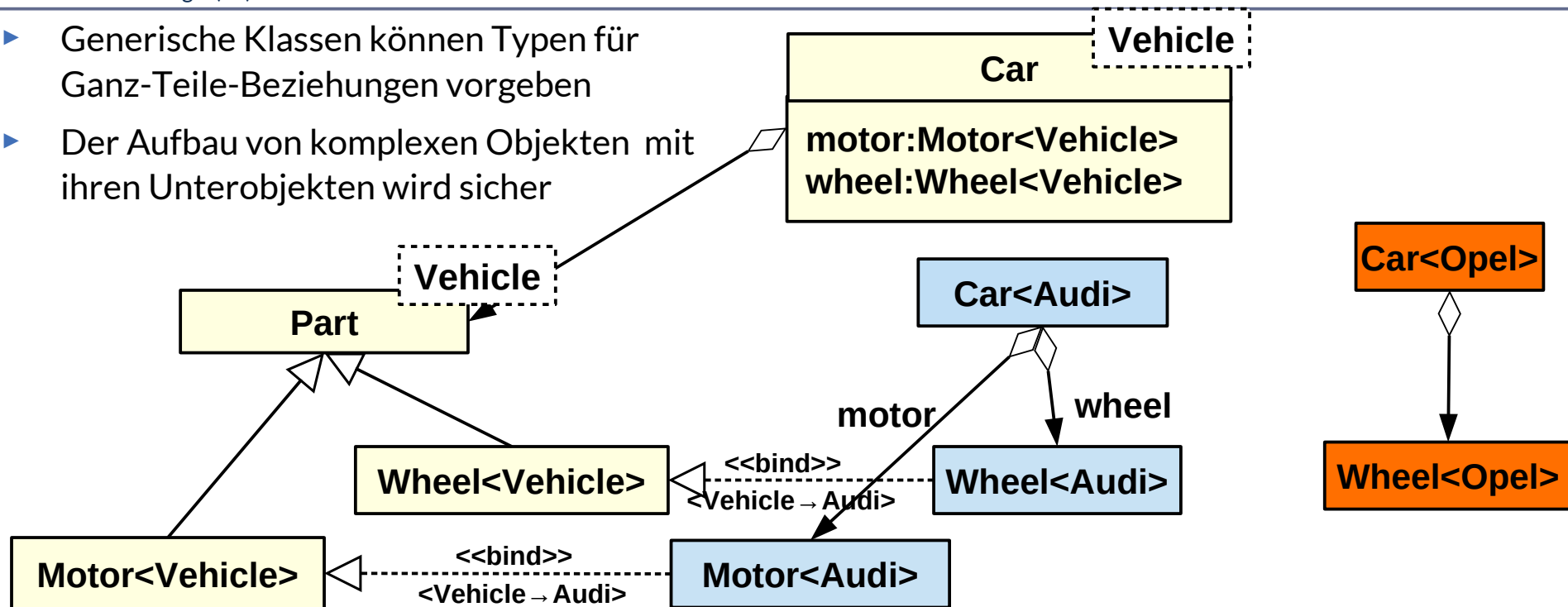


Lies: „Auto *hat ein* Rad“

Einsatzzweck: Typen von Nachbarn vorgeben, Typsichere Aggregation (has-a)



- ▶ Generische Klassen können Typen für Ganz-Teile-Beziehungen vorgeben
- ▶ Der Aufbau von komplexen Objekten mit ihren Unterobjekten wird sicher



```

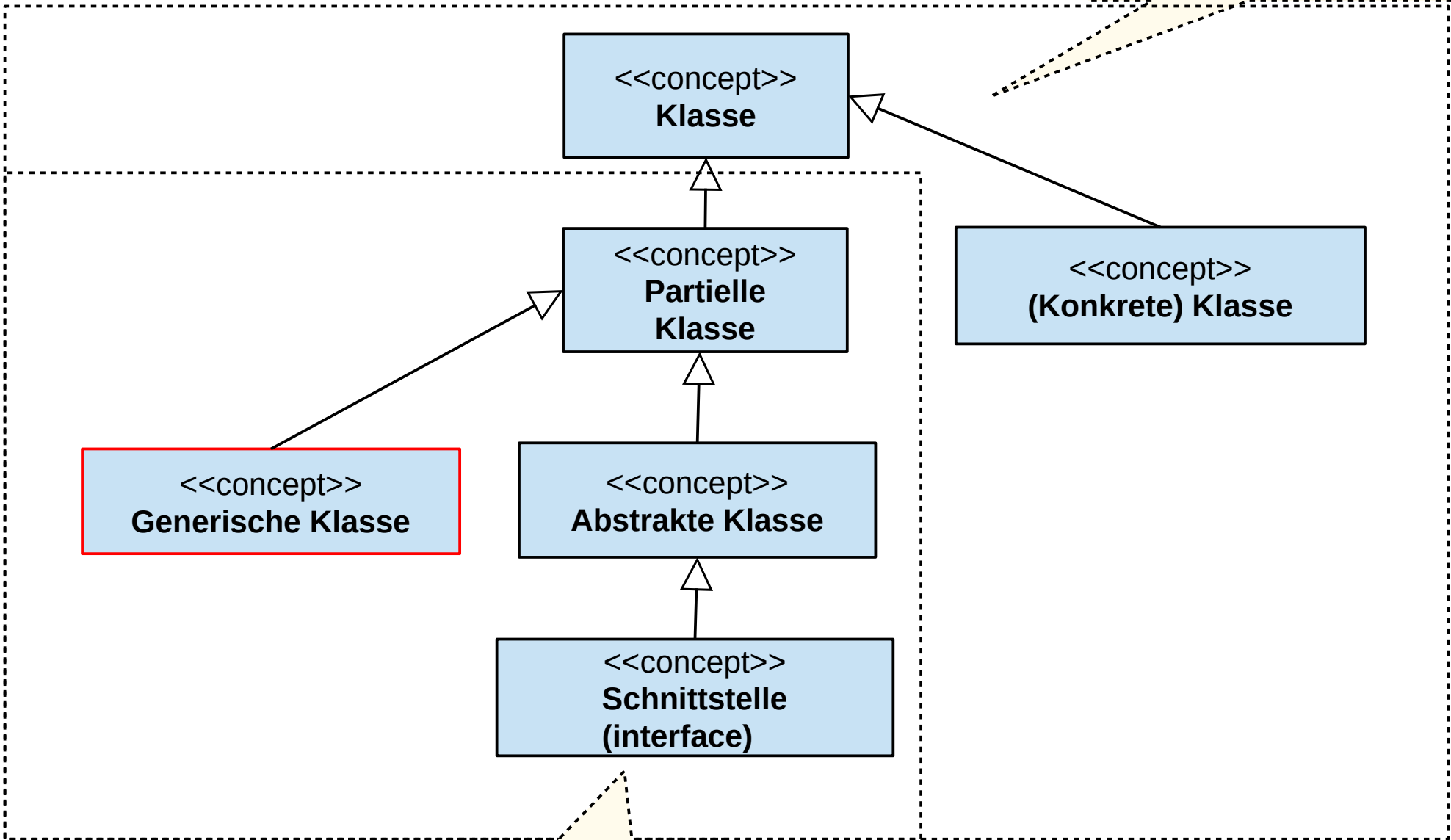
/* Type definition and initialization with object */
Motor<Audi> motorOfAudi = new Motor<Audi>;
Wheel<Audi> wheelOfAudi = new Wheel<Audi>;
Wheel<Opel> wheelOfOpel = new Wheel<Opel>;

/* Checks of assignments can use the improved typing */
Car<Audi> audi = new Car<Audi>();
audi.motor = motorOfAudi;
audi.wheel = wheelOfOpel;
    
```



Q2: Begriffshierarchie von Klassen (Erweiterung)

Fürs Programmieren mit
Gemeinsamkeiten



Fürs Programmieren mit
Löchern

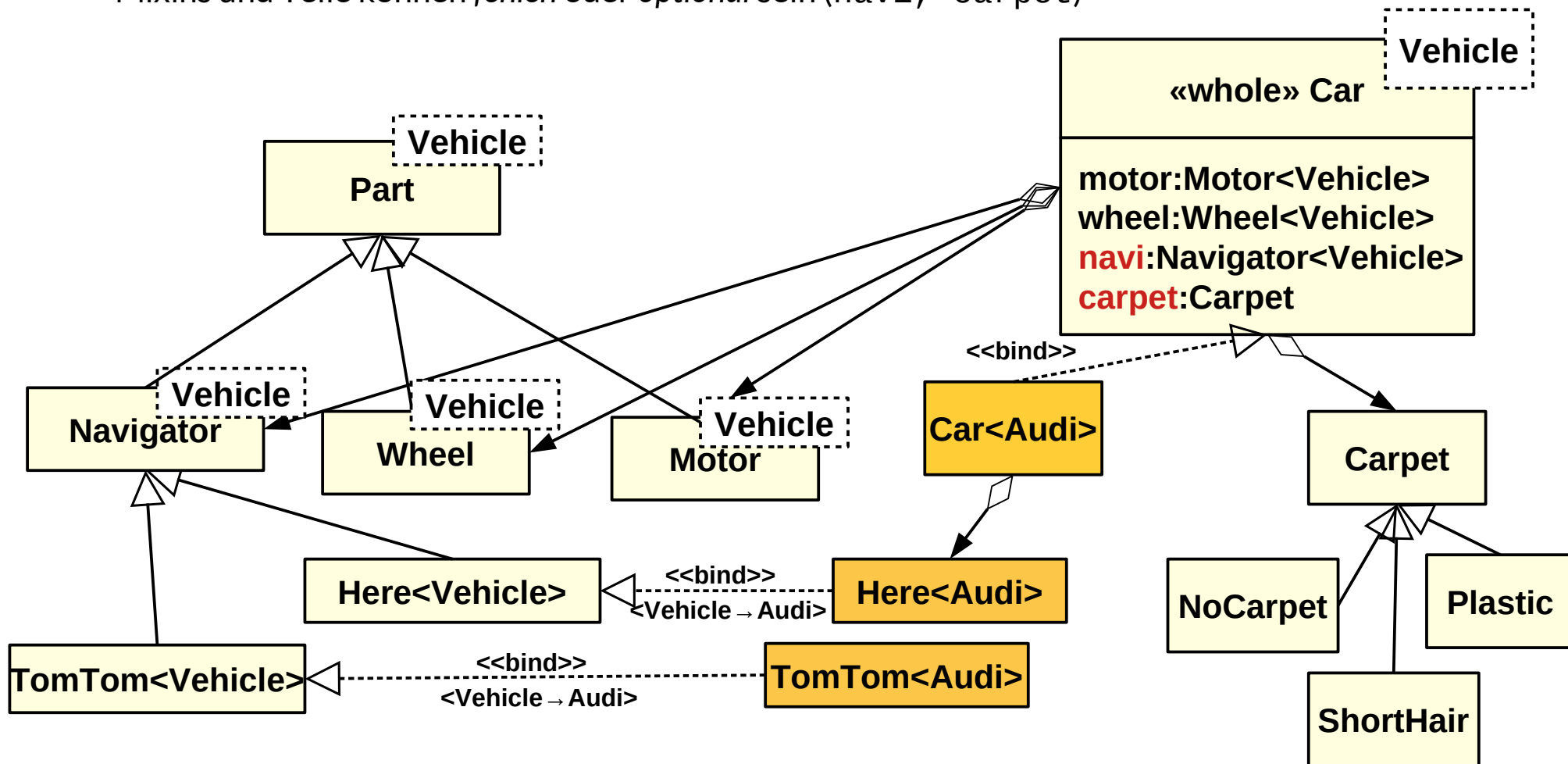
14.3. Kern-Klassen (Mixinhalter-Klassen) und ihre austauschbaren Mixins

... Programmieren mit Löchern für Ganzes und Teile

- Kernobjekte haben Unterobjekte (Mixins), abhängige Unterobjekte
- Mixins können *fehlen*, d.h. optional sein oder nur temporär vorhanden (temporäre “Löcher”)

Fehlende oder optionale Teile werden durch Mixins dargestellt

- ▶ Def.: Ein **Mixin** ist ein Unterobjekt, das semantisch von einem Hauptobjekt (**Kern-Objekt**) abhängig ist. Die Klasse des Kerns heißt **Kernklasse**.
- ▶ Def.: Ein **Teil** ist ein Mixin, das ein Teil eines Ganzen darstellt, das durch der Kern repräsentiert wird.
- ▶ Mixins und Teile können *fehlen* oder *optional* sein (navi, carpet)

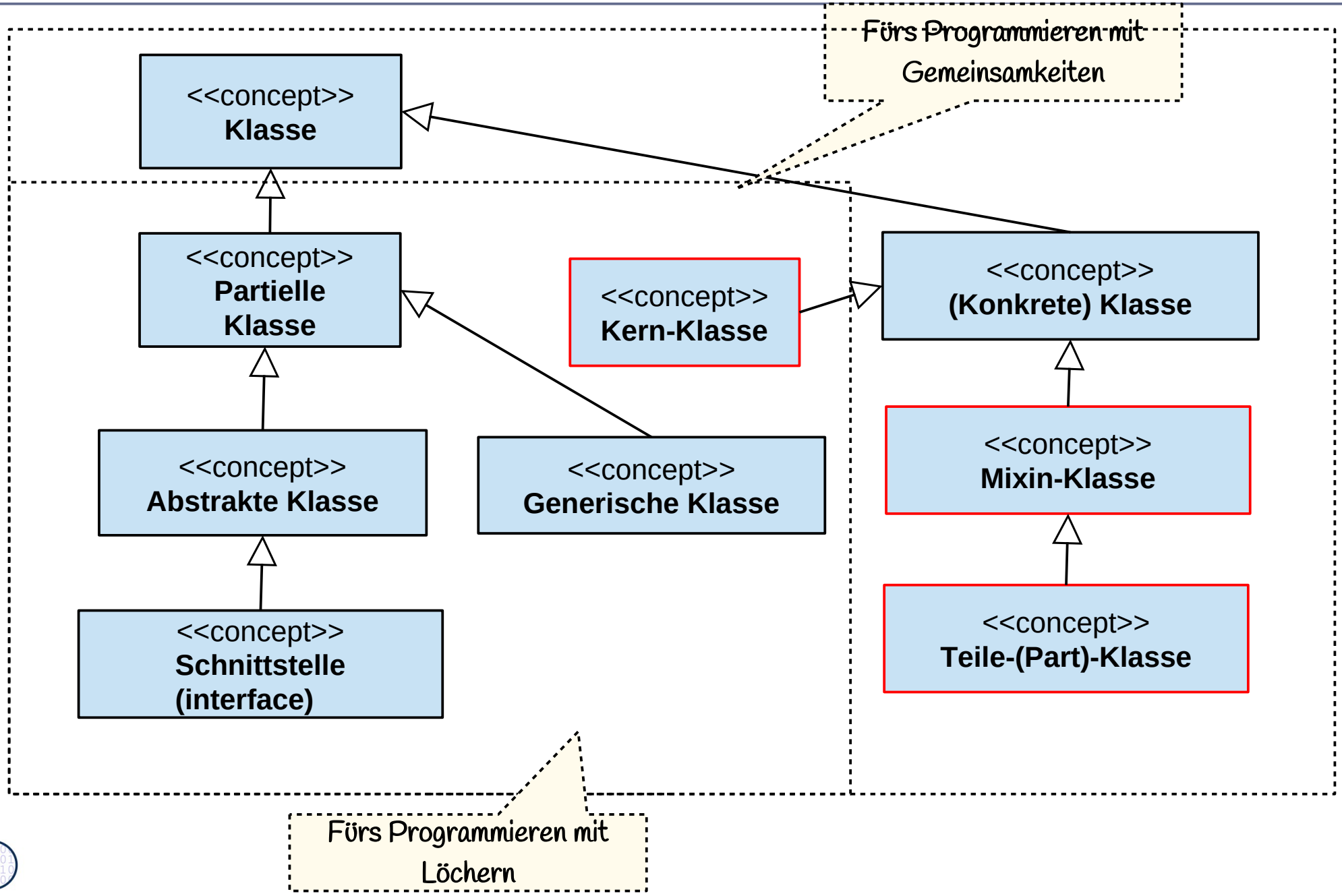


Polymorphie mit Mixins

- ▶ Def.: **Mixin-Polymorphie** entsteht, wenn ein Mixin seine Klasse gegen alternative Klassen austauscht (Wandel der Gestalt des Mixins)
- ▶ Mixins können optional sein, also oft fehlen (temporäre Löcher)

```
/* Type definition and initialization with object */  
.. similar to before..  
Carpet nocarp = new NoCarpet;  
Carpet shairc = new ShortHair();  
Carpet plastc = new Plastic();  
  
Navigator<Audi> navi1 = new Here<Audi>;  
Navigator<Audi> navi2 = new TomTom<Audi>;  
  
/* mixin polymorphy over navi and carpet */  
Car<Audi> audi = new Car<Audi>();  
audi.carpet = new NoCarpet();  
audi.navi = navi1;      // now we enjoy Here  
.... later in the life of the car ...  
audi.carpet = plastc;  // now we have a short hair carpet  
.... later in the life of the car ...  
audi.carpet = shairc;  // now we have a short hair carpet  
audi.navi = navi2;     // now we have a new navi, the TomTom
```


Q2: Begriffshierarchie von Klassen (Erweiterung)



- ▶ Schnittstellen als auch abstrakte Klassen erlauben es, Anwendungsprogrammierern Struktur vorzugeben
 - Sie definieren “Haken”, in die Unterklassen konkrete Implementierungen schieben
 - Schnittstellen sind vollständig abstrakte Klassen
- ▶ Generische Klassen ermöglichen typsichere Wiederverwendung von Code über Typ-Parameter → der Compiler meldet mehr Fehler
- ▶ Kernobjekte haben Mixins für optionale oder fehlende Teile
- ▶ Mixinklassen können durch generische Parameter oder normale Klassen getypt sein

Warum ist das wichtig?

- ▶ **Bau von Frameworks (Rahmenwerken)** ist eines der Hauptprobleme des Software Engineering, weil es Wiederverwendung organisiert
 - Von Projekt zu Projekt
 - Von Produkt zu Produkt (Produktfamilien, Produktlinien)
- ▶ Abstrakte Klassen, Schnittstellen und generische Klassen können Code-Replikate und Code-Explosion weitgehend vermeiden und **gleichzeitig Vorgaben für Erweiterungen machen**
- ▶ Wiederverwendung **mit Frameworks** ist das Hauptmittel der Softwarefirmen, um **profitabel** arbeiten zu können

- ▶ Geben Sie eine Begriffshierarchie des Klassenbegriffs an. Welche Klassenarten kennen Sie? Wie spezialisieren sie sich?
- ▶ Erklären Sie den Unterschied der “Löcher” in abstrakten Klassen und in generischen Klassen.
- ▶ Gibt es einen Unterschied zwischen `{abstract}Employee` und *Employee* ?
- ▶ Warum wird die Wiederverwendung von Software durch Frameworks vereinfacht? Wozu gibt der Framework-Konstrukteur Vorgaben für die Löcher vor?
- ▶ Java hat kein Mixin-Sprachkonstrukt. Wie könnte man dennoch aufzeichnen, dass das Mixin semantisch von seinem Hauptobjekt abhängig ist?
 - Viele moderne Sprachen wie C# und Scala kennen Mixin-Vererbung
- ▶ Sollte ein Mixin privat oder öffentlich sein? Ist es gut, auf Attribute des Mixins direkt zugreifen zu können? Was passiert bei Polymorphie des Mixins?

14. Programmieren mit Löchern

Die Basismittel für objektorientierte Frameworks

Prof. Dr. rer. nat. Uwe Aßmann
Lehrstuhl Softwaretechnologie
Fakultät für Informatik
TU Dresden
Version 22-0.1, 30.04.22

- 1) Abstrakte Klassen und Schnittstellen
- 2) Generische Klassen
- 3) Kern- und Mixinklassen

Keep your code clean, take your stance:
Use single inheritance.
Instead copy, move it up
Superclass will be its hub.
Breathe into your objects life -
Let them polymorphy drive.
Abstract classes, interface
Shape their childrens' pretty face.

- ▶ ST für Einsteiger Kap. 4
 - Störrle, Kap. 5.2.6, 5.6
 - Zuser Kap 7, Anhang A
- ▶ Java
 - <http://docs.oracle.com/javase/tutorial/java/index.html> is the official Oracle tutorial on Java classes
 - Generics tutorial
<https://docs.oracle.com/javase/tutorial/java/generics/index.html>
 - Balzert LE 9-10
 - Boles Kap. 7, 9, 11, 12

Vorlesungssprechstunde Mo 11:10
<https://bbb.tu-dresden.de/b/uwe-7dz-bps-p4q>

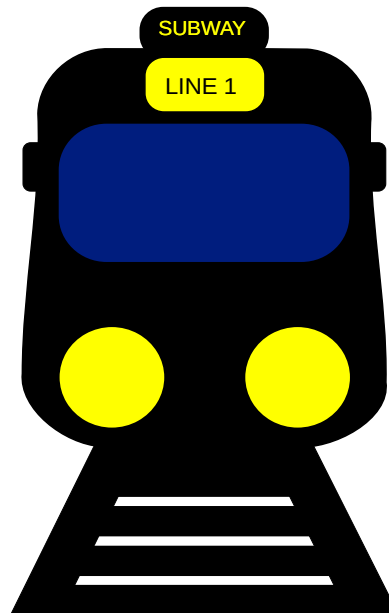
Ziele

- ▶ “Programmieren mit Löchern” nutzt Abstrakte Klassen, Generische Parameter, und Mixins zum Erstellen von Code, der wiederverwendet werden kann
- ▶ Abstrakte Klassen und Schnittstellen verstehen
- ▶ Generische Typen zur Vermeidung von Fehlern (Nachbartypschränken)

Q0.1 Java Herunterladen

4 Softwaretechnologie (ST)

- ▶ Das Java Development Kit (JDK)
- ▶ <http://adaptopenjdk.java.net/>



If you have not yet downloaded Java, and started the compiler and the VM, you are already rather late and in danger to miss the train.

Def.: Ein Framework bildet eine besondere Art von wiederverwendbarer Komponente, denn es ist mit Löchern programmiert, an denen man es erweitern kann.

- ▶ Ein Framework hat immer einen essentiellen Kern-Ring und einen Erweiterungs-Ring (die Löcher). Es gibt 3 Sorten von Löchern, die wir betrachten werden:
 - fehlende Implementierungen ("Methodenlöcher") ("method hooks")
 - fehlende Typen ("Typlöcher") in generischen Klassen ("type hooks")
 - fehlende oder optionale Mixins ("Mixin-Löcher") in Großobjekten ("mixin hooks")
- ▶ Die Löcher machen Vorgaben für Erweiterungen, z.B. geben sie Typen oder Typschränken für Erweiterungen vor
 - Damit kann der Framework-Konstrukteur dem Anwendungs-Konstrukteur (Plugin-Konstrukteur) Vorgaben machen, um die Funktionsfähigkeit des Frameworks in der Anwendung zu garantieren
- ▶ Komponenten und Frameworks bilden die Basis aller objekt-orientierter Wiederverwendung in Firmen.

14.1 Schnittstellen und Abstrakte Klassen für das Programmieren von Löchern

Typen können verschiedene Formen annehmen. Eine partiell spezifizierte Klasse (Schnittstelle, abstrakte Klasse, generische Klasse) macht Vorgaben für Anwendungsentwickler

Schnittstelle:

Loch an Loch – und sie hält doch.

Eine “abstrakte Methode” ist ein
“Implementierungsloch” – ihre
Implementierung fehlt.

Eine Schnittstelle enthält ausschließlich
abstrakte Methoden.

Schnittstellen und Klassen in Java geben “Hooks” vor (“abstract”)

```
interface NamedThing {
    String getName(); // no implementation
}
abstract class Person implements NamedThing {
    String name;
    String getName() { return name; } // implementation exists
    abstract void drinkBeer(); // no implementation
}
abstract class Employee extends Person {
    abstract void getSalary(); // no implementation
}
class Professor extends Employee { // concrete class
    void getSalary() { goToBank(); withDraw(); }
    void drinkBeer() { .. select Guinness(); enjoy(); }
}
class Student extends Person { // concrete class
    void visitLecture() { stepIn(); payAttention(); }
    void drinkBeer() { .. select Radeberger(); enjoy(); }
}
```

Alle in einer Schnittstelle definierten Methodensignaturen bilden “Löcher”, “Haken”, “hooks”.

Eine konkrete Unterklasse muss sie alle füllen, also “Methoden einhängen”.

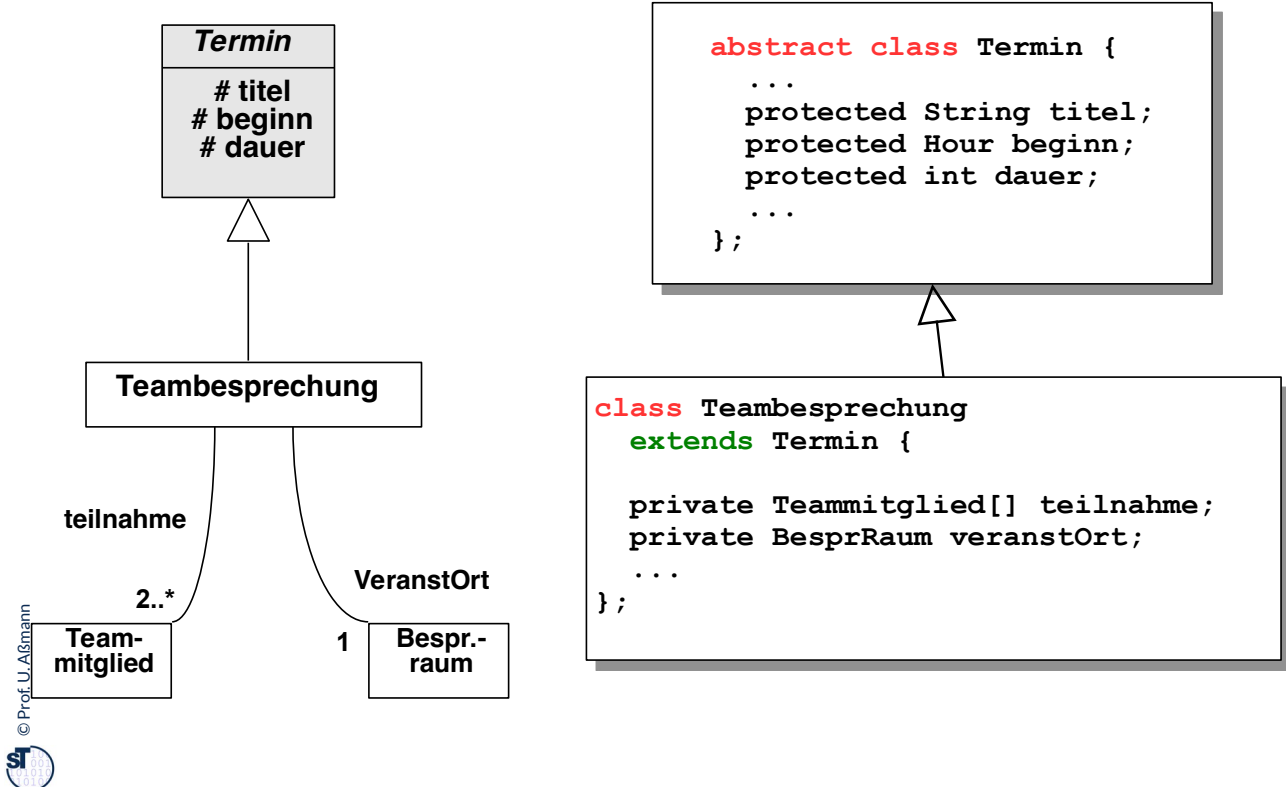
Schnittstellen und Klassen in Java geben "Hooks" (Löcher, Haken) vor

9 Softwaretechnologie (ST)

```
interface NamedThing {
    String getName(); // no implementation
}
abstract class Person implements NamedThing {
    String name;
    String getName() { return name; } // implementation exists
    abstract void drinkBeer(); // no implementation
}
abstract class Employee extends Person {
    abstract void getSalary(); // no implementation
}
class Professor extends Employee { // concrete class
    void getSalary() { goToBank(); withdraw(); }
    void drinkBeer() { .. select Guinness(); enjoy(); }
}
class Student extends Person { // concrete class
    void visitLecture() { stepIn(); payAttention(); }
    void drinkBeer() { .. select Radeberger(); enjoy(); }
}
class Alumnus extends Person {
    // new concrete class must fit to Person and NamedThing
    void workInComp() { .. serve... }
    void drinkBeer() { ...select Wine... }
}
```

Unterklassen dürfen neue Methoden definieren. Wenn sie aber eine Methodensignatur einer Schnittstelle füllen wollen, müssen sie dieser genau entsprechen.

Laufendes Beispiel Terminverwaltung



Die Klasse `Termin` definiert keine Methodensignaturen, aber geschützte Attribute. Auch hier können Unterklassen weitere Attribute hinzufügen.

Beispiel (2): Abstrakte Klassen und Abstrakte Operationen

11 Softwaretechnologie (ST)

```
abstract class Termin {  
    ...  
    protected String titel;  
    protected Hour beginn;  
    protected int dauer;  
    ...  
    abstract boolean verschieben (Hour neu);  
};
```

Jede abstrakt deklarierte Methode muß in einer Unterklasse realisiert werden - sonst können keine Objekte der Unterklasse erzeugt werden!

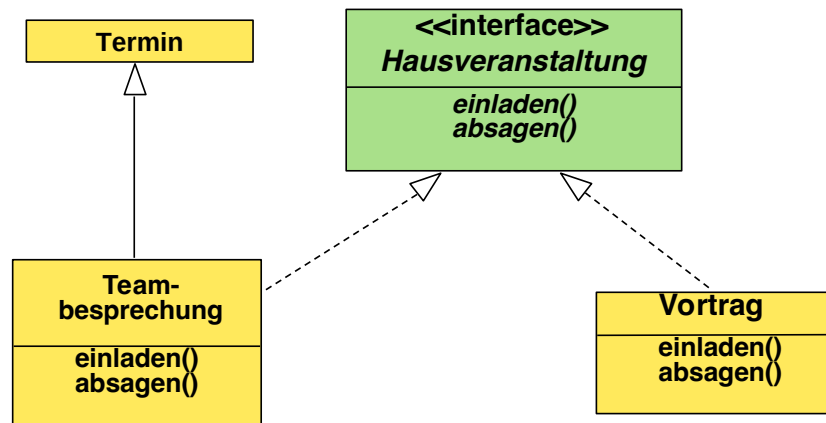
```
class Teambesprechung  
    extends Termin {  
    ...  
    boolean verschieben (Hour neu) {  
        boolean ok =  
            abstimmen(neu, dauer);  
        if (ok) {  
            beginn = neu;  
            raumFestlegen();  
        };  
        return ok;  
    };  
};
```

```
class Betriebsversammlung  
    extends Termin {  
    ...  
    boolean verschieben (Hour neu) {  
        beginn = neu;  
        raumFestlegen();  
        return ok;  
    };  
};
```

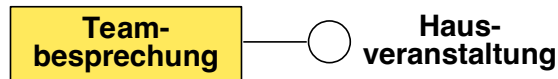
Fügt man eine abstrakte Methode hinzu, muss man auch die Klasse als abstrakt deklarieren.

Diese können dann in konkreten Unterklassen gefüllt werden.

Einfache Vererbung von Typen durch Schnittstellen



Hinweis: "Lutscher"-Notation (*lollipop*) für Schnittstellen „Klasse bietet Schnittstelle an“:



Das Abteil "angebotene Funktionen" kann in UML durch einen Lollipop benannt und abgekürzt werden.

Vergleich von Schnittstellen und abstrakte Klassen

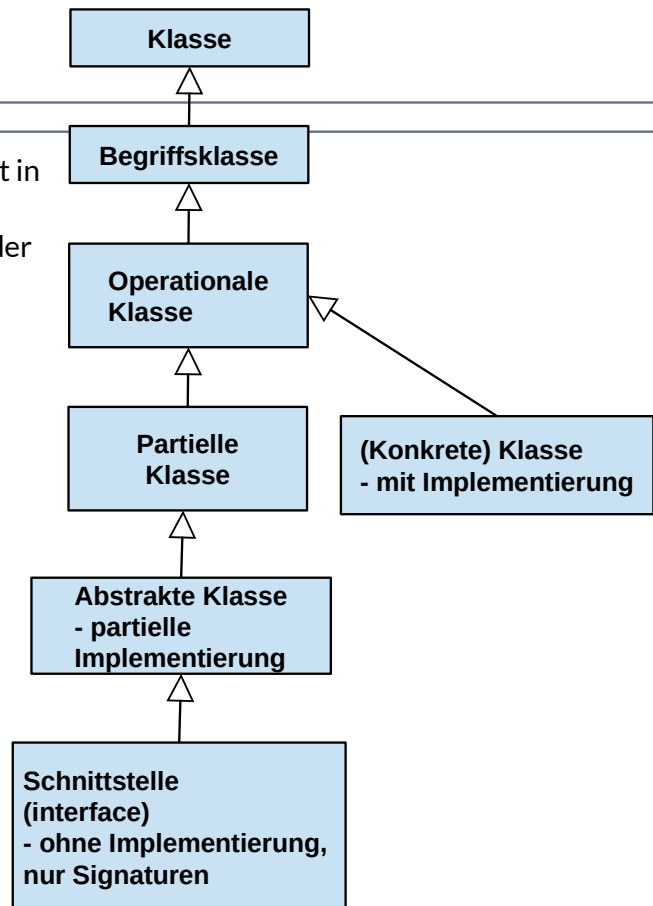
Abstrakte Klasse	Schnittstelle (voll abstrakt)
<p>Enthält Attribute und Operationen</p> <p>Kann Default-Verhalten festlegen</p> <p>Wiederverwendung von Schnittstellen und Code, aber keine Instanzbildung</p> <p>Default-Verhalten kann in Unterklassen überdefiniert werden</p> <p>Java: Unterklasse kann nur von einer Klasse erben</p>	<p>Enthält nur Operationen (und ggf. Konstante)</p> <p>Kann kein Default-Verhalten festlegen</p> <p>Redefinition unsinnig</p> <p>Java und UML: Eine Klasse kann mehrere Schnittstellen implementieren</p> <p>Schnittstelle ist eine spezielle Sicht auf eine Klasse</p>



Q2: Begriffshierarchie von Klassen (Erw.)

14 Softwaretechnologie (ST)

- ▶ *Operationale Klassen* werden unterteilt in Klassen mit, ohne, und mit Implementierung einer Untermenge der Operationen
- ▶ *Schnittstellen* und *Abstrakte Klassen* dienen dem Teilen von Typen und partiellem Klassen-Code

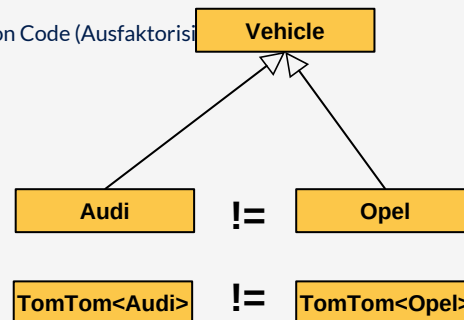


Wieder eine Taxonomie, diesmal von Klassen. Partielle Klassen geben vor, wie man Unterklassen bzw. Typen bildet.

14.2. Generische Klassen (Klassenschablonen, Template-Klassen, Parametrische Klassen)

... bieten eine weitere Art, mit Löchern zu programmieren, um Vorgaben zu machen

- Generische Klassen lassen den Typ von einigen Attributen und Referenzen offen ("generisch")
- Sie ermöglichen typisierte Wiederverwendung von Code (Ausfaktorisierung von Gemeinsamkeiten)
- Sie helfen, Nachbarn zu spezialisieren

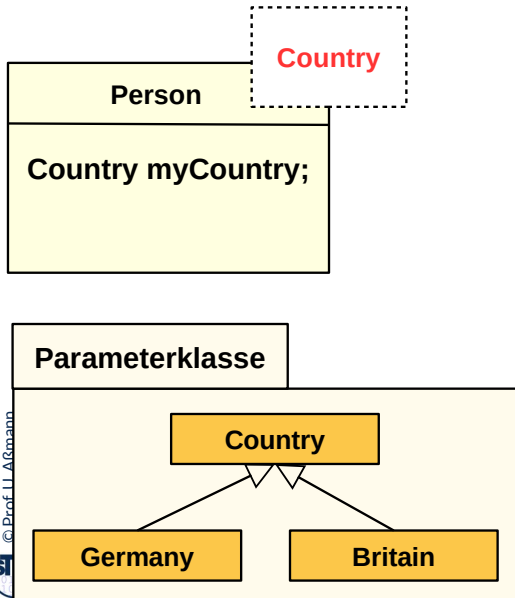


<https://www.uml-diagrams.org/template.html>

Generische Klassen bieten eine weitere Möglichkeit des “Programmierens mit Löchern”. Während abstrakte Klassen einzelne Methoden “offen” lassen, lässt eine generische Klasse einen ganzen Typ offen, i.d.R. den Typ eines Attributs seiner Objekte.

Def.: Eine **generische (parametrische, Template-) Klasse** ist eine Klassenschablone, die mit einem oder mehreren Typparametern (für Attribute, Referenzen, Methodenparameter) versehen ist.

► In UML



► In Java

- Sprachregelung: "Person of Country"

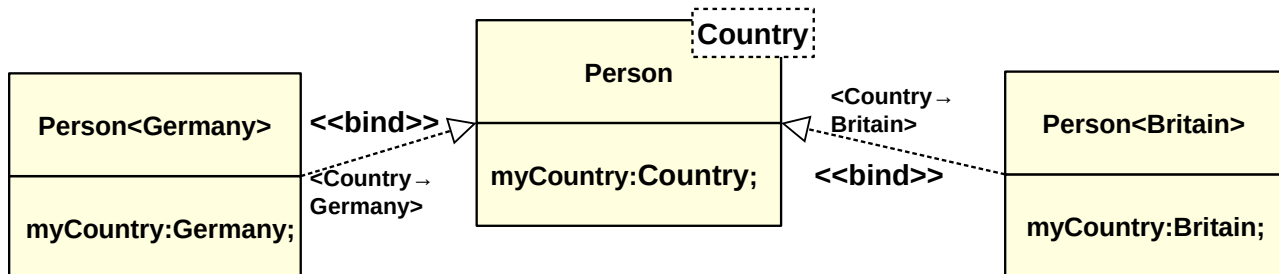
```
// Definition of a generic class
class Person<Country> {
    Country myCountry;
}
```

```
/* Type definition using a generic type */
Person<Germany> egon;
Person<Britain> john;
```

Generische Parameter sind "Typ-Löcher" – es fehlt ein genauer Typ.

Feinere statische Typüberprüfung für Attribute

- ▶ Zwei Attributtypen, die durch Parameterisierung aus einer generischen Klassenschablone entstanden sind, sind nicht miteinander kompatibel
- ▶ Der Übersetzer entdeckt den Fehler (**statische Typprüfung**)
- ▶ Die generische Klasse beschreibt das Gemeinsame (Generalisierung); der Parameter die Verschiedenheiten; die ausgeprägte Klasse die Spezialisierung



```
/* Type definition and initialization with object */
Person<Germany> egon = new Person<Germany>;
Person<Britain> john = new Person<Britain>;

/* Checks of assignments can use the improved typing */
john = egon;
```

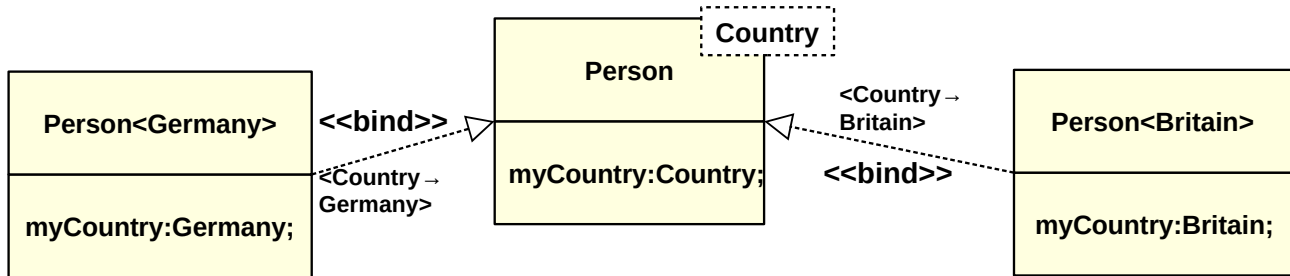
Die Parameterisierungsoperation heißt in UML “bind”:

- Ein Template wird an eine Instanz “gebunden” (“expandiert”), indem ein formaler Parameter des Templates an einen aktuellen Parameter gebunden wird

•

Einsatzzweck: Typen von Nachbarn vorgeben

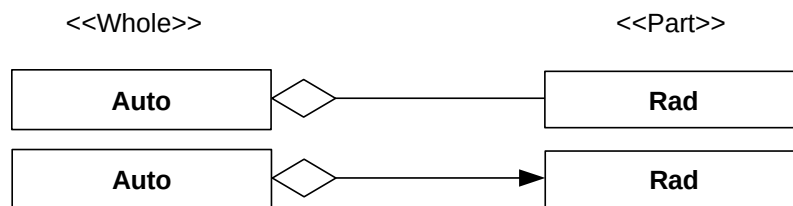
- ▶ **Def.:** Wenn ein formaler Typparameter einer generischen Klasse einen Nachbarn beschreibt, nennen wir ihn **Nachbartypschränke**



- ▶ **Bsp.:**
 - C gibt für die generische Klasse den Typ des Landes Country vor (Nachbar)
 - C ist eine Nachbartypschränke

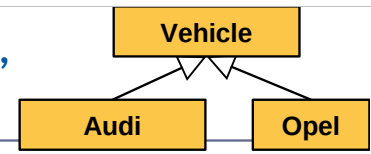
Einsatzzweck: Typen von Nachbarn vorgeben, Typsichere Aggregation (has-a)

- ▶ **Def.:** Wenn eine Assoziation den Namen „hat-ein“ oder „besteht-aus“ tragen könnte, handelt es sich um eine **Aggregation** zwischen einem *Aggregat*, dem *Ganzen*, und seinen *Teilen* (Ganzes/Teile-Relation, whole-part relationship).
 - Die auftretenden Aggregationen bilden auf den Objekten immer eine transitive, antisymmetrische Relation (einen gerichteten zyklensfreien Graphen, *dag*).
 - Ein Teil kann zu mehreren Ganzen gehören (*shared*), zu einem Ganzen (*owns-a*) und exklusiv zu einem Ganzen (*exclusively-owns-a*)
 - Exklusiv zugeeignete Objekte heißen **Unterobjekte**

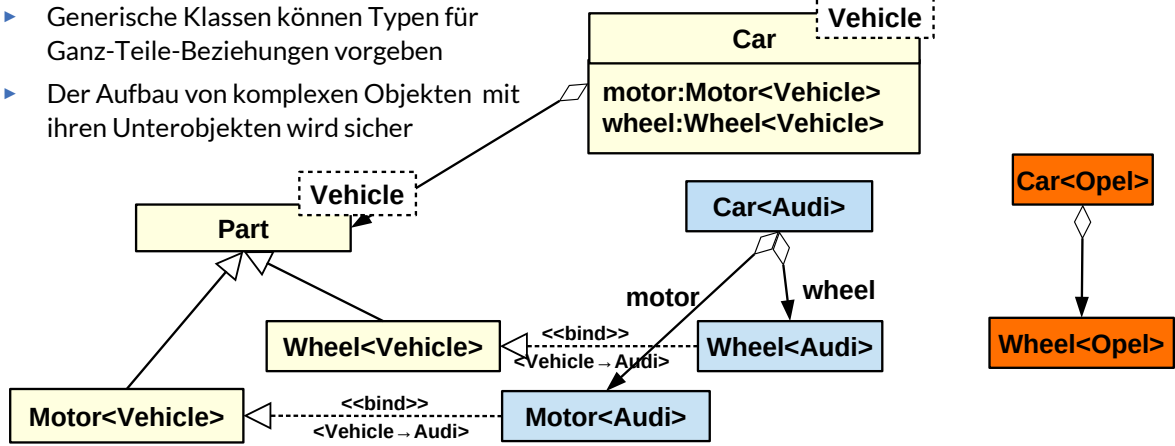


Lies: „Auto hat ein Rad“

Einsatzzweck: Typen von Nachbarn vorgeben, Typsichere Aggregation (has-a)



- ▶ Generische Klassen können Typen für Ganz-Teile-Beziehungen vorgeben
- ▶ Der Aufbau von komplexen Objekten mit ihren Unterobjekten wird sicher



```

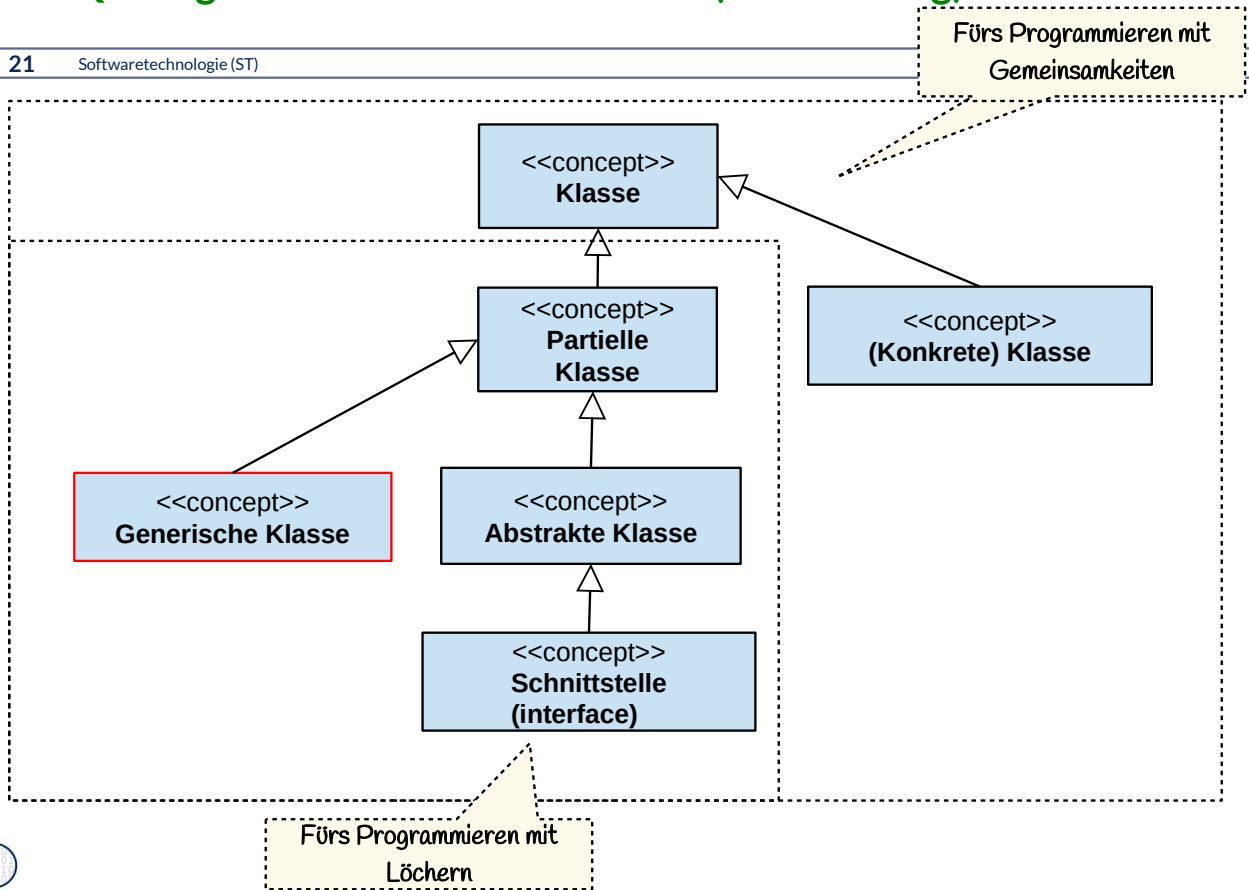
/* Type definition and initialization with object */
Motor<Audi> motorOfAudi = new Motor<Audi>;
Wheel<Audi> wheelOfAudi = new Wheel<Audi>;
Wheel<Opel> wheelOfOpel = new Wheel<Opel>;

/* Checks of assignments can use the improved typing */
Car<Audi> audi = new Car<Audi>();
audi.motor = motorOfAudi;
audi.wheel = wheelOfOpel;
    
```



Typparameter einer generischen Klasse werden zum feinen Typisieren von Attributtypen eingesetzt (Zeile 1-3). Die feinen Unterscheidungen der konkreten Typen helfen, Äpfel von Bananen zu unterscheiden und früh Fehler zu entdecken.

Q2: Begriffshierarchie von Klassen (Erweiterung)



Jetzt können wir verstehen, welche Unterkonzepte der “Klasse” zum Programmieren mit Löchern vorgesehen sind.

14.3. Kern-Klassen (Mixinhalter-Klassen) und ihre austauschbaren Mixins

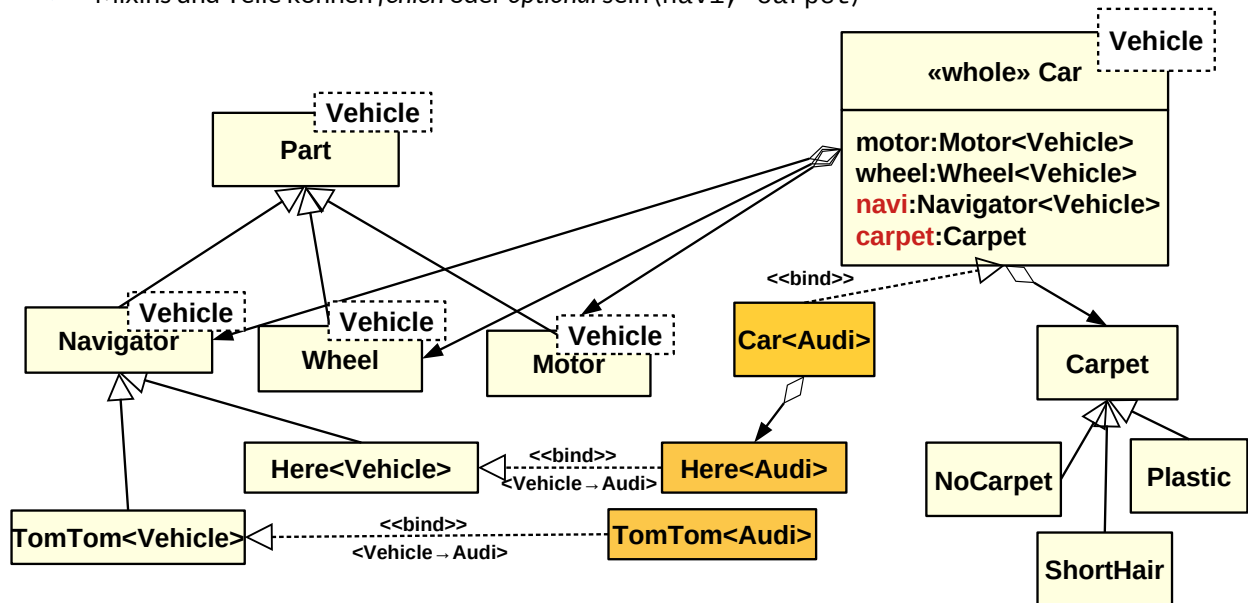
... Programmieren mit Löchern für Ganzes und Teile

- Kernobjekte haben Unterobjekte (Mixins), abhängige Unterobjekte
- Mixins können *fehlen*, d.h. optional sein oder nur temporär vorhanden (temporäre "Löcher")

Generische Klassen bieten eine weitere Möglichkeit des "Programmierens mit Löchern". Während abstrakte Klassen einzelne Methoden "offen" lassen, lässt eine generische Klasse einen ganzen Typ offen, i.d.R. den Typ eines Attributs seiner Objekte.

Fehlende oder optionale Teile werden durch Mixins dargestellt

- ▶ Def.: Ein **Mixin** ist ein Unterobjekt, das semantisch von einem Hauptobjekt (**Kern-Objekt**) abhängig ist. Die Klasse des Kerns heißt **Kernklasse**.
- ▶ Def.: Ein **Teil** ist ein Mixin, das ein Teil eines Ganzen darstellt, das durch der Kern repräsentiert wird.
- ▶ Mixins und Teile können *fehlen* oder *optional* sein (navi, carpet)



navi und carpet sind nicht unbedingt für jedes Auto notwendig (optionale Teile).

Oft können sie nachgekauft werden (fehlende Teile).

Mixins und Teile können durch generische Klassen getypt sein (navi), oder durch auch einfache Klassen (carpet).

- ▶ Def.: **Mixin-Polymorphie** entsteht, wenn ein Mixin seine Klasse gegen alternative Klassen austauscht (Wandel der Gestalt des Mixins)
- ▶ Mixins können optional sein, also oft fehlen (temporäre Löcher)

```
/* Type definition and initialization with object */
.. similar to before..
Carpet nocarp = new NoCarpet;
Carpet shairc = new ShortHair();
Carpet plastc = new Plastic();

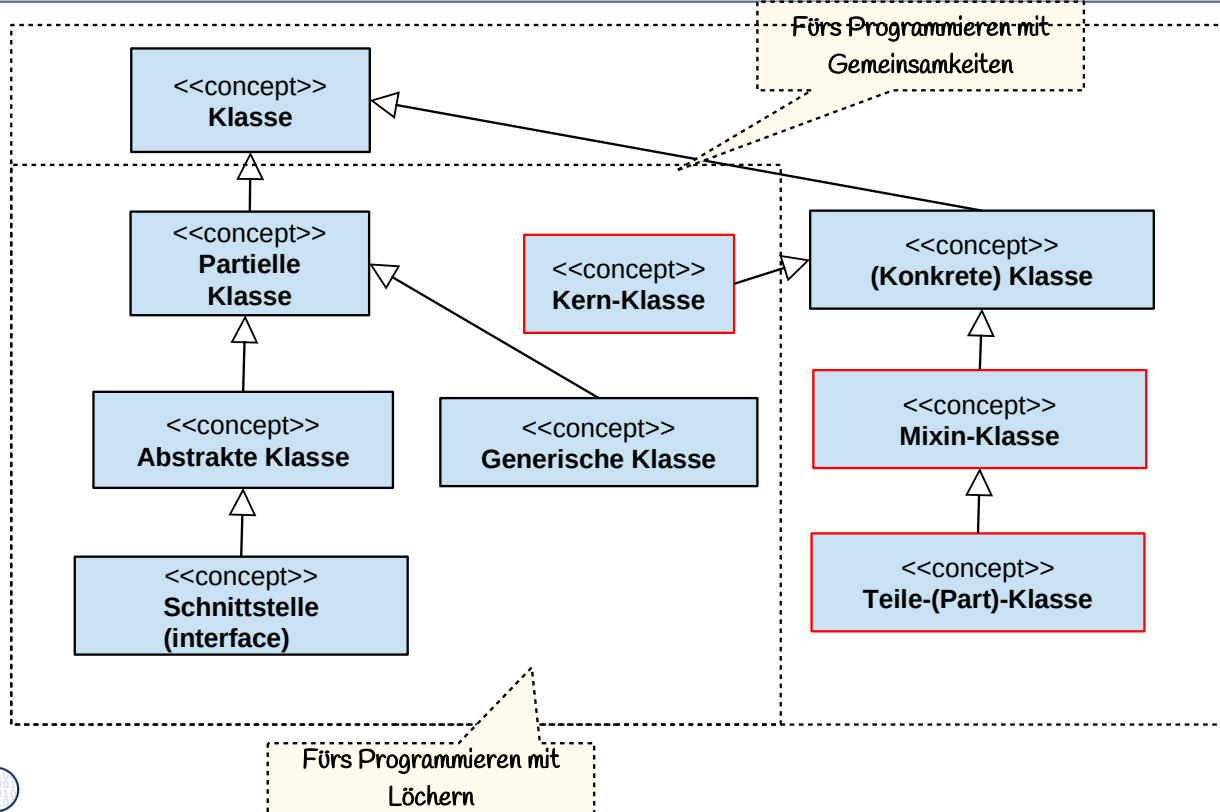
Navigator<Audi> navi1 = new Here<Audi>;
Navigator<Audi> navi2 = new TomTom<Audi>;

/* mixin polymorphy over navi and carpet */
Car<Audi> audi = new Car<Audi>();
audi.carpet = new NoCarpet();
audi.navi = navi1;      // now we enjoy Here
.... later in the life of the car ...
audi.carpet = plastc;   // now we have a short hair carpet
.... later in the life of the car ...
audi.carpet = shairc;   // now we have a short hair carpet
audi.navi = navi2;     // now we have a new navi, the TomTom
```

Mixins können für Mixin-Polymorphie genutzt werden, d.h. Mixins können einem Lebenszyklus unterliegen, z.B. von NoCarpet -> Plastic -> ShortHair.

Mixin-Polymorphie zeichnet sich dadurch aus, dass das Kernobject Car<Vehicle> stabil bleibt, aber ds Mixin seinen Gestalt (Typ) wechselt.

Q2: Begriffshierarchie von Klassen (Erweiterung)



Jetzt können wir verstehen, welche Unterkonzepte der "Klasse" zum Programmieren mit Löchern vorgesehen sind.

Eine Kernklasse ist keine partielle Klasse, ein Kernobjekt muss aber kein Mixin enthalten. Die Kernklasse muss aber ein Attribut für das Mixinobjekt vorhalten (das leer sein kann).

- ▶ Schnittstellen als auch abstrakte Klassen erlauben es, Anwendungsprogrammierern Struktur vorzugeben
 - Sie definieren “Haken”, in die Unterklassen konkrete Implementierungen schieben
 - Schnittstellen sind vollständig abstrakte Klassen
- ▶ Generische Klassen ermöglichen typsichere Wiederverwendung von Code über Typ-Parameter → der Compiler meldet mehr Fehler
- ▶ Kernobjekte haben Mixins für optionale oder fehlende Teile
- ▶ Mixinklassen können durch generische Parameter oder normale Klassen getypt sein

Warum ist das wichtig?

- ▶ **Bau von Frameworks (Rahmenwerken)** ist eines der Hauptprobleme des Software Engineering, weil es Wiederverwendung organisiert
 - Von Projekt zu Projekt
 - Von Produkt zu Produkt (Produktfamilien, Produktlinien)
- ▶ Abstrakte Klassen, Schnittstellen und generische Klassen können Code-Replikat und Code-Explosion weitgehend vermeiden und **gleichzeitig Vorgaben für Erweiterungen machen**
- ▶ Wiederverwendung **mit Frameworks** ist das Hauptmittel der Softwarefirmen, um **profitabel** arbeiten zu können

- ▶ Geben Sie eine Begriffshierarchie des Klassenbegriffs an. Welche Klassenarten kennen Sie? Wie spezialisieren sie sich?
- ▶ Erklären Sie den Unterschied der “Löcher” in abstrakten Klassen und in generischen Klassen.
- ▶ Gibt es einen Unterschied zwischen `{abstract}Employee` und *Employee* ?
- ▶ Warum wird die Wiederverwendung von Software durch Frameworks vereinfacht? Wozu gibt der Framework-Konstrukteur Vorgaben für die Löcher vor?
- ▶ Java hat kein Mixin-Sprachkonstrukt. Wie könnte man dennoch aufzeichnen, dass das Mixin semantisch von seinem Hauptobjekt abhängig ist?
 - Viele moderne Sprachen wie C# und Scala kennen Mixin-Vererbung
- ▶ Sollte ein Mixin privat oder öffentlich sein? Ist es gut, auf Attribute des Mixins direkt zugreifen zu können? Was passiert bei Polymorphie des Mixins?