Hochschule Aalen
Fakultät Elektronik und Informatik
Studiengang Informatik

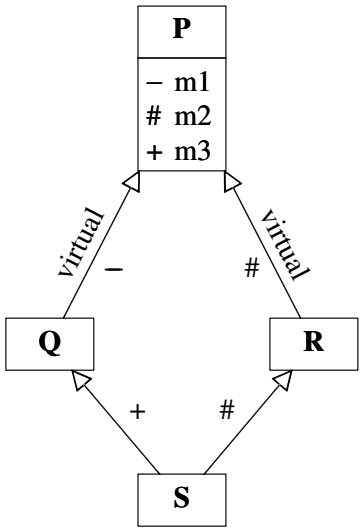
Programmieren in C++

Vorlesung im Wintersemester 2017/2018
Prof. Dr. habil. Christian Heinlein

6. Übungsblatt (7. Dezember 2017)

Aufgabe 10: Zugriffsrechte

Gegeben sei die folgende Klassenhierarchie:



Teilaufgabe 10.a)

Welche Zugriffsrechte besitzen die Elemente `m1`, `m2` und `m3` der Klasse `P` in den abgeleiteten Klassen `Q`, `R` und `S`?

✎

- `P` ist eine private Basisklasse von `Q`.
  Deshalb sind `m1`/`m2`/`m3` in `Q` isoliert/privat/privat.

- `P` ist eine halböffentliche Basisklasse von `R`.
  Deshalb sind `m1`/`m2`/`m3` in `R` isoliert/halböffentlich/halböffentlich.

- `P` wäre via `Q` eine isolierte indirekte Basisklasse von `S`, aber via `R` ist es eine halböffentliche indirekte Basisklasse.
  Deshalb sind `m1`/`m2`/`m3` in `S` isoliert/halböffentlich/halböffentlich.

✎

Teilaufgabe 10.b)

Gegeben seien Objekte `p`, `q`, `r` und `s` mit Typ `P`, `Q`, `R` bzw. `S`.

Welche der Elementverwendungen `p.m2`, `q.m2`, `r.m2` und `s.m2` sind innerhalb der Klassen `P`, `Q`, `R` und `S` erlaubt? Begründen Sie Ihre Antwort jeweils anhand der in der Vorlesung besprochenen Regeln!

Welche Änderungen ergeben sich, wenn `R` ein Freund von `Q` und/oder umgekehrt ist?

✎

- Ohne Freundschaftsbeziehungen:

Element-verwendung	in der Klasse			
	`P`	`Q`	`R`	`S`
`p.m2`	+	−	−	−
`q.m2`	−	+	−	−
`r.m2`	−	−	+	−
`s.m2`	−	+	−	+

- Begründungen:
  - In jeder Klasse `X` ist die Elementverwendung `x.m2` für `x` vom Typ `X` erlaubt, weil `m2` in jeder Klasse mindestens privat (also nicht isoliert) ist und deshalb in der Klasse selbst verwendet werden darf.
  - Die Verwendung `p.m2` in den Klassen `Q`, `R` und `S` ist aufgrund der Zusatzregel für halböffentliche nichtstatische Elemente nicht erlaubt.
  - Die Verwendung `q.m2` in den Klassen `P`, `R` und `S` ist nicht erlaubt, weil `m2` in `Q` privat und deshalb nur innerhalb von `Q` zugänglich ist.
  - Die Verwendung `r.m2` in den Klassen `P`, `Q` und `S` ist nicht erlaubt, weil `m2` in `R` halböffentlich und deshalb nur in `R` sowie in davon abgeleiteten Klassen (also `S`) zugänglich ist.
    Die Verwendung in `S` ist aber wieder aufgrund der Zusatzregel für halböffentliche nichtstatische Elemente nicht erlaubt.
  - Die Verwendung `s.m2` in den Klassen `P`, `Q` und `R` ist nicht erlaubt, weil `m2` in `S` halböffentlich und deshalb nur in `S` sowie in davon abgeleiteten Klassen (die es nicht gibt) zugänglich ist.
    Ausnahme: In der Klasse `Q` ist `s.m2` aufgrund der „Zwischenklassenregel“ mit `Z` gleich `Q` erlaubt: `Z` ist eine überall zugängliche Basisklasse von `S` und `m2` ist in `Q` als Element von `Z` zugänglich.

- Wenn `R` ein Freund von `Q` ist, ist in `R` zusätzlich `q.m2` und `s.m2` erlaubt, weil diese Verwendungen in `Q` erlaubt sind.

- Wenn `Q` ein Freund von `R` ist, ist in `Q` zusätzlich `r.m2` erlaubt, weil diese Verwendung in `R` erlaubt ist.

✐

Aufgabe 11: Schnittstellen- und Implementierungshierarchie

Teilaufgabe 11.a)

Definieren Sie folgende Schnittstellen mit sinnvollen Vererbungsbeziehungen unter Verwendung geeigneter „Schnittstellenbausteine“ (vgl. §4.8.10 der Vorlesung):

- Ein allgemeines geometrisches Objekt `Figure` besitzt Mittelpunktskoordinaten `x` und `y`, eine Breite `width`, eine Höhe `height` sowie eine Fläche `area`.
  `move(dx, dy)` verschiebt den Mittelpunkt in x-Richtung um `dx` und in y-Richtung um `dy`.
  `scale(f)` multipliziert Breite und Höhe jeweils mit dem Faktor `f`.

- Ein regelmäßiges geometrisches Objekt `RegularFigure` besitzt zusätzlich eine Größe `size`, die mit `width` und `height` übereinstimmt.

- Ein Rechteck `Rectangle` ist ein allgemeines geometrisches Objekt, das zusätzlich eine Diagonalenlänge `diag` besitzt.

- Ein Quadrat `Square` ist ein regelmäßiges Rechteck.

- Eine Ellipse `Ellipse` ist ein allgemeines geometrisches Objekt.

- Ein Kreis `Circle` ist eine regelmäßige Ellipse, der zusätzlich einen Durchmesser `diam` und einen Radius `rad` besitzt, wobei `diam` mit `size` übereinstimmt und `rad` die Hälfte davon ist.

Teilaufgabe 11.b)

Erstellen Sie modulare Implementierungskomponenten wie z. B. `CenterComp` zur Implementierung der Funktionen `x`, `y` und `move` für beliebige geometrische Objekte oder `RectangleComp` zur Implementierung der Funktionen `diag` und `area` für Rechtecke!

Teilaufgabe 11.c)

Definieren Sie zu den Schnittstellen `Rectangle`, `Ellipse`, `Square` und `Circle` jeweils eine zugehörige Implementierungsklasse, indem sie die zuvor erstellten Komponenten jeweils geeignet kombinieren!

Die Objekte dieser Klassen sollen nur die wirklich benötigten Datenelemente besitzen, d. h. die Mittelpunktskoordinaten sowie eine bzw. zwei Abmessungen bei regulären bzw. allgemeinen geometrischen Objekten.

Die Konstruktoren der Klassen sollen als Parameter die Abmessung(en) des Objekts erhalten und die Mittelpunktskoordinaten jeweils mit 0 initialisieren.

Teilaufgabe 11.d)

Definieren Sie die Konstruktoren der Implementierungsklassen privat oder halböffentlich, damit sie nicht direkt aufgerufen werden können.

Erstellen Sie stattdessen befreundete globale Erzeugungsfunktionen wie z. B. `make_rectangle`!

Lösung

```
#include <cmath>
#include <iostream>
using namespace std;

#define interface struct
```

Schnittstellenbausteine

```
interface FigureOps {
    virtual double x () = 0;
    virtual double y () = 0;
    virtual double width () = 0;
    virtual double height () = 0;
    virtual double area () = 0;
    virtual void move (double dx, double dy) = 0;
    virtual void scale (double f) = 0;
};

interface RegularFigureOps {
    virtual double size () = 0;
};

interface RectangleOps {
    virtual double diag () = 0;
};

interface CircleOps {
    virtual double diam () = 0;
    virtual double rad () = 0;
};
```

Eigentliche Schnittstellen

```
interface Figure : virtual FigureOps {};

interface Rectangle : virtual Figure, virtual RectangleOps {};

interface Ellipse : virtual Figure {};

interface RegularFigure : virtual Figure, virtual RegularFigureOps {};

interface Square : Rectangle, RegularFigure {};

interface Circle : Ellipse, RegularFigure, virtual CircleOps {};
```

Implementierungsklassen

```
const double PI_4 = acos(0) / 2;

class CenterComp : public virtual FigureOps {
    double _x = 0, _y = 0;
public:
    virtual double x () { return _x; }
    virtual double y () { return _y; }
    virtual void move (double dx, double dy) { _x += dx; _y += dy; }
};

class RegularComp
        : public virtual FigureOps, public virtual RegularFigureOps {
    double s;
protected:
    RegularComp (double s) : s(s) {}
public:
    virtual double width () { return s; }
    virtual double height () { return s; }
    virtual double size () { return s; }
    virtual void scale (double f) { s *= f; }
};

class GeneralComp : public RegularComp {
    double h;
protected:
    GeneralComp (double w, double h) : RegularComp(w), h(h) {}
public:
    virtual double height () { return h; }
    virtual void scale (double f) { RegularComp::scale(f); h *= f; }
};

class RectangleComp
            : public virtual FigureOps, public virtual RectangleOps {
    virtual double area () { return width() * height(); }
    virtual double diag () {
        double w = width(), h = height();
        return sqrt(w*w + h*h);
    }
};

class EllipseComp : public virtual FigureOps {
    virtual double area () { return PI_4 * width() * height(); }
};

class CircleComp
        : public virtual RegularFigureOps, public virtual CircleOps {
    virtual double diam () { return size(); }
    virtual double rad () { return diam() / 2; }
};
```

Verbindungsklassen

```
class RectangleObj : public Rectangle,
        private CenterComp, private GeneralComp, private RectangleComp {
    friend Rectangle* make_rectangle (double w, double h);
    using GeneralComp::GeneralComp;
};

class EllipseObj : public Ellipse,
        private CenterComp, private GeneralComp, private EllipseComp {
    friend Ellipse* make_ellipse (double w, double h);
    using GeneralComp::GeneralComp;
};

class SquareObj : public Square,
        private CenterComp, private RegularComp, private RectangleComp {
    friend Square* make_square (double s);
    using RegularComp::RegularComp;
};

class CircleObj : public Circle,
        private CenterComp, private RegularComp,
        private EllipseComp, private CircleComp {
    friend Circle* make_circle (double d);
    using RegularComp::RegularComp;
};
```

Erzeugungsfunktionen

```
Square* make_square (double s) {
    return new SquareObj(s);
}

Rectangle* make_rectangle (double w, double h) {
    return new RectangleObj(w, h);
}

Circle* make_circle (double d) {
    return new CircleObj(d);
}

Ellipse* make_ellipse (double w, double h) {
    return new EllipseObj(w, h);
}
```

Test

```
void print (Figure* f) {
    cout << f->x() << " " << f->y() << " "
        << f->width() << " " << f->height() << " "
        << f->area();
    if (RegularFigure* r = dynamic_cast<RegularFigure*>(f)) {
        cout << " regular " << r->size();
    }
    if (Rectangle* r = dynamic_cast<Rectangle*>(f)) {
        cout << " rectangle " << r->diag();
    }
    if (Circle* c = dynamic_cast<Circle*>(f)) {
        cout << " circle " << c->diam() << " " << c->rad();
    }
    cout << endl;
}

int main () {
    Figure* f;
    f = make_rectangle(3, 4); print(f);
    f = make_square(5); print(f);
    f = make_ellipse(3, 4); print(f);
    f = make_circle(5); print(f);
    f->move(2, 3); f->scale(2); print(f);
}
```