4.6 Vererbung von Konstruktoren

- Wenn eine Klasse die „gleichen“ Konstruktoren wie eine ihrer direkten Basisklassen besitzen soll, können diese mittels `using` geerbt werden.

- Jeder geerbte Konstruktor besitzt die gleichen Parameter wie der entsprechende Konstruktor der Basisklasse und ruft diesen mit den Werten der Parameter auf. Wenn er weitere Konstruktoren (von weiteren direkten Basisklassen, indirekten virtuellen Basisklassen oder eigenen Datenelementen) aufrufen muss, werden diese ohne Parameter aufgerufen. Wenn die zugehörigen Klassen dann keinen entsprechenden Konstruktor besitzen, wird der geerbte Konstruktor gelöscht. (Das heißt, man erhält nur dann eine entsprechende Fehlermeldung, wenn man einen solchen Konstruktor später verwenden will.)

- Das heißt insbesondere: Wenn es indirekte virtuelle Basisklassen gibt, die keine entsprechenden Konstruktoren besitzen, funktioniert die Vererbung von Konstruktoren nicht wie gewünscht.

- Prinzipiell können die Konstruktoren mehrerer Basisklassen geerbt werden sowie zusätzlich weitere Konstruktoren definiert werden.

Beispiel

```
// Eine beliebige Klasse.
struct A {
  ......
};

// Künstliche Zwischenklassen, damit AA zweimal von A erben kann.
struct A1 : A {
  // A1 erbt die Konstruktoren von A.
  using A::A;
};
struct A2 : A {
  // A2 erbt die Konstruktoren von A.
  using A::A;
};

struct AA : A1, A2 {
  ......
};
```

4.7 Zugriffsrechte

4.7.1 Private, halböffentliche und öffentliche Elemente und Basisklassen

- Eine Klasse kann mit den Schlüsselwörtern `public`, `proteced` und `private` (jeweils gefolgt von einem Doppelpunkt) in beliebig viele Abschnitte in beliebiger Reihenfolge unterteilt werden, deren Elemente jeweils *öffentlich*, *halböffentlich* bzw. *privat* sind.

- Der Abschnitt vor dem ersten solchen Schlüsselwort ist implizit privat, wenn die Klasse mit dem Schlüsselwort `class` definiert wird, andernfalls (Schlüsselwort `struct` oder `union`) öffentlich.

- Ebenso kann jede direkte Basisklasse einer Klasse öffentlich (Normalfall), halb-öffentlich oder privat deklariert werden.

- Wenn für eine Basisklasse keine Angabe gemacht wird, ist sie implizit privat, wenn die abgeleitete Klasse mit dem Schlüsselwort `class` definiert wird, andernfalls (Schlüsselwort `struct`) öffentlich. (Das heißt: Bei Verwendung des Schlüsselworts `class` muss man normalerweise alle Basisklassen explizit mit dem Schlüsselwort `public` deklarieren.)

4.7.2 Zugriffsrechte von Elementen direkter und indirekter Basisklassen

- Die öffentlichen und halböffentlichen Elemente einer öffentlichen direkten Basisklasse sind in der abgeleiteten Klasse ebenfalls öffentlich bzw. halböffentlich.

- Die öffentlichen und halböffentlichen Elemente einer halböffentlichen direkten Basisklasse sind in der abgeleiteten Klasse jeweils halböffentlich.

- Die öffentlichen und halböffentlichen Elemente einer privaten direkten Basisklasse sind in der abgeleiteten Klasse jeweils privat.

- Die privaten Elemente einer beliebigen direkten Basisklasse sind in der abgeleiteten Klasse zwar vorhanden, aber *gesperrt* (oder „eingesperrt“), d. h. nicht direkt zugänglich, ebenso die gesperrten Elemente einer beliebigen direkten Basisklasse.

- Die Elemente einer indirekten Basisklasse werden anhand dieser Regeln schrittweise in die abgeleitete Klasse übernommen, zum Beispiel:
  - Wenn `A` eine öffentliche direkte Basisklasse von `B` und `B` ein private direkte Basisklasse von `C` ist, sind die öffentlichen und halböffentlichen Elemente von `A` in `B` ebenfalls öffentlich bzw. halböffentlich und in `C` privat.
  - Wenn `A` eine private direkte Basisklasse von `B` und `B` ein öffentliche direkte Basisklasse von `C` ist, sind die öffentlichen und halböffentlichen Elemente von `A` in `B` privat und damit in `C` gesperrt.
  - Die privaten und gesperrten Elemente von `A` sind in jedem Fall in `B` und damit auch in `C` gesperrt.
  - Wenn direkte Basisklassen von Klassen immer öffentlich sind (Normalfall), bleiben die Zugriffsrechte von öffentlichen und halböffentlichen Elementen in allen abgeleiteten Klassen unverändert, während private Elemente in abgeleiteten Klassen grundsätzlich immer gesperrt sind.

- Wenn sich für ein Element einer virtuellen Basisklasse auf unterschiedlichen Ableitungswegen unterschiedliche Zugriffsrechte in der abgeleiteten Klasse ergeben, gilt das „Maximum“ dieser Zugriffsrechte, wobei `public` > `protected` > `private` > „gesperrt“ ist.
  Von einem Element einer replizierten Basisklasse gibt es in der abgeleiteten Klasse entsprechend mehrere „Exemplare“, von denen jedes unterschiedliche Zugriffsrechte besitzen kann.

- Eine Deklaration `using B::m` in einem öffentlichen/halböffentlichen/privaten Abschnitt der abgeleiteten Klasse macht das bzw. die Elemente mit dem Namen `m` der Basisklasse `B` jedoch zu entsprechenden Elementen der abgeleiteten Klasse (sofern sie nicht gesperrt sind).

- Ausnahme: Mittels `using B::B` geerbte Konstruktoren (vgl. § 4.6) besitzen in der abgeleiteten Klasse immer die gleichen Zugriffsrechte wie in der Basisklasse `B`.

- Eine indirekte Basisklasse ist öffentlich/halböffentlich/privat/gesperrt, wenn ein gedachtes öffentliches Element dieser Basisklasse in der abgeleiteten Klasse öffentlich/halböffentlich/privat/gesperrt wäre, das heißt:

  a) `B` ist eine öffentliche Basisklasse von `C`, wenn es im Ableitungsgraphen einen Weg von `C` nach `B` gibt, dessen Kanten alle öffentlich sind.

  b) `B` ist eine halböffentliche Basisklasse von `C`, wenn es keinen Weg gemäß a) gibt, aber einen Weg, dessen Kanten alle öffentlich oder halböffentlich sind.

  c) `B` ist eine private Basisklasse von `C`, wenn es keinen Weg gemäß a) und b) gibt, aber einen Weg, bei dem genau die erste Kante privat ist.

  d) Andernfalls (d. h. wenn alle Wege von `C` nach `B` eine private Kante enthalten, die nicht die erste Kante ist) ist `B` eine gesperrte Basisklasse von `C`.

  Wenn `B` eine replizierte Basisklasse von `C` ist, muss jedes „Exemplar“ von `B` separat betrachtet werden.

4.7.3 Zugängliche Basisklassen

- Eine öffentliche (direkte oder indirekte) Basisklasse einer Klasse `C` ist überall *zugänglich* (oder *zugreifbar*, engl. *accessible*).
- Eine halböffentliche Basisklasse von `C` ist nur innerhalb der Klasse `C` zugänglich sowie in abgeleiteten Klassen `D`, für die `C` eine öffentliche, halböffentliche oder private (d. h. keine gesperrte) Basisklasse ist.
- Eine private Basisklasse von `C` ist nur innerhalb der Klasse `C` zugänglich.
- Eine gesperrte Basisklasse von `C` ist nirgends zugänglich.
- Außerdem ist eine beliebige Basisklasse `B` von `C` an einer bestimmten Stelle zugänglich, wenn es eine an dieser Stelle zugängliche Basisklasse `Z` von `C` gibt und `B` wiederum eine an dieser Stelle zugängliche Basisklasse von `Z` ist.
- Eine implizite Aufwärtsumwandlung einer Klasse in eine (eindeutige) Basisklasse (vgl. § 4.2) ist an einer bestimmten Stelle nur möglich, wenn die Basisklasse an dieser Stelle zugänglich ist. Dasselbe gilt für explizite Aufwärtsumwandlungen mittels `static_cast`.
- Mit einem „klassischen C-Cast“ ist aber auch eine Aufwärtsumwandlung in eine nicht zugängliche Basisklasse möglich.

4.7.4 Zugängliche Elemente

- Ein öffentliches Element einer Klasse `C` ist überall zugänglich.
- Ein halböffentliches Element einer Klasse `C` ist nur innerhalb der Klasse `C` zugänglich sowie in abgeleiteten Klassen `D`, für die `C` eine öffentliche, halböffentliche oder private (d. h. keine gesperrte) Basisklasse ist.
- Ein privates Element einer Klasse `C` ist nur innerhalb der Klasse `C` zugänglich.
- Ein gesperrtes Element einer Klasse `C` ist nirgends zugänglich.
- Außerdem ist ein Element `m` einer Klasse `C` an einer bestimmten Stelle zugänglich, wenn es eine an dieser Stelle zugängliche Basisklasse `Z` von `C` gibt und `m` als Element von `Z` an dieser Stelle zugänglich ist.

4.7.5 Verwendung statischer Elemente

- Für ein statisches Element `m` einer Klasse `C` ist die Verwendung `C::m` an einer bestimmten Stelle erlaubt, wenn `m` ein an dieser Stelle zugängliches Element von `C` ist.
- Innerhalb der Klasse `C` ist `m` äquivalent zu `C::m`.

4.7.6 Verwendung nichtstatischer Elemente

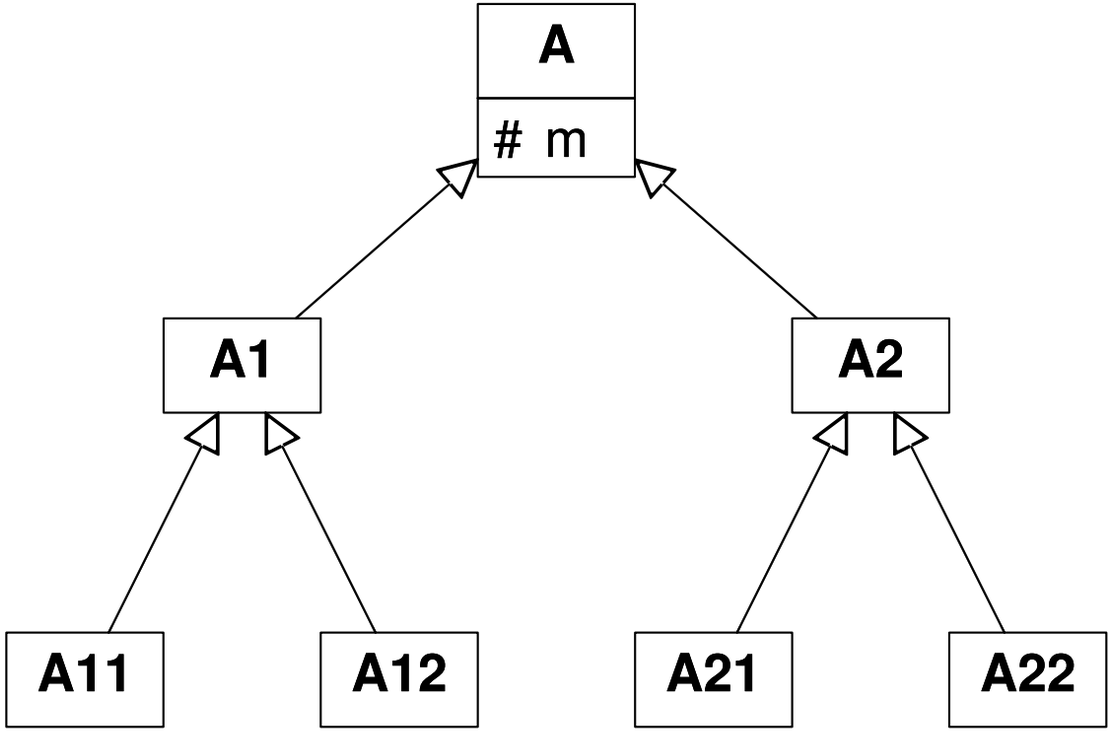
- Für ein nichtstatisches Element `m` einer Klasse `C` sowie ein Objekt `c` dieser Klasse bzw. einen Zeiger `p` auf ein solches Objekt ist die Verwendung `c.m` bzw. `p->m` an einer bestimmten Stelle erlaubt, wenn `m` ein an dieser Stelle zugängliches Element von `C` ist.

- Wenn der Name des Elements `m` mit dem Namen `B` einer Basisklasse qualifiziert ist (`c.B::m` bzw. `p->B::m`), muss sich das Zielobjekt `c` bzw. `p` implizit in diese Basisklasse umwandeln lassen und `m` muss ein an dieser Stelle zugängliches Element von `B` sein.

- Innerhalb einer Klasse ist `m` bzw. `B::m` äquivalent zu `this->m` bzw. `this->B::m`.

- Zusatzregel für halböffentliche nichtstatische Elemente:
  Wenn `m` ein halböffentliches Element von `C` (bzw. `B`) ist, darf es in einer bestimmten Klasse nur für Zielobjekte verwendet werden, die zu dieser Klasse oder einer von ihr abgeleiteten Klasse gehören.

❒ Beispiel:



❍ In der Klasse `A1` darf das halböffentliche nichtstatische Element `m` für Objekte der Klassen `A1...`, aber nicht für Objekte der Klassen `A` und `A2...` verwendet werden.

❍ Die Klassen `A1...` sind gerade diejenigen Klassen, an deren Implementierung die Klasse `A1` „beteiligt“ ist.

4.7.7 Befreundete Klassen und Funktionen

- Eine Klasse kann andere Klassen sowie Funktionen aller Art als *Freunde* deklarieren. (Eine Klasse oder Funktion kann sich aber nicht selbst zum Freund einer anderen Klasse machen.)

- Ein Freund einer Klasse besitzt die gleichen „Privilegien“ wie ein Element dieser Klasse, d. h. wenn etwas (ein Element oder eine Basisklasse irgendeiner Klasse) in einer bestimmten Klasse verwendet werden darf, dann darf es auch in den Freunden dieser Klasse verwendet werden. („Darf verwendet werden“ bedeutet aufgrund der in § 4.7.6 genannten Zusatzregel für halböffentliche nichtstatische Elemente u. U. mehr als nur „ist zugänglich“.)

- Ob sich eine Freundschaftsdeklaration in einem öffentlichen, halböffentlichen oder privaten Abschnitt einer Klasse befindet, spielt keine Rolle.

- Freundschaftsbeziehungen werden weder an abgeleitete Klassen vererbt noch sind sie transitiv.

4.7.8 Anmerkungen

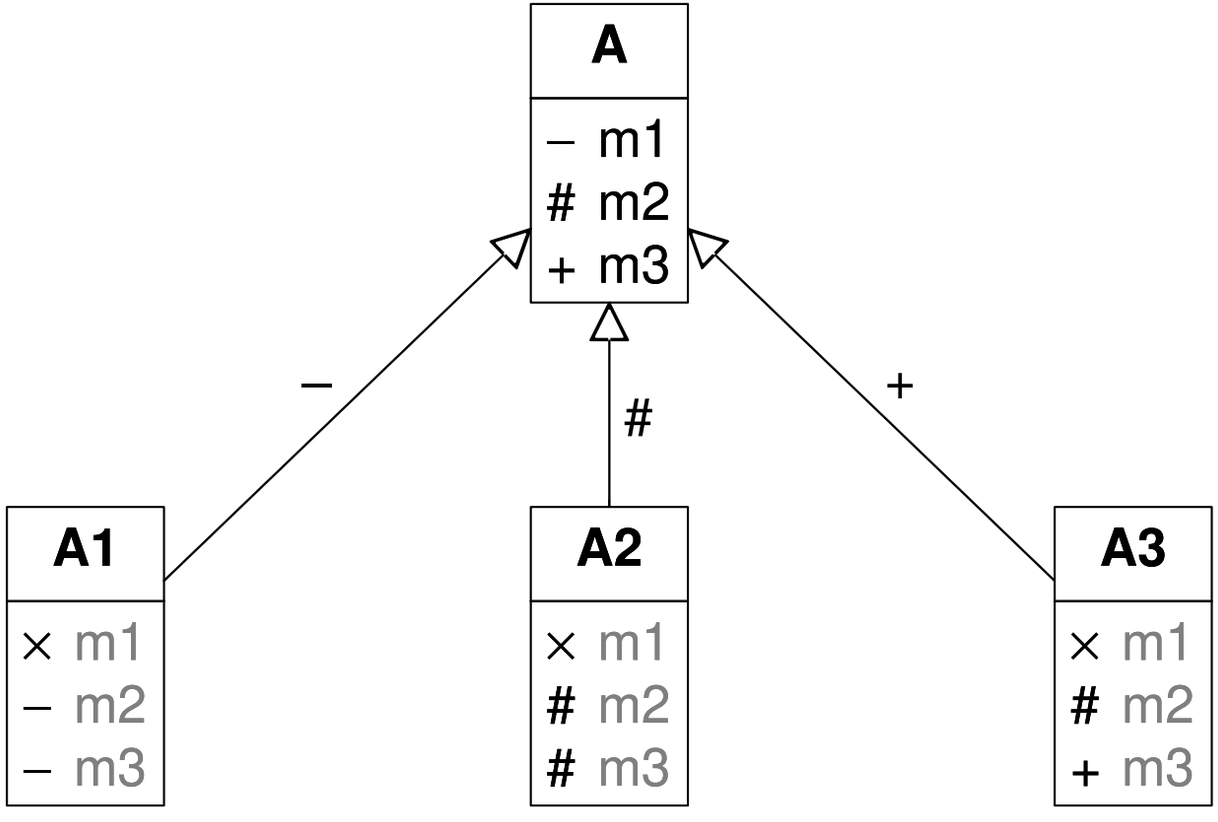
- Nicht zugängliche Elemente einer Klasse sind trotzdem *sichtbar* und werden z. B. bei der Auflösung überladener Funktionen berücksichtigt. Wenn die am besten passende Funktion dann aber nicht zugänglich ist, erhält man eine Fehlermeldung.

- Die Implementierung einer Elementfunktion gehört logisch zu ihrer Klasse, auch wenn sie sich textuell außerhalb der Klassendefinition befindet.

- Die Redefinition einer virtuellen Elementfunktion kann ein anderes Zugriffsrecht haben als die ursprüngliche Definition. Auch private und gesperrte Elementfunktionen können redefiniert werden.

- `c.B::m` bzw. `p->B::m` ist (abgesehen von eventuellen `const`- oder `volatile`-Qualifizierern) bezüglich der Zugriffsrechte äquivalent zu `static_cast<B>(c).m` bzw. `static_cast<B*>(p)->m`.
  Wenn `m` jedoch eine virtuelle Elementfunktion ist, wird der Funktionsaufruf in den Fällen mit `static_cast` dynamisch gebunden (weil der Name `m` unqualifiziert ist), während er bei Verwendung von `B::m` statisch gebunden wird (vgl. § 4.5.1).

4.7.9 Beispiel



- Die Abbildung zeigt, welche Zugriffsrechte die Elemente `m1` (privat), `m2` (halböffentlich) und `m3` (öffentlich) der Klasse `A` in den abgeleiteten Klassen `A1` (privat), `A2` (halböffentlich) und `A3` (öffentlich) besitzen.

- Das Kreuz steht hierbei für „gesperrt“, die graue Schrift zeigt an, dass es sich um geerbte Elemente handelt.

```
class A {
  void f ();
  int m1;
protected:
  int m2;
public:
  int m3;
} a;

class A1 : private A {
  void f ();
} a1;

class A2 : protected A {
  void f ();
} a2;

class A3 : public A {
  void f ();
} a3;
```

```
void A::f () {
  a1.m1; a1.m2; a1.m3; // Fehler; Fehler; Fehler;
  a2.m1; a2.m2; a2.m3; // Fehler; Fehler; Fehler;
  a3.m1; a3.m2; a3.m3; // OK;     OK;     OK;
}

void A1::f () {
  a1.m1; a1.m2; a1.m3; // Fehler; OK;     OK;
  a2.m1; a2.m2; a2.m3; // Fehler; Fehler; Fehler;
  a3.m1; a3.m2; a3.m3; // Fehler; Fehler; OK;
}

void A2::f () {
  a1.m1; a1.m2; a1.m3; // Fehler; Fehler; Fehler;
  a2.m1; a2.m2; a2.m3; // Fehler; OK;     OK;
  a3.m1; a3.m2; a3.m3; // Fehler; Fehler; OK;
}

void A3::f () {
  a1.m1; a1.m2; a1.m3; // Fehler; Fehler; Fehler;
  a2.m1; a2.m2; a2.m3; // Fehler; Fehler; Fehler;
  a3.m1; a3.m2; a3.m3; // Fehler; OK;     OK;
}
```

- In der Klasse `A` gilt:
  - `a3.m3` ist natürlich korrekt, weil `m3` in `A3` öffentlich ist.
  - Für alle anderen Kombinationen `aI.mJ` gilt:
    `mJ` ist in `AI` höchstens halböffentlich und deshalb gemäß der ersten drei Regeln in § 4.7.4 nur in `AI` oder eventuell in davon abgeleiteten Klassen zugänglich.
  - Für `Z` gleich `A` gilt jedoch:
    - `Z` ist eine öffentliche und deshalb überall zugängliche Basisklasse von `A3` (aber nicht von `A1` und `A2`).
    - `mJ` ist als Element von `Z` zugänglich.

    Deshalb ist `mJ` gemäß der vierten Regel in § 4.7.4 auch als Element von `A3` (aber nicht von `A1` und `A2`) zugänglich.
  - Die in § 4.7.6 genannte Zusatzregel ist für `a3.m2` ebenfalls erfüllt.

- In den Klassen `A1` und `A2` gilt zum Beispiel für `a3.m2`:
  - `m2` ist in `A3` halböffentlich und deshalb gemäß der ersten drei Regeln in § 4.7.4 nur in `A3` oder davon abgeleiteten Klassen zugänglich.
  - Gemäß der vierten Regel gilt jedoch wiederum mit `Z` gleich `A`:
    - `Z` ist eine öffentliche und deshalb überall zugängliche Basisklasse von `A3` (aber nicht von `A1` und `A2`).
    - `m2` ist als Element von `Z` zugänglich, weil `A1` und `A2` von `Z` abgeleitete Klassen sind.

    Deshalb ist `m2` auch als Element von `A3` zugänglich.
  - Die in § 4.7.6 genannte Zusatzregel ist für `a3.m2` jedoch nicht erfüllt.
  - Deshalb ist `a3.m2` jeweils fehlerhaft (und damit die vorigen Überlegungen, ob `m2` in `A3` zugänglich ist, unnötig).
  - Für `a3.A::m2` anstelle von `a3.m2` gilt (vgl. § 4.7.6):
    - `a3` lässt sich implizit in `A` umwandeln.
    - `m2` ist an dieser Stelle ein zugängliches Element von `A`.

    Trotzdem ist `a3.A::m2` fehlerhaft, weil die in § 4.7.6 genannte Zusatzregel wiederum nicht erfüllt ist.

- Für eine weitere Klasse

```
class A23 : public A2 { ..... };
```

gilt:

- `A` ist eine halböffentliche (indirekte) Basisklasse von `A23` und deshalb aufgrund der ersten drei Regeln in § 4.7.3 nur innerhalb von `A23` und davon abgeleiteten Klassen zugänglich.
- Für `Z` gleich `A2` gilt jedoch innerhalb von `A2`:
  - `Z` ist eine öffentliche und damit überall zugängliche Basisklasse von `A2`.
  - `A` ist eine in `A2` zugängliche Basisklasse von `A2`.

  Deshalb ist `A` in `A2` eine zugängliche Basisklasse von `A23`.

4.7.10 Adapter als Beispiel für private Basisklassen

```
#include <vector>
using IntVector = std::vector<int>;

class IntStack : private IntVector {
public:
  using IntVector::size;
  using IntVector::begin;  // begin und end werden für
  using IntVector::end;    // "range-based for loops" benötigt.
  void push (int x) { push_back(x); }
  int top () { return back(); }
  int pop () { int x = top(); pop_back(); return x; }
};

IntStack s;
s.push(5); s.push(7);
for (int x : s) cout << x << endl;  // "range-based for loop"
cout << s.pop() << endl;
```

- Die Tatsache, dass `IntStack` von `IntVector` abgeleitet ist, ist ein Implementierungsdetail, das für Klienten von `IntStack` irrelevant ist.
- Falls es später geändert wird, z. B. indem `list<int>` statt `vector<int>` verwendet wird oder eine Implementierung ohne Basisklasse verwendet wird, sollten Klienten davon nichts „merken“ (außer dass sie neu übersetzt werden müssen).
- Deshalb sollte `IntStack` (außerhalb der Klasse selbst) nicht polymorph als `IntVector` verwendbar sein.
- Außerdem sollen viele öffentliche Elementfunktionen von `IntVector` (z. B. `push_front` und `pop_front`) für Klienten von `IntStack` nicht verwendbar sein.
- Deshalb ist `IntVector` keine öffentliche, sondern eine private Basisklasse von `IntStack`.
  (Falls die Details von `IntVector` auch in einer von `IntStack` abgeleiteten Klasse verwendbar sein sollen, sollte `IntVector` stattdessen eine halböffentliche Basisklasse von `IntStack` sein.)
- Mittels `using` können einzelne Elemente von `IntVector` (konkret `size`, `begin` und `end`) trotzdem für Klienten von `IntStack` verwendbar gemacht werden.

- Die Verwendung eines privaten Datenelements mit Typ `B` anstelle einer privaten Basisklasse `B` zur Implementierung einer Klasse `C` hätte im allgemeinen folgende Nachteile:
  - `using` kann nicht verwendet werden.
    Stattdessen müsste man die gewünschten Elementfunktionen neu schreiben, zum Beispiel (wenn das Datenelement `v` heißt):

    ```
    IntVector::size_type size () { return v.size(); }
    ```

    Hierfür muss man die jeweiligen Funktionen wesentlich genauer kennen, um ihre Parameter- und Resultattypen korrekt angeben zu können.
    Für `begin` und `end` muss man außerdem wissen bzw. daran denken, dass es jeweils zwei derartige Funktionen gibt (eine mit und eine ohne `const`-Qualifizierer).
  - Wenn die Klasse `B` virtuelle Elementfunktionen hätte, dann könnten diese in der Klasse `C` nicht überschrieben werden.
  - Wenn die Klasse `B` abstrakt wäre, dann könnte man sie gar nicht als Typ eines Datenelements verwenden.

4.8 Schnittstellen (interfaces)

4.8.1 Prinzip

- Da eine Klasse mehr als eine Basisklasse besitzen kann, besteht keine Notwendigkeit für Schnittstellen wie in Java.
- Eine Klasse, die nur rein virtuelle Funktionen enthält, entspricht einer Schnittstelle in Java.
- Die `implements`-Beziehung zwischen einer Klasse und einer Schnittstelle in Java entspricht einer normalen Vererbungsbeziehung.

4.8.2 Schnittstellen- und Implementierungshierarchie

Problemstellung

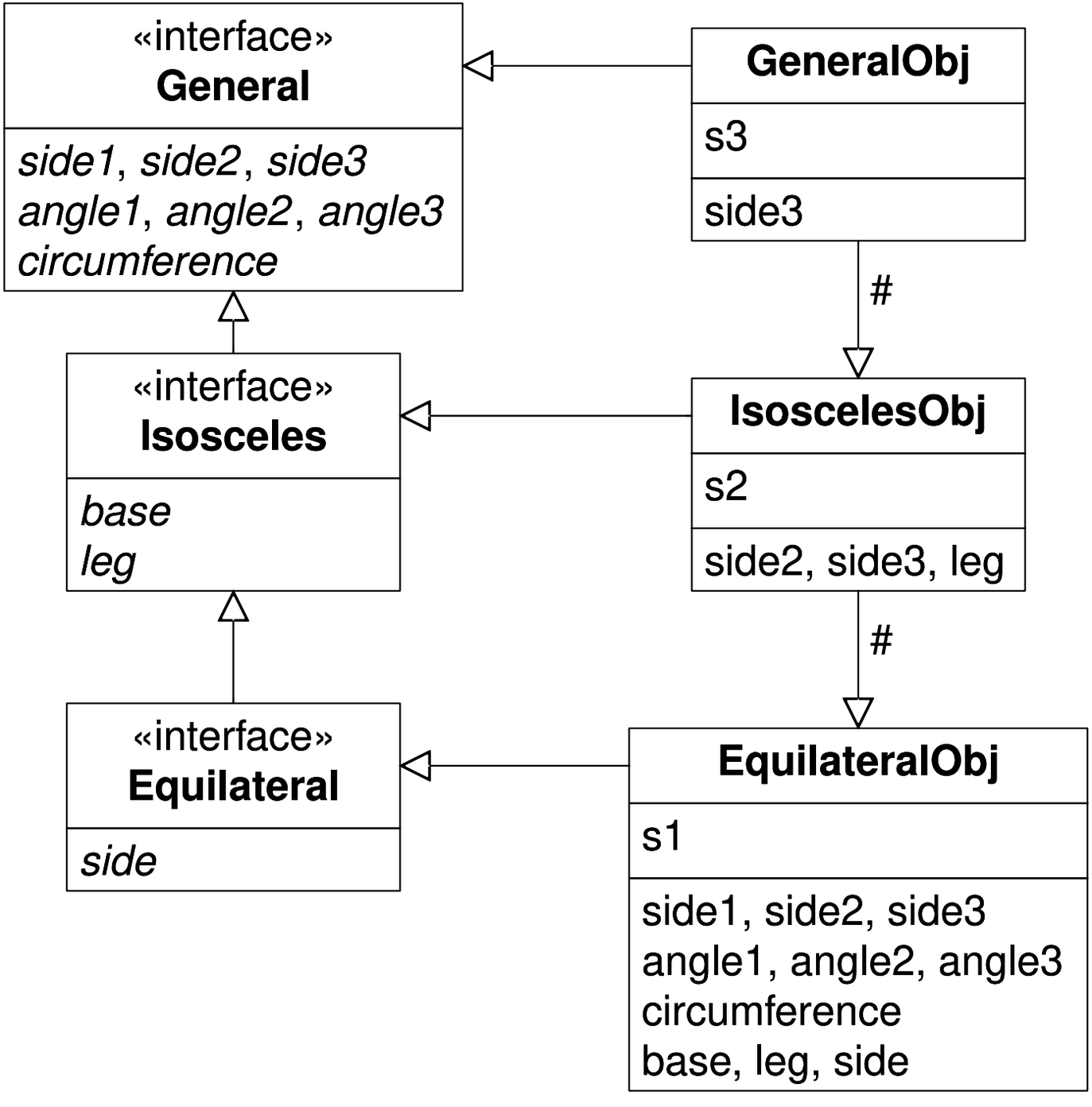
- Es sollen Klassen zur Repräsentation allgemeiner (`General`), gleichschenkliger (`Isosceles`) und gleichseitiger (`Equilateral`) Dreiecke mit sinnvollen Vererbungsbeziehungen definiert werden.

- Da jedes gleichseitige Dreieck auch ein gleichschenkliges und jedes gleichschenklige auch ein allgemeines Dreieck ist, sollten die Objekte der Klassen entsprechend polymorph verwendbar sein, das heißt: `Equilateral` → `Isosceles` → `General`.

- Andererseits werden zur Speicherung eines allgemeinen bzw. gleichschenkligen bzw. gleichseitigen Dreiecks drei bzw. zwei bzw. ein Datenelement benötigt, sodass die umgekehrte Vererbung `General` → `Isosceles` → `Equilateral` aus Implementierungssicht praktischer wäre.

Lösungsidee

- Es gibt separate Schnittstellen- und Implementierungshierarchien:


«interface»
General
side1, side2, side3
angle1, angle2, angle3
circumference
GeneralObj
s3
side3
#
«interface»
Isosceles
base
leg
IsoscelesObj
s2
side2, side3, leg
#
«interface»
Equilateral
side
EquilateralObj
s1
side1, side2, side3
angle1, angle2, angle3
circumference
base, leg, side

- Die Implementierungen der Elementfunktionen `side1`, `side2`, `side3`, `base`, `leg` und `side` in der Klasse `EquilateralObj` liefern alle den Wert des Datenelements `s1`. Dementsprechend müssen sie in den Klassen `IsoscelesObj` und `GeneralObj` teilweise überschrieben werden, um dort den Wert von `s2` bzw. `s3` zu liefern.

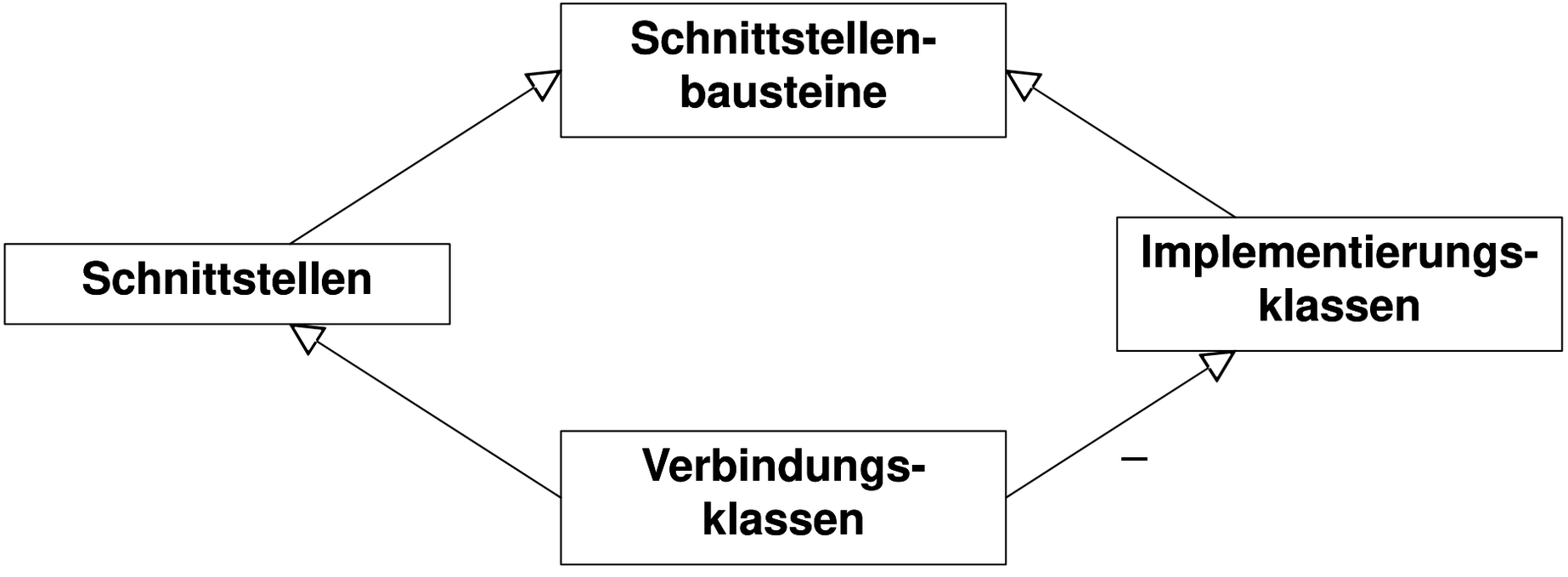
- Die Elementfunktionen `angle1`, `angle2`, `angle3` und `circumference` können in der Klasse `EquilateralObj` unter Verwendung von `side1`, `side2` und `side3` gleich allgemein implementiert werden, sodass sie für alle Arten von Dreiecken korrekt funktionieren.

- Durch die halböffentliche Vererbung zwischen den Implementierungsklassen können entsprechende Objekte immer nur an die „richtigen“ Schnittstellen zugewiesen werden. Beispielsweise kann ein Objekt der Klasse `IsoscelesObj` polymorph als `Isosceles` und als `General` verwendet werden, aber nicht als `Equilateral`.

- Trotzdem besitzt die Lösung noch einen gravierenden Mangel: Anhand der in § 4.5.3 genannten Regeln, wird ein dynamischer Typtest *jede* Art von Dreieck auch als `Isosceles` und `Equilateral` klassifizieren und dementsprechende Umwandlungen erlauben.

Verbesserung

- Um das zuvor genannte Problem zu vermeiden, darf es zwischen Implementierungsklassen und Schnittstellen *keinerlei* Vererbungsbeziehung geben.
- Stattdessen gibt es „zweiarmige Verbindungsklassen“ zwischen ihnen.
- Damit sich dann Schnittstellen und Implementierungsklassen auf *dieselben* virtuellen Funktionen beziehen, müssen diese in separate „Schnittstellenbausteine“ ausgelagert werden (die dann jeweils als virtuelle Basisklassen verwendet werden).
- Aufgrund der vollständigen Trennung zwischen Schnittstellen und Implementierungsklassen, können Vererbungsbeziehungen zwischen Implementierungsklassen auch wieder öffentlich sein.
- Die Vererbungsbeziehungen zwischen Verbindungs- und Implementierungsklassen sollten aber privat sein, damit Objekte der Verbindungsklassen nur an die zugehörigen Schnittstellen und nicht an die Implementierungsklassen zugewiesen werden können.
- Sowohl die Schnittstellenbausteine als auch die Implementierungsklassen können ggf. in kleinere Bestandteile oder „Komponenten“ zerlegt werden.


Schnittstellen-
bausteine
Schnittstellen
Implementierungs-
klassen
Verbindungs-
klassen

Umsetzungsmöglichkeit

Schnittstellenbausteine

```
#include <cmath>

#define interface struct

interface Sides {
  virtual double side1 () = 0;
  virtual double side2 () = 0;
  virtual double side3 () = 0;
};

interface Angles {
  virtual double angle1 () = 0;
  virtual double angle2 () = 0;
  virtual double angle3 () = 0;
};

interface Circumference {
  virtual double circumference () = 0;
};
```

```
interface IsoSides {
  virtual double base () = 0;
  virtual double leg () = 0;
};

interface EquSides {
  virtual double side () = 0;
};
```

Eigentliche Schnittstellen

```
interface General
    : virtual Sides, virtual Angles, virtual Circumference {};

interface Isosceles : General, virtual IsoSides {};

interface Equilateral : Isosceles, virtual EquSides {};
```

Implementierungsklassen

```
class AnglesImp : public virtual Sides, public virtual Angles {
  // Kosinussatz: s1*s1 = s2*s2 + s3*s3 - 2 * s2 * s3 * cos(a1)
  double angle (double s1, double s2, double s3) {
    return acos((s2*s2 + s3*s3 - s1*s1) / (2*s2*s3));
  }
  virtual double angle1 ()
                    { return angle(side1(), side2(), side3()); }
  virtual double angle2 ()
                    { return angle(side2(), side3(), side1()); }
  virtual double angle3 ()
                    { return angle(side3(), side1(), side2()); }
};

class CircumferenceImp
             : public virtual Sides, public virtual Circumference {
  virtual double circumference ()
                    { return side1() + side2() + side3(); }
};
```

```
class Side1Imp : public virtual Sides,
                 public virtual IsoSides, public virtual EquSides {
  double s1;
public:
  Side1Imp (double s) : s1{s} {}
  virtual double side1 () { return s1; }
  virtual double side2 () { return s1; }
  virtual double side3 () { return s1; }
  virtual double base () { return s1; }
  virtual double leg () { return s1; }
  virtual double side () { return s1; }
};

class Side2Imp : public Side1Imp {
  double s2;
public:
  Side2Imp (double b, double l) : Side1Imp{l}, s2{b} {}
  virtual double side2 () { return s2; }
  virtual double base () { return s2; }
};
```

```
class Side3Imp : public Side2Imp {
  double s3;
public:
  Side3Imp (double s1, double s2, double s3)
                                : Side2Imp{s1, s2}, s3{s3} {}
  virtual double side3 () { return s3; }
};
```

Verbindungsklassen

```
class EquilateralObj : public Equilateral, private Side1Imp,
                       private AnglesImp, private CircumferenceImp {
  using Side1Imp::Side1Imp;
};

class IsoscelesObj : public Isosceles, private Side2Imp,
                     private AnglesImp, private CircumferenceImp {
  using Side2Imp::Side2Imp;
};
```

```
class GeneralObj : public General, private Side3Imp,
                   private AnglesImp, private CircumferenceImp {
  using Side3Imp::Side3Imp;
};
```

Erzeugungsfunktionen

```
General* make_general (double s1, double s2, double s3) {
  return new GeneralObj{s1, s2, s3};
}

Isosceles* make_isosceles (double b, double l) {
  return new IsoscelesObj{b, l};
}

Equilateral* make_equilateral (double s) {
  return new EquilateralObj{s};
}
```

Anmerkungen

- Zwischen Schnittstellenbausteinen gibt es keine Vererbungsbeziehungen.
- Eine Implementierungsklasse wird (virtuell) von allen Schnittstellenbausteinen abgeleitet, deren Funktionen sie entweder implementiert oder verwendet.
- Zwischen Implementierungsklassen können beliebige Vererbungsbeziehungen bestehen.
- Bei Verwendung der Erzeugungsfunktionen („Fabrikmuster“) muss man die Implementierungs- und Verbindungsklassen gar nicht kennen. Wenn man ihre Konstruktoren privat oder halböffentlich macht und die Erzeugungsfunktionen als ihre Freunde deklariert, können die Konstruktoren ausschließlich von diesen Funktionen aufgerufen werden; damit bleiben diese Klassen dann vollkommen verborgen.