ILV Datenstrukturen und Algorithmen

03: Polymorphismus, RTTI, OOP-Prinzipien

Stefan Huber

FH Salzburg, Studiengang MMT / 2012

Kapitel

Polymorphismus

Polymorphismus: Motivation

```
1  class GeometryObject
2  {
3      Vector center;
4    public:
5      GeometryObject(const Vector& diff);
6      double area();
7  };
8  class Circle : public GeometryObject
9  {
10   public:
11     Circle(const Vector& center, double radius);
12     double area();
13 };
14 class Square : public GeometryObject
15 {
16   public:
17     Square(const Vector& center, double width);
18     double area();
19 };
```

Wie implementiert man die folgende Funktion?

```
1  double sumOfAreas(GeometryObject* objArray[], unsigned size)
```

Polymorphismus: Motivation

- Wünschen würde man sich die Implementierung unten.
- Idee: `GeometryObject` bietet die Methode `area()` und es wird `Circle::area()` oder `Square::area()` aufgerufen, je nachdem von welchem Typ `objArray[i]` „tatsächlich“ ist.
- Wie sagt man dem Compiler, dass bei `objArray[i]->area()` in Wirklichkeit `area()` von `Circle` bzw. `Square` aufgerufen werden soll?

```
1  double sumOfAreas(GeometryObject* objArray[], unsigned size)
2  {
3    double sum=0.0;
4    for(unsigned i=0; i<size; ++i)
5      sum += objArray[i]->area();
6    return sum;
7  }
8  void test()
9  {
10   Square obj1(Vector(1,0), 2);
11   Circle obj2(Vector(1,-2), 3);
12   GeometryObject* array[] = {&obj1, &obj2};
13   cout << "Total area: " << sumOfAreas(array, 2) << endl;
14 }
```

Virtuelle Funktionen: Syntax & Semantik

```
1  class A
2  {
3    public:
4      /* Can be "overridden" by a derived class */
5      virtual void func();
6  };
7  class B : public A
8  {
9    public:
10     void func();
11 };
12 void test()
13 {
14   B b;
15   A& a = b;
16   a.func();        // Calls B::func()
17
18   A* pa = &b;
19   pa->func();      // Calls B::func()
20
21   A a2 = b;
22   a2.func();       // Calls A::func()
23 }
```

Virtuelle Funktionen: Syntax & Semantik

```
1  class A
2  {
3    public:
4      /* Can be "overridden" by a derived class */
5      virtual void func();
6  };
7  class B : public A
8  {
9    public:
10     void func();
11 };
```

- Gleiche Signatur, gleicher Rückgabetyp.
- Man sagt: `B::func` „überschreibt (overrides)“ `A::func`.
- `B` muss `func` nicht überschreiben. Es kann etwa erst eine Unterklasse `C` von `B` `func` überschreiben.
- Man kann bei `B::func` auch `virtual` dazu schreiben (empfehlenswert!).
- Polymorphismus: „Vielgestaltigkeit“, `func` kann viele Gestalten haben. Man nennt `A` eine „polymorphe“ Klasse.

Aufruf von virtuellen Basisklassenfunktionen

```
1  class Article {
2    public:
3        string name;
4        int number;
5        virtual void edit() {
6            cin >> name;
7            cin >> number;
8        }
9  };
10 class Book : public Article {
11     public:
12         string author;
13         virtual void edit() {
14                 Article::edit();  // Enter prop. of article
15                 cin >> author;
16         }
17 };
18 void editAll(Article* articles[], unsigned size) {
19   for(unsigned i=0; i<size; i++)
20     articles[i]->edit();
21
22   // Explicitly call Article::edit(), no polymorphism
23   articles[0]->Article::edit();
24 }
```

Abstrakte Klassen, pure virtual functions

- Manche Klassen dienen nur als Basisklassen für andere Klassen:
  - Beispiel: `GeometryObject`
  - Wozu ein Objekt vom Typ `GeometryObject` erzeugen?
  - Was soll `GeometryObject::area()` liefern?
- C++ kennt hierfür sogenannte *pure virtual functions*.
- Damit sagt man dem Compiler:
  - Ich implementiere diese virtuelle member function der Klasse `A` nicht.
  - Dafür dient `A` lediglich als Basisklasse.
  - Und damit kann `A` nicht instantiiert werden.

Abstrakte Klassen, pure virtual functions

```
1  class A
2  {
3    public:
4      virtual void func() = 0;   // Not implemented
5  };
6  class B : public A
7  {
8    public:
9      void func();
10 };
```

- `A::func` wird nicht definiert, nur deklariert.
- Man nennt `func` eine „pure virtual function“.
- Eine „abstrakte Klasse“ hat eine oder mehrere pure virtual functions.
  - Abstrakte Klassen, z.B. `A`, können nicht instantiiert werden.
- Definiert `B` die Funktion `func` auch nicht, so ist `B` ebenfalls eine abstrakte Klasse — und kann ebenfalls nicht instantiiert werden!

Code demo

- `GeometryObject, Square, Circle, area(), sumOfAreas()`

Dynamischer Typ vs. statischer Typ

```
1  class A
2  {
3    public:
4      virtual void func();
5  };
6  class B : public A
7  {
8    public:
9      void func();
10 };
11
12 void test(A& a)
13 {
14   a.func();
15 }
```

- Welches `func` wird hier aufgerufen?
  - Hängt vom „wirklichen“ Typ jenes Objekts ab, das „hinter“ `a` steht.
  - Steht erst zur Laufzeit fest, also dynamisch.

Dynamischer Typ vs. statischer Typ

```
1  class Animal
2  {
3    ...
4  };
5  class Human : public Animal
6  {
7    ...
8  };
9  class Student : public Human
10 {
11   ...
12 };
13 void test()
14 {
15   //Static type 'Animal', dynamic type 'Student'
16   Animal* p = new Student();
17
18   Student h;
19   //Static type 'Animal', dynamic type 'Student'
20   Animal& a = h;
21 }
```

Dynamischer Typ vs. statischer Typ

```
1  //Static type 'Animal', dynamic type 'Student'
2  Animal* p = new Student();
```

- Man muss gedanklich unterscheiden:
  - Das Objekt von einem gewissen (dynamischen) Typ.
  - Die Variable (Pointer, Referenz) von einem gewissen (statischen) Typ, über welche auf das Objekt zugegriffen wird.
- Die Variable ist lediglich ein Bezeichner, ein Konzept einer Programmiersprache, um auf Objekte zuzugreifen.
- Bei virtuellen Funktionen wird die Implementierung des dynamischen Typs aufgerufen.
- Der dynamische Typ eines Objekts wird bei der Instantiierung festgelegt.
- Durch Casting ändert man den statischen Typ, nicht den dynamischen Typ.
- Der Compiler kann type checking nur bezogen auf den statischen Typ machen.
  - Über `p` kann man nur auf Funktionen und Attribute zugreifen, die von `Animal` definiert wurden.
  - Lediglich auf dem Umweg über virtuelle Funktionen kann man Funktionen zum dynamischen Typ aufrufen.

Code sample

- Exceptions, getMessage()

Konstruktor

- Polymorphismus wirkt noch nicht im Konstruktor.
- Keine pure virtual function im Konstruktor aufrufen!
- Braucht man Polymorphismus um ein Objekt zu initialisieren, dann kann man eine member Funktion, etwa `initialize`, anlegen, die explizit nach dem Konstruktor aufgerufen werden muss.

Konstruktor

```
1  class A
2  {
3    public:
4      A();
5      void initialize();
6      virtual void func();
7  };
8
9  A::A()
10 {
11   //No polymorphismus here
12 }
13 void A::initialize()
14 {
15   // Put relevant code from constructor to here...
16   func();
17 }
18
19 void test()
20 {
21   //Construct object and call 'initialize'
22   A a;
23   a.initialize();
24 }
```

Destruktor

- Wird in einer Unterklasse `B` von `A` Speicher reserviert und `delete` über einen Pointer `A*` aufgerufen, so wird der Speicher nicht freigegeben!

```
1  class A
2  {
3    public:
4      ~A();
5  };
6  class B : public A
7  {
8      int* memory;
9    public:
10     ~B();
11 };
12 void test()
13 {
14   A* obj = new B();
15   delete obj; //Does not call B::~B() !!!
16 }
```

Destruktor

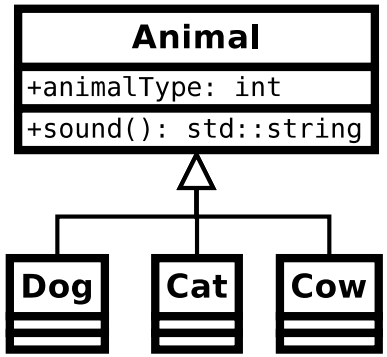
- Wird in einer Unterklasse `B` von `A` Speicher reserviert und `delete` über einen Pointer `A*` aufgerufen, so wird der Speicher nicht freigegeben!
- Lösung: Destruktor virtuell deklarieren.
  - Polymorphe Klassen sollten i.A. virtuelle Destruktoren definieren!

```
1  class A
2  {
3    public:
4      virtual ~A(); // Virtual destructor
5  };
6  class B : public A
7  {
8      int* memory;
9    public:
10     ~B();
11 };
12 void test()
13 {
14   A* obj = new B();
15   delete obj; //Does call B::~B()
16 }
```

Kapitel

RTTI

Zugriff auf Unterklassen



- Wir haben ein Array von Animals: `Animal* array`
- Aufgabe 1: Man gebe alle Tierlaute aus.
- Aufgabe 2: Man zähle die Hunde.

- **Type fields:** schlechter Stil!
  Der „dynamische Typ“ wird in einer Variable `type` der Basisklasse vermerkt und im Konstruktor der Unterklassen gesetzt.

  Katastrophaler Stil: Alle Tierlaute werden in `Animal::sound()` implementiert. Die Basisklasse ist massiv abhängig von den Unterklassen — man kann sich hier Vererbung gleich sparen.
- **Virtual functions:** schöner Stil
  - `Animal::sound()` ist eine pure virtual function.
  - Für Aufgabe 2 wäre es aber nicht angebracht extra eine virtuelle Funktion `isDog()` einzuführen. (Wie würde man das machen, ohne für alle Animals diese Funktion implementieren zu müssen?)
- **dynamic cast:** manchmal sinnvoll, z.B. für Aufgabe 2

Dynamic cast

```
1 void test(Animal* pet)
2 {
3   Dog* d = dynamic_cast<Dog*> (pet);
4   if( d == 0 )
5     cout << "pet is no dog!" << endl;
6   else
7     cout << "pet is really dog!" << endl;
8 }
```

- `dynamic_cast<Type*> (p)` versucht den Pointer `p` auf den angegebenen Typ zu casten.
  - Das geht, falls `Type` dem dynamischen Typ von `p` entspricht, oder einer Basisklasse davon.
  - Ansonsten bekommen wir den Null-Pointer.
- Aufgabe 2: teste für jedes Animal, ob `dynamic_cast<Dog*>` ungleich `0` liefert.
  - Wenn `Dog` weitere Unterklassen hat (`GermanShepherd`, ...), dann werden auch Instanzen davon mitgezählt.
  - `test(new GermanShepherd())` liefert `pet is really a dog!`.

RTTI

- Dynamic cast gehört zum sogenannten RTTI System (runtime type information) von C++
- C++ kann das nur für polymorphe Typen.
  - Dynamic cast funktioniert nur für polymorphe Typen!
- Neben `dynamic_cast` bietet C++ RTTI auch `typeid`.
  - Zum Vergleichen von Typen.
  - Zum Ausgeben des (internen) Namens eines Typs.

```
1 #include <typeinfo>
2 Animal* p = new Dog();
3 Dog d;
4 assert( typeid(*p) == typeid(d) );
5 cout << "type of *p: " << typeid(*p).name() << endl;
6     // type of *p: 3Dog  -- or something similar
```

Type casting: revisited

Das klassische type casting im C-style kennen wir:

- `Animal *a = (Animal*) pointer;`
- Veraltet und eher fehleranfällig.

C++ definiert eigene casting Operatoren:

- `Dog* d = dynamic_cast<Dog*>(pet);`
  - Inspiziert zur Laufzeit das Objekt, ob casting legitim ist.
- `Dog* a = static_cast<Dog*>(pet);`
  - Prüft nicht zur Laufzeit (etwas effizienter), trotzdem ist casting zu Unterklassen ist möglich — Programmierer ist verantwortlich für Korrektheit.
- `Dog* d = reinterpret_cast<Apple*>(apple);`
  - Damit lassen sich beliebige Typen ineinander konvertieren — kein type checking, gefährlich!
- `Dog& d = const_cast<Dog&>(constdog);`
  - Entfernt von `const Dog& constdog` den const qualifier.
  - Sollte in einem guten Programm-Design nicht notwendig sein!

Kapitel

Objektorientierte Prinzipien

Grundprinzipien der OOP

Objekt-orientierte Programmierung fußt auf diesen Grundprinzipien:

- **Abstraktion**
  - Man modelliert Objekte durch Angabe der Attribute und Funktionalitäten und fasst diese Gesamtheit als Klasse auf.
- **Datenkapselung**
  - Implementierungsdetails werden versteckt (`private`).
- **Vererbung**
  - Klassen können in Vererbungsbeziehungen untereinander stehen. Abgeleitete Klassen übernehmen die Eigenschaften der Basisklassen.
- **Polymorphismus**
  - Ermöglicht den Zugriff auf Funktionalität einer speziellen Klasse selbst wenn der Bezeichner (Variable) von allgemeinerem Typ ist.

Gutes Klassendesign

- **Modularisierung:**
  Dinge, die nicht zusammengehören trennen. Dinge, die zusammengehören, in eigene Module kappseln.
- **Abhängigkeiten reduzieren:**
  Die Abhängigkeiten zwischen Modulen reduzieren. Eine Veränderung interner Details eines Moduls soll nicht Änderungen in anderen Modulen nach sich ziehen.
- **Wiederverwendbarkeit und Erweiterbarkeit:**
  - Ähnlichen Code nicht immer und immer wieder neu schreiben. Wiederverwendbaren Code entwickeln.
  - Haben zwei Klassen viele Ähnlichkeiten, dann sollen ggf. eine gemeinsame Basisklasse eingeführt werden, die wiederverwendet werden kann.
  - Erweiterung der Klassenhierarchie durch Vererbung soll möglich sein. (Keine Abhängigkeiten der Basisklasse von Unterklassen!)

Interfaces

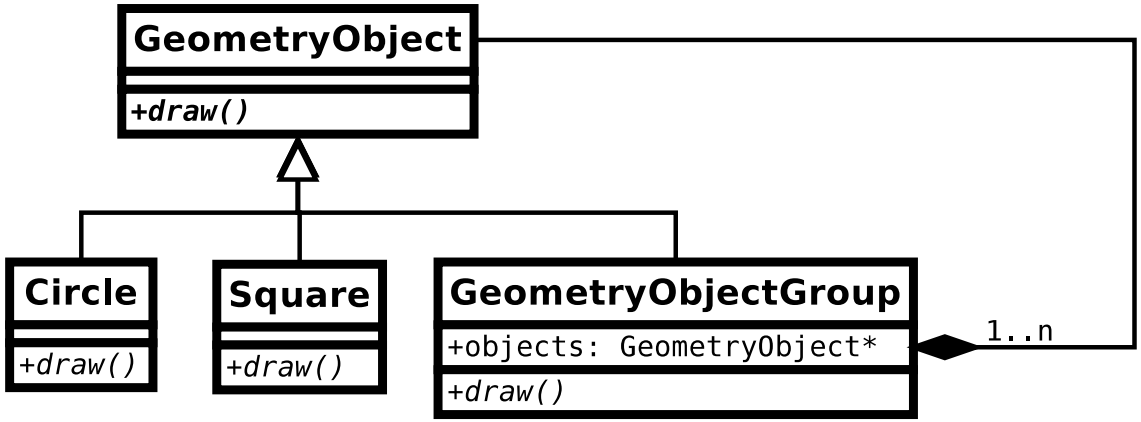
- Ein Interface ist eine Klasse, die ausschließlich pure virtual functions definiert (und keine member variablen und keine weiteren member functions).
- Mit Interfaces gibt man an, was eine Klasse, welche dieses Interface implementiert, können muss.
- Mit Interfaces entkoppelt man Module, denn die Funktion `print` weiß lediglich, dass `obj` ein `toString` anbietet, und mehr braucht sie auch nicht zu wissen.

```
1  /** Any class derived from Stringable can be
2   *  converted to a string. */
3  class Stringable
4  {
5    public:
6      virtual string toString() const = 0;
7  };
8
9  /** Print any stringable object */
10 void print(const Stringable& obj)
11 {
12   cout << obj.toString();
13 }
```

Design patterns

- Gute Klassendesigns weisen oft wiederkehrende Muster und Bauweisen in Klassenhierarchien auf.
- Gamma et. al haben 23 sogenannte design patterns vorgestellt.
- Ein design pattern ist ein allgemeiner, wiederkehrender Bauklotz in Klassenhierarchien.

Composite Pattern



- Wir wollen eine Gruppe von `GeometryObject`-Instanzen, die sich wie ein `GeometryObject` verhält und z.B. gezeichnet werden kann.
- Idee: `GeometryObjectGroup` selbst **ist ein** `GeometryObject` und **hat ein** Array von `GeometryObject`-Objekten.
- `GeometryObject::draw()` ist polymorph.
- `GeometryObjectGroup` implementiert ebenfalls `draw` und zeichnet alle Objekte, die es **enthält**.
- `GeometryObjectGroup` braucht nicht zu wissen, ob es `Circle`- oder `Square`-Objekte enthält — sie müssen sich nur wie ein `GeometryObject` verhalten.
- Beachte: Eine `GeometryObjectGroup` kann selbst weitere `GeometryObjectGroup` enthalten!