Programmentwicklung II

Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Brandt

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2021

Aufteilung in Klassen

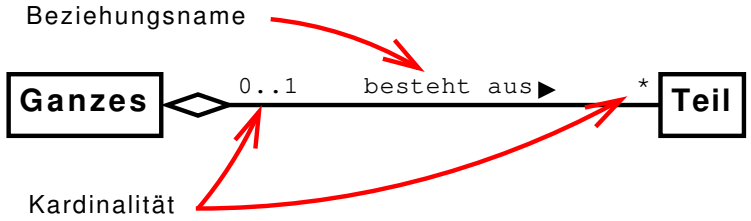
Welche Beziehungen gibt es zwischen den Klassen?

- *Vererbung:* Spezialisierung einer anderen Klasse.
- *Assoziation:* Man kennt und hilft sich, indem gegenseitig Methoden aufgerufen werden.
- *Aggregation und Komposition:* Objekte sind Teile eines anderen Objekts.

Wie werden diese Beziehungen

- in UML dargestellt?
- in C++ implementiert?

Aggregation



Eine Aggregation ist eine Relation zwischen Klassen, deren beteiligte Klassen eine Ganzes-Teile-Hierarchie darstellen.

Die *Kardinalität* gibt an, mit wie vielen Objekten der gegenüber liegenden Klasse das Objekt in Beziehung steht:

- 0..1 Ganzes besteht aus * Teilen

Beispiel:

- 1 Unternehmen besteht aus 1..* Abteilungen
- 1 Abteilung besteht aus 1..* Mitarbeitern

Aggregation

- Das Ganze (Aggregat) nimmt stellvertretend für seine Teile Aufgaben wahr:

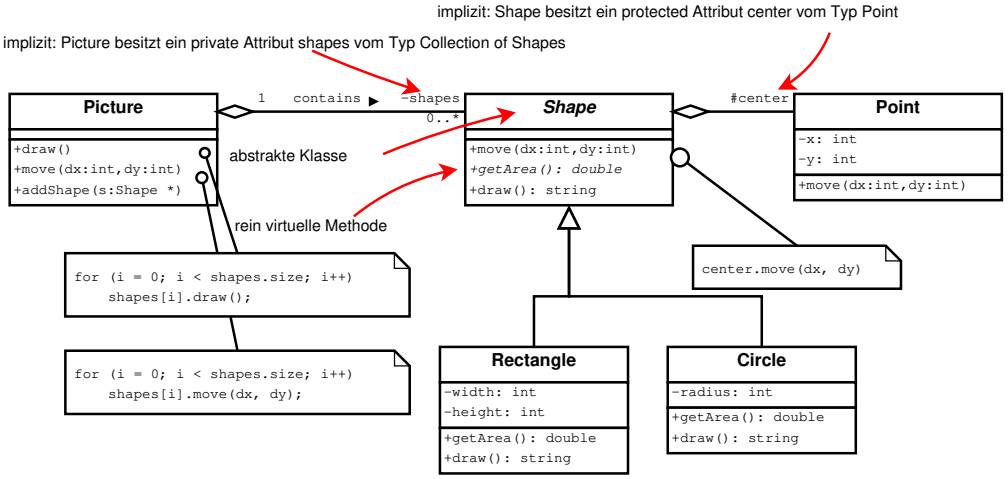
  Die Aggregatklasse enthält Operationen, die keine unmittelbare Veränderung im Aggregat selbst bewirken, sondern die Nachricht an seine Einzelteile weiterleitet.

$\Rightarrow$ Die beteiligten Klassen sind nicht gleichberechtigt:

Das Aggregat übernimmt stellvertretend für die Einzelteile die Verantwortung und Führung.

Was heißt das? Wie wird das in C++ implementiert?

Noch einmal eine Variante unseres Shape-Beispiels, diesmal erweitert um eine Klasse `Point`, die den Mittelpunkt eines geometrischen Objekts darstellen soll:



```
class Point {
    friend class Shape;
    // warum ist das notwendig???

private:
    int _x, _y;

public:
    Point(int x = 0, int y = 0) {
        _x = x;
        _y = y;
    }

    void move(int dx, int dy) {
        _x += dx;
        _y += dy;
    }
};
```

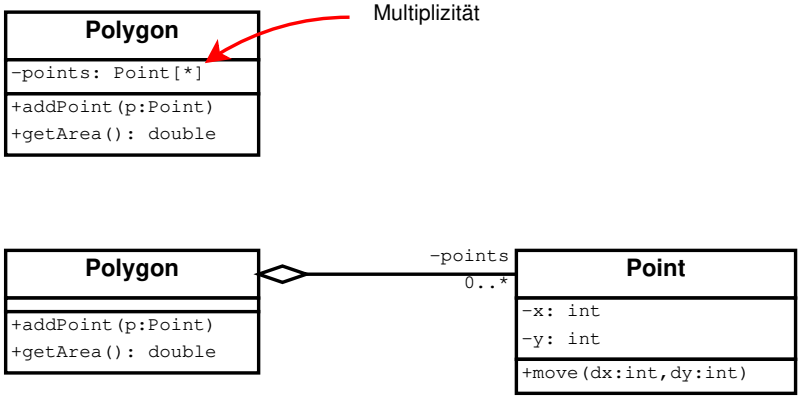
```
class Shape {
protected:
    Point _center;
public:
    Shape(Point p = 0) {
        _center = p;
    }
    virtual string draw() {   // Standard-Implementierung
        ostringstream os;
        os << "(" << _center._x << ", ";
        os << _center._y << ")" << endl;
        return os.str();
    }
    void move(int dx, int dy) {    // Delegation
        _center.move(dx, dy);
    }
    virtual double getArea() = 0;
};
```

```
class Rectangle: public Shape {
protected:
    int _w, _h;

public:
    Rectangle(Point p, int w, int h): Shape(p) {
        _w = w;
        _h = h;
    }
    string draw() {     // Basis-Implementierung + Erweiterung
        ostringstream os;
        os << "Rect:" << _w << "," << _h << ",";
        return os.str() + Shape::draw();
    }
    double getArea() {    // definiert getArea()
        return _w * _h;
    }
};
```

```
class Picture {
private:
    Liste<Shape *> shapes;
public:
    void addShape(Shape *s) {
        shapes.append(s);
    }
    string draw() {
        string res;
        for (int i = 0; i < _shapes.size(); i++)
            res += _shapes[i]->draw();
        return res;
    }
    void move(int dx, int dy) {
        for (int i = 0; i < shapes.size(); i++)
            shapes[i]->move(dx, dy);
    }
};
```

Wir können die Klasse `Point` auch für ein `Polygon` nutzen. Die beiden folgenden Darstellungen sind gleichwertig:



Wie die Speicherung der Punkte erfolgt (Liste, Array, ...), wird durch das Klassendiagramm nicht festgelegt. Aber die Art der Speicherung hat Auswirkungen darauf, wie effizient die Methoden der Klasse implementiert werden können.

Beide folgenden Implementierungen entsprechen den obigen Darstellungen:

```
class Polygon {
    Liste<Point> points;

public:
    void addPoint(Point p) {
        points.append(p);
    }
    double getArea() {
        .....
    }
};
```

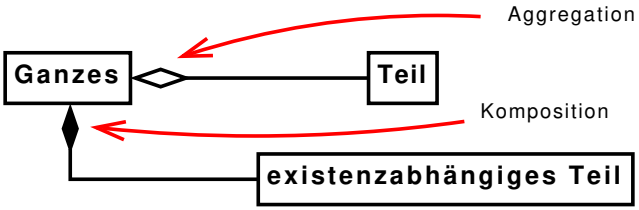
Werden die Punkte in einem Array gespeichert, ist die Implementierung komplizierter:

Aggregation

```
class Polygon {
    Point *points;
    int size, next;
public:
    Polygon() {
        next = 0;
        size = 8;
        points = new Point[size];
    }
    void addPoint(Point p) {
        if (next == size)
            .....  // resize array

        points[next] = p;
        next += 1;
    }
    double getArea();
};
```

Komposition ist eine strengere Form der Aggregation: die Teile sind vom Ganzen existenzabhängig, d.h. die Teile können ohne das Ganze nicht existieren. (nicht eindeutig in der Literatur)



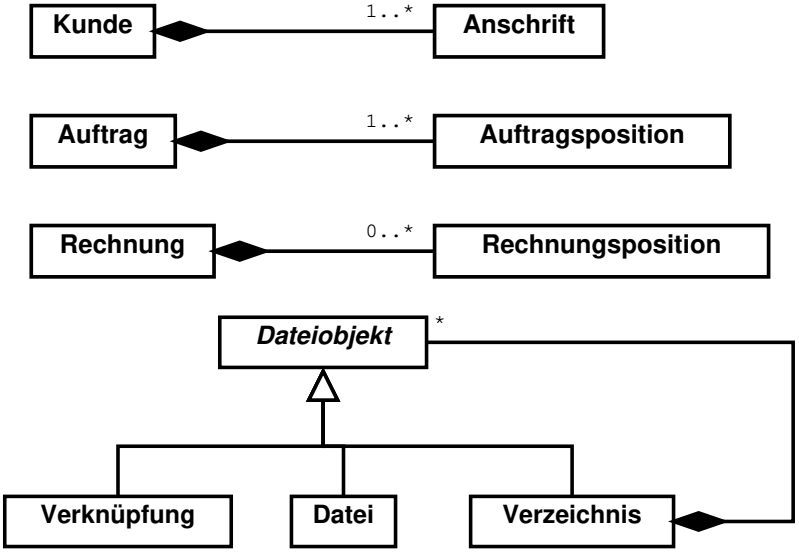
Das Ganze ist verantwortlich für das Erzeugen und Löschen der Teile.

- Beim Erzeugen des Aggregat-Objekts werden auch die einzelnen Teile erzeugt.
- Wird das Ganze gelöscht, werden automatisch auch alle seine Teile gelöscht.

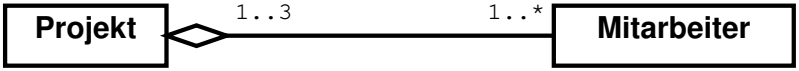
Ein Teil ist jedoch nicht untrennbar mit seinem Ganzen verbunden, sondern darf vorher einem anderen Aggregat-Objekt zugeordnet werden. Beispiel: Eine Datei kann in ein anderes Verzeichnis verschoben werden, bevor das Verzeichnis gelöscht wird.

Komposition

Beispiele:



Bei einer *Aggregation* ist die Kardinalität auf der Seite des Aggregats oft 1 (Standard, wenn Angabe fehlt). Das ist aber nicht zwingend so: Ein Teil kann gleichzeitig zu mehreren Aggregationen gehören.



Ein `Mitarbeiter` kann in bis zu drei Projekten arbeiten. Wird in C++ oft realisiert über Zeiger:

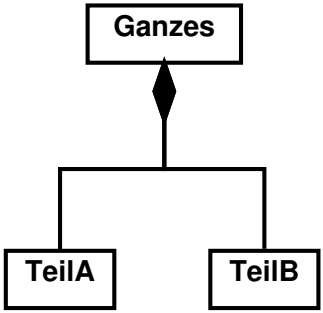
```
class Projekt {
private:
    Liste<Mitarbeiter *> mitarbeiter;
    .....
};
```

Warum realisieren wir das mittels Zeiger?

Bei einer *Komposition* kann die Kardinalität auf der Seite des Aggregats nur 1 sein. Jedes Teil ist Teil genau eines Kompositionsobjekts, sonst gäbe es einen Widerspruch zur Existenzabhängigkeit!

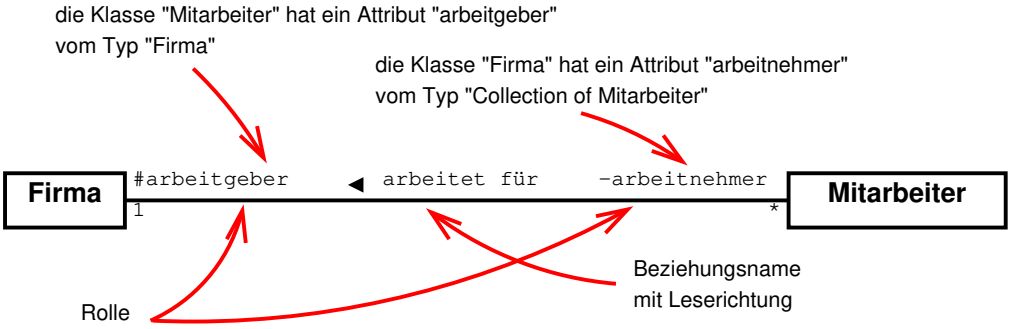
Realisierung in C++:

```
class Ganzes {
    TeilA _tA;
    TeilB *_tB;
public:
    Ganzes() {
        _tB = new TeilB();
    }
    ~Ganzes() {
        delete _tB;
    }
};
```



Die *Assoziation* ist eine schwächere Form der Aggregation: eine gleichrangige Beziehung zwischen Objekten, d.h. die Objekte kennen sich, haben aber keine stärkere Beziehung zueinander.

Beispiel:

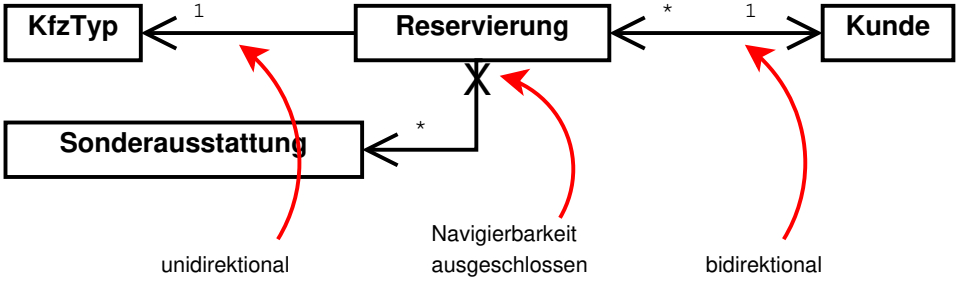


Hatten wir dieses Beispiel nicht schon einmal bei Aggregation?

Nochmal zur *Realisierung:* die beteiligten Klassen erhalten entsprechende Referenzattribute:

- Die Klasse `Mitarbeiter` hat ein Attribut `arbeitgeber` als Referenz auf ein Objekt der Klasse `Firma`.
- Die Klasse `Firma` hat ein Attribut `arbeitnehmer` mit einer Sammlung (engl.: Collection, z.B. Array oder Liste) von `Mitarbeiter`-Objekten.
- Diese Attribute werden nicht explizit in den jeweiligen Attribut-Rubriken der Klassen aufgeführt und die Attribute haben die bekannten Sichtbarkeitsattribute.

gerichtete Assoziation: Man kann von einer beteiligten Rolle zur anderen navigieren, aber nicht umgekehrt.



- Man kann von der `Reservierung` zum `KfzTyp` navigieren, die andere Richtung ist noch undefiniert.
- Man kann sowohl von `Kunde` zur `Reservierung` navigieren, als auch umgekehrt.
- Man kann von `Reservierung` zur `Sonderausstattung` navigieren, aber nicht umgekehrt.

Assoziation

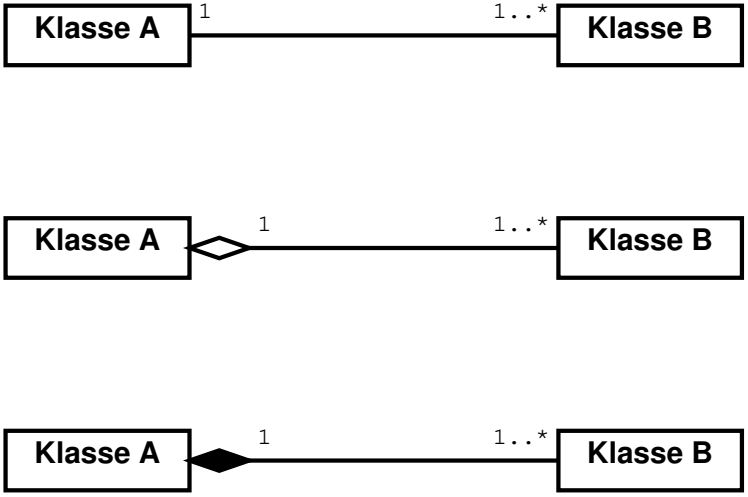
Problem: Oft ist die Abgrenzung zwischen Assoziation und Aggregation schwierig.

- Die Stabilität eines Software-Produkts ist nicht abhängig vom *richtigen* Symbol im Klassendiagramm (Raute oder nicht).
- **Aber:** Analysiere, wie eng die Beziehung zwischen den Objekten der Klassen ist und welche Konsequenzen sich daraus ergeben.

Beispiel: Die Beziehung zwischen `Rechnung` und `Anschrift` ist nur ein Verweis.

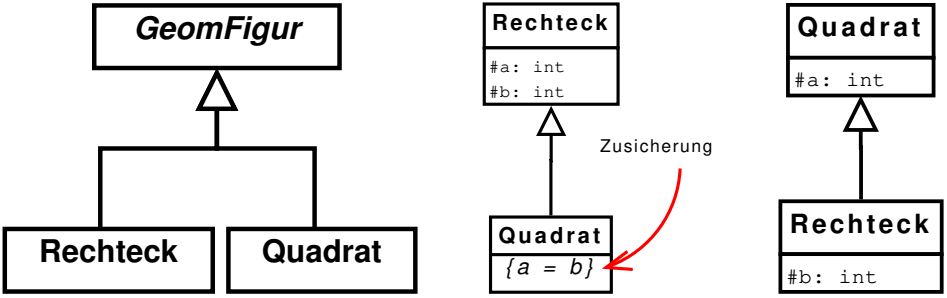
- Vorteil: Ist weniger speicherintensiv, als wenn jede `Rechnung` eine Kopie der `Anschrift` aggregiert.
- Nachteil: Eine Änderung der `Anschrift` hat auch Auswirkungen auf jede vergangene `Rechnung` des Kunden.

Was ist der Unterschied zwischen folgender Modellierung?



Vererbung ist nicht der Weisheit letzter Schluss. Zu oft setzen wir Vererbung ein, obwohl es nicht sinnvoll ist.

Wir werden uns einige Beispiele ansehen, wo Vererbung nicht angebracht ist und durch Delegation ersetzt werden sollte.



Vererbung kritisch prüfen:

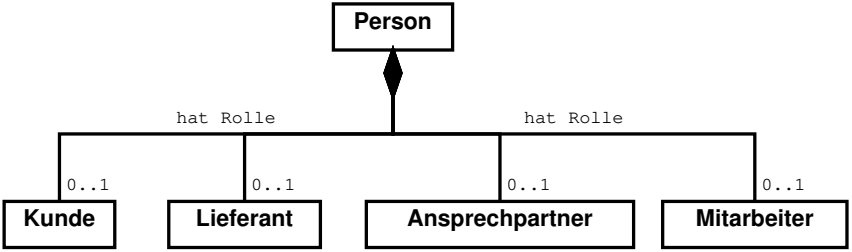
- In diesem Beispiel ist `Geschäftspartner` ein Oberbegriff für `Kunde`, `Lieferant`, `Ansprechpartner` und `Mitarbeiter` und
- die Begriffe `Unternehmen` sowie `Privatperson` existieren nicht eigenständig, sondern nur im Zusammenhang mit einem Geschäftspartner.
- Ein Kunde kann eine Privatperson oder ein Unternehmen sein.

Ist Vererbung hier angebracht? Nein!

- Lieferanten und Mitarbeiter können auch Kunden sein!
- Ein Objekt kann seine Klassenzugehörigkeit nicht wechseln, z.B. von Lieferant zu Kunde. Das wäre allerdings auch falsch: Der Geschäftspartner ist nicht abwechselnd mal Lieferant und mal Kunde, sondern beides gleichzeitig.

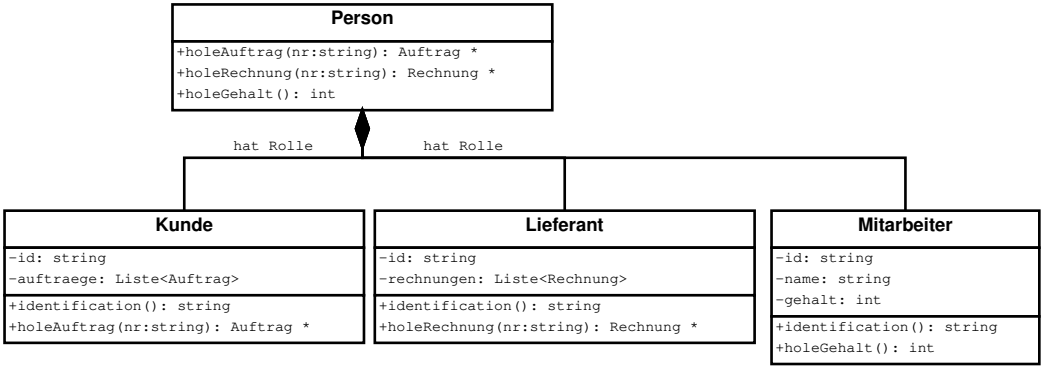
→ Kunde, Lieferant usw. sind keine Spezialisierungen von Geschäftspartner, sondern mögliche Eigenschaften!

Eine Rolle definiert eine spezielle Sichtweise auf ein Objekt: Die Anwender des Systems nehmen die Geschäftspartner in bestimmten Situationen in einer bestimmten Rolle wie Lieferant oder Kunde wahr.



Wie implementiert man das?

Vielleicht so?



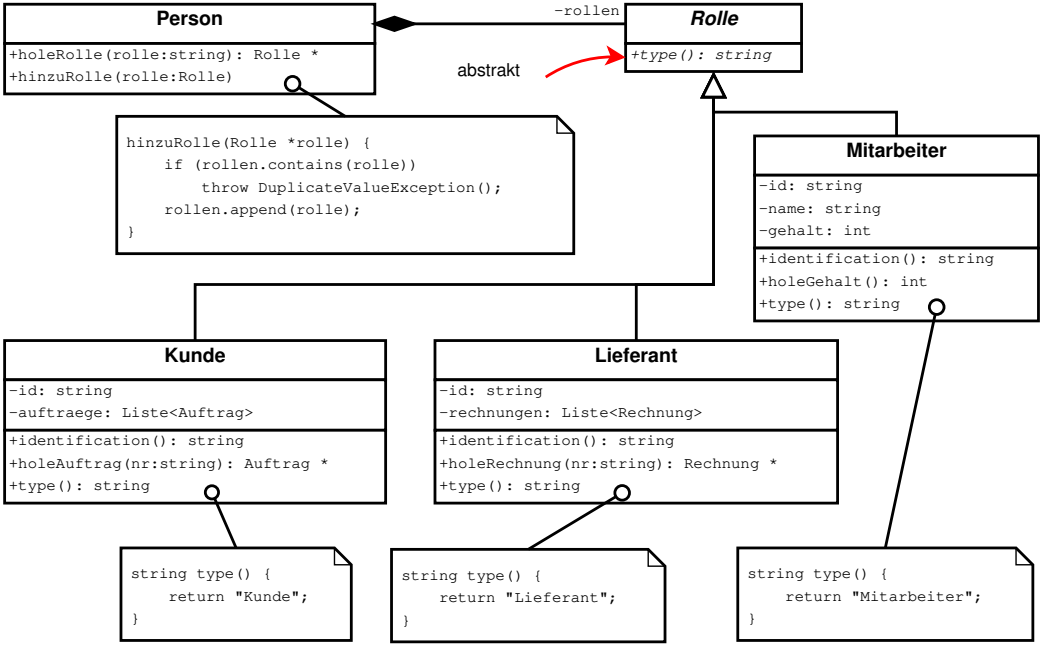

```
Person pers;
.....
Auftrag *auftr = pers.holeAuftrag("4711");
Rechnung *rech = pers.holeRechnung("0815");
```

Das wäre wohl nicht so günstig, denn

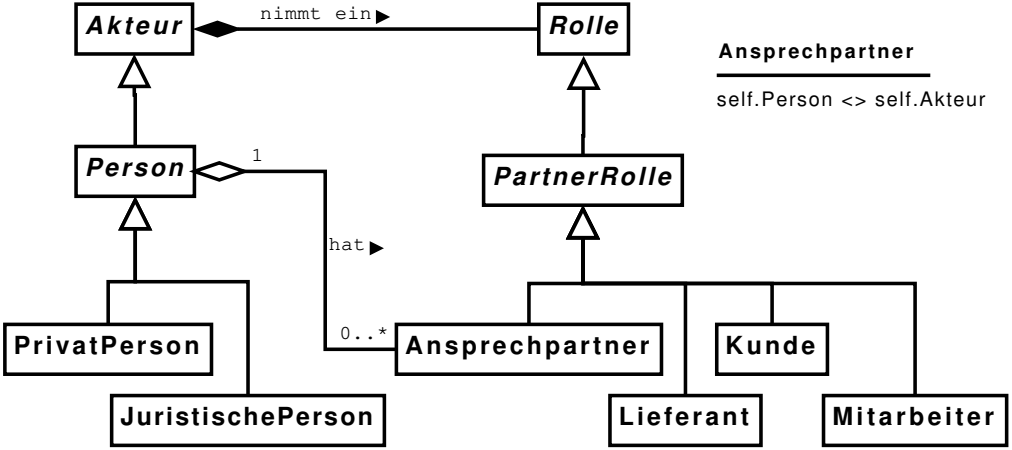
- die Klasse `Person` müsste jedesmal erweitert werden, wenn eine neue Rolle hinzu kommt.
- die Modellierung ist falsch: Die Methode `holeGehalt` ist eine Methode der Klasse `Mitarbeiter`, aber nicht der Klasse `Person`.

Wie sollen wir es denn dann implementieren?

Wie kann eine Person alle Rollen annehmen, also die Schnittstellen der Klassen `Kunde`, `Mitarbeiter`, `Lieferant` usw. bereitstellen?


Person
+holeRolle(rolle:string): Rolle *
+hinzuRolle(rolle:Rolle)
-rollen
Rolle
+type(): string
abstrakt
hinzuRolle(Rolle *rolle) {
if (rollen.contains(rolle))
throw DuplicateValueException();
rollen.append(rolle);
}
Mitarbeiter
-id: string
-name: string
-gehalt: int
+identification(): string
+holeGehalt(): int
+type(): string
Kunde
-id: string
-auftraege: Liste<Auftrag>
+identification(): string
+holeAuftrag(nr:string): Auftrag *
+type(): string
Lieferant
-id: string
-rechnungen: Liste<Rechnung>
+identification(): string
+holeRechnung(nr:string): Rechnung *
+type(): string
string type() {
return "Kunde";
}
string type() {
return "Lieferant";
}
string type() {
return "Mitarbeiter";
}

Akteur-Rolle-Entwurfsmuster



Zusicherung: Eine Person darf sich nicht selbst als Ansprechpartner referenzieren.

Frage: Warum ist die Klasse `Person` von `Akteur` abgeleitet?

Um gemeinsame Funktionalität für verschiedene Akteure bereitzustellen. So kann der Akteur auch eine Maschine mit unterschiedlichen Rollen sein. Die Basisklassen `Akteur` und `Rolle` bleiben erhalten, aber die darunterliegenden Hierarchien sind andere.

```
class Akteur {
private:
    Liste<Rolle *> rollen;
    .....
public:
    void hinzuRolle(Rolle *r) {
        if (rollen.contains(r))
            throw DuplicateValueException();
        rollen.append(r);
    }
    .....
};
```

Die Methode `hinzuRolle` muss in der Klasse `Person` und allen Unterklassen der Basisklasse nicht mehr implementiert werden.

Frage: Warum sind die Klassen `Kunde`, `Mitarbeiter` usw. von der Klasse `Rolle` abgeleitet?

Die Methode `hinzuRolle(Rolle *r)` der Klasse `Person` erwartet als Parameter eine *allgemeine Rolle*, da wir die konkreten Rollen zum Zeitpunkt der Definition von `Person` noch gar nicht kennen.

```
Person pers;
.....
pers.hinzuRolle(new Kunde(...));
pers.hinzuRolle(new Mitarbeiter(...));
pers.hinzuRolle(new Lieferant(...));
```

Frage: Wie kann eine `Person` gleichzeitig `Mitarbeiter` und `Kunde` sein?

```
Liste<Person *> kundenliste;
Liste<Person *> mitarbliste;
...
Adresse adr("Rosenweg", "7b", "01234", "Kaff");
Person *p = new Person("Meier", "Max", adr);
p->hinzuRolle(new Kunde(...));
p->hinzuRolle(new Mitarbeiter(...));
...
kundenliste.append(p);
mitarbliste.append(p);
```

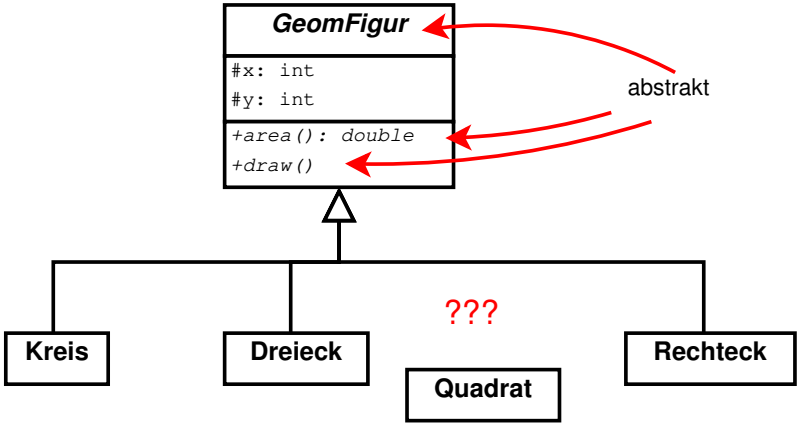
Hinweis: Akteur-Rolle-Entwurfsmuster wird nicht im Buch der *Gang of Four* beschrieben.

Ohne dieses Muster sähe es so aus:

```
Liste<Kunde *> kundenliste;
Liste<Mitarbeiter *> mitarbliste;
...
Adresse adr("Rosenweg", "7b", "01234", "Kaff");
Kunde *k = new Kunde("Meier", "Max", adr);
Mitarbeiter *m = new Mitarbeiter("Meier", "Max", adr);
...
kundenliste.append(k);
mitarbliste.append(m);
...
k.changeName("Schulze");
m.changeName("Schulze");     // nicht vergessen!
```

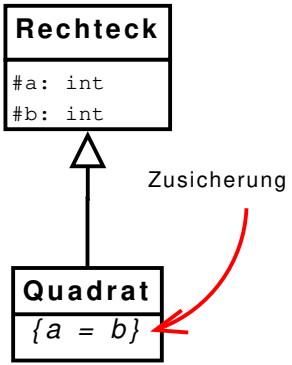
Vererbung kritisch prüfen:

Rechteck-Quadrat-Problem oder analog Ellipse-Kreis-Problem!



Ähnliche Frage: Ist eine reelle Zahl eine komplexe Zahl? Oder ist ein Pinguin ein Vogel, obwohl er nicht fliegen kann?

1. Versuch: Ein Quadrat *ist ein* Rechteck.



→ redundante Kanteninformation

Problem:
Wenn wir die Klasse `Rechteck` um eine Methode `resize(int dx, int dy)` erweitern, dann erbt die Klasse `Quadrat` diese Methode und ein Quadrat-Objekt kann zu einem Rechteck verzerrt werden.

Wir müssen alle gefährlichen Methoden von `Rechteck` durch überschreiben ausblenden.

Nach jeder Änderung in `Rechteck` ist zu prüfen, ob auch `Quadrat` zu ändern ist.

trozdem: Wir können ein `Quadrat` explizit in ein `Rechteck` umwandeln (type cast) und dann die `Rechteck`-Methode `resize` aufrufen.

Durch diesen Umweg können wir das `Quadrat` zu einem `Rechteck` verzerren!

Was lernen wir daraus?

> Keine Zusicherungen auf Attribute geben, die in der Oberklasse liegen!

Wir müssen bei all unseren Modellierungen das *Liskovsche Substitutionsprinzip* beachten: Objekte der abgeleiteten Klasse können stets an die Stelle von Objekten der Oberklasse treten.

In C++ kann eine Typumwandlung verhindert werden:

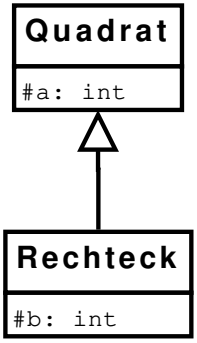
```
class Rechteck : public GeomFig {
protected:
    int _w, _h;
public:
    Rechteck(int w = 0, int h = 0) : _w(w), _h(h) {}

    Rechteck(const Rechteck& r) = default;
    Rechteck(Quadrat) = delete;
    Rechteck operator=(Quadrat) = delete;

    virtual int area() {
        return _w * _h;
    }

    virtual void resize(int fx, int fy) {
        _w *= fx;
        _h *= fy;
    }
};
```

2. Versuch: „technische Vererbung“ oder „Vererbung aus Bequemlichkeit“ (inheritance by convenience)



`Rechteck` ist eine Spezialisierung von `Quadrat`, weil es eine neue Eigenschaft ergänzt: die Länge der anderen Seite.

Problem:

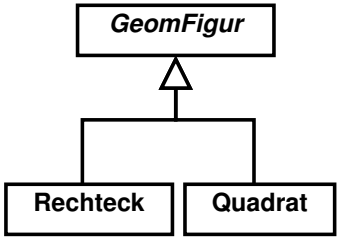
- Modellierung ist nicht intuitiv: Die Klasse verhält sich nicht erwartungskonform.

→ Bei einer Weiterentwicklung sind Fehler vorprogrammiert!

- Geerbte Methoden wie `area` gelten für `Rechteck` nicht.

→ Der Wartungsaufwand ist groß!

3. Versuch:



Problem: Unsere Klassen enthalten redundanten Code, denn ein `Quadrat` ähnelt einem `Rechteck` sehr.

Um den Wartungsaufwand klein zu halten, schreiben wir möglichst wenig redundanten Code, d.h. wir verwenden kein Copy-And-Paste!

Um redundanten Code zu vermeiden, können wir eine spezifische Basisklasse `Viereck` für `Quadrat` und `Rechteck` einführen.

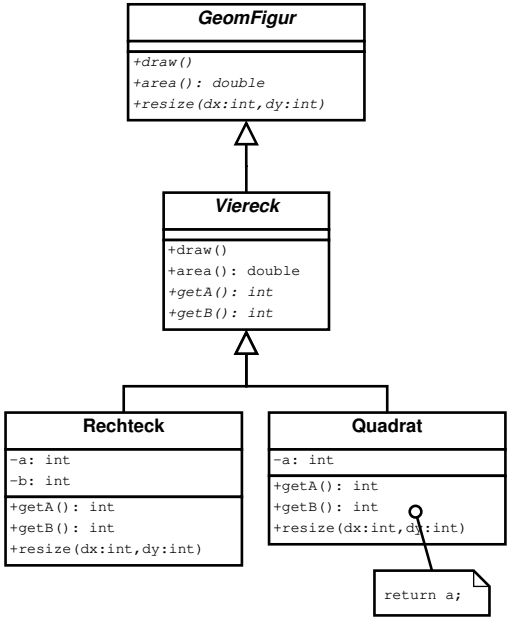
`Viereck` implementiert die gemeinsamen Methoden `draw` und `area` über Verwendung der Get-Methoden:

```
double Viereck::area(void) {
    return getA() * getB();
}
```

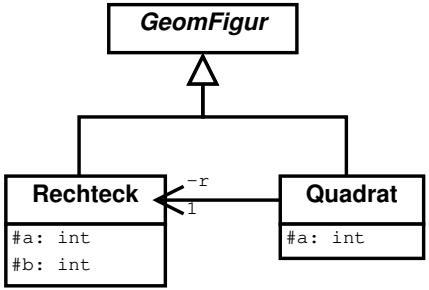
Damit das funktioniert, muss `getB` der Klasse `Quadrat` den Wert `a` liefern.

Die Methode `resize` muss jeweils in `Rechteck` und `Quadrat` implementiert werden.

Warum ist die Viereck-Lösung nicht geeignet? Welche Probleme gibt es bei dieser Art der Implementierung?



4. Versuch: Delegation



`Quadrat` bekommt ein eigenes, privates Attribut `r` vom Typ `Rechteck`

```
Quadrat::Quadrat(int a) {
    r = Rechteck(a, a);
    this->a = a;
}
```

`Quadrat` ruft alle brauchbaren Methoden von `Rechteck` über `r` auf (simple Einzeiler!)

```
double Quadrat::area() {
    return r->area();
}
```

Anmerkungen:

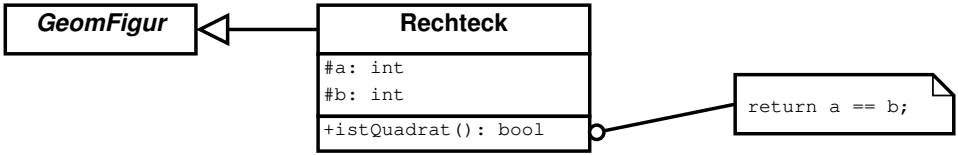
- `Quadrat`-spezifische Methoden werden dort realisiert, wo sie hingehören: innerhalb der Klasse `Quadrat`.
- Gute Lösung, solange die Basisklasse rein abstrakt ist.

Problem:

- Beim Einführen neuer Methoden wie bspw. `move` in `GeomFigur` müssen wir in `Quadrat` die Methode überschreiben und an `Rechteck` delegieren.

→ Wartungsaufwand ist groß!

5. Versuch: `Quadrat` als Eigenschaft von `Rechteck`



Problem: Dies führt zu `if/else`-Konstrukten, die wir durch Polymorphie vermeiden wollten!

Beispiel:

```
void Rechteck::resize(int dx, int dy) {
    if (istQuadrat() && dx != dy)
        throw "can not resize";
    ....
}
```

Tut mir leid: Ich habe keine Lösung für das Problem!