Objekt Orientiertes Programmieren

Eine Einführung in einige grundlegende Ideen

Andreas Meyer
II.Inst.Exp.Phys.
H1 Experiment, DESY
Andreas.Meyer@desy.de

Objekte, Interfaces, Klassen, Vererbung, Polymorphismus

anhand einiger Beispiele (C++ und RooT)

http://www.desy.de/~ameyer/c++/oo/C++kurs2.pdf

Quellen

Material zu dieser Vorlesung:
`http://www.desy.de/~ameyer/c++/oo`

Basics:
`http://www-pat.fnal.gov/root/CPlusPlus/index.html`
`http://www.zib.de/Visual/people/mueller/Course/Tutorial/`
`http://www.slac.stanford.edu/BFROOT/www/Computing/Programming/`

Talks:
`http://www.slac.stanford.edu/BFROOT/www/Computing/Programming/Programming.html`
`http://www.slac.stanford.edu/BFROOT/www/Computing/Environment/Training/CompTraining.html`

Websites/Linklists:
`http://www.research.att.com/~bs/C++.html`

Bücher:
S.Oualline, 'Practical C++ Programming' O'Reilly, 1997
`ftp://ftp.oreilly.de/pub/ora/examples/nutshell/practcpp/`

Kursmaterial:
'Principles of Object Oriented Design with UML', Object Mentor, Inc.

Übersicht

–Einleitung
–Software Management
–Black Boxes
–Objekte: Interfaces, Encapsulation
–Klassen in C++
–Beispiel: komplexe Zahlen `comp.h, comp.cpp, testcomp.cpp`
–Beispiel: StoppUhr
–Vererbung
–Beispiel: Instrumente `Instrument.h, testInstr.cpp`
–Polymorphismus
–Dynamic Binding
–Starke Interfaces: (Pure Virtual Functions)
–Beispiel: Rekonstruktion von J/Psi aus Muonen (4–Vektoren)
`vec.root, jpsi_macro.C`
–Zusammenfassung

Ist OO die Lösung aller Probleme?

Nein!
OO Software ist immer noch Software ...

Dieser Vortrag ist viel zu kurz, um ein Kurs zu sein
Ich bin Physiker und kein Softwareexperte

Die Beispiele werden am 19.1.1 (nächste Woche)
am Computer diskutiert
Bitte vorher genau ansehen und Fragen überlegen

Entwicklung der Programmierstile

schnell Op−code, Assembler
  Statement,
  Modules,
  Data Structures,
  Abstract data types: Algorithms and data together
langsam Objektorientierung

Paradigmenwechsel:
- möglich durch leistungsfähige Rechner.
- notwendig durch Größe der Programme

Einige Bemerkungen vorweg

OO ist nicht neu (~30 Jahre alt)
OO ist anders als prozedurales Programmieren
OO ist schwierig zu lernen
OO ist schwierig in der Anwendung
OO ist nicht vollkommen, aber
gutes OO kann sich lohnen und wird benutzt in großen Projekten (>10000 Zeilen)

Software Management

(Große) Software Projekte sind immer unübersichtlich

Beispiel HEP−Experimente: 1−2 Mio Zeilen Rekonstruktionscode

Herkömmliches Software Modell

Wasserfallmodell:

Requirements −>
Design −>
Coding −>
Testing −>
Maintenance

Voraussetzung:

Gutes Verständnis der Probleme im Vorhinein
Spätere Änderungen sehr schwierig

Wiederverwendung der Software:

Meist vollkommen unmöglich
Programmierer müssen oft das gesamte Projekt überblicken
Maintenance stellt sich als sehr teuer heraus

Beispiel:

Änderung der Datenstruktur −> Änderung aller Teile, die die Daten benutzen.

Ausweg: Black Box Approach

Methodologie, die die Möglichkeit von
Korrektheit, Robustheit, Flexibilität, Erweiterbarkeit, Wiederverwendbarkeit
nicht ausschließt

Black Boxes:
Komplex von innen, Einfach von außen: –> Interface

Ausgangspunkt:
Menschen können sich immer nur 7+/–2 Dinge gleichzeitig merken.

Objekt Orientiertes Programmieren: Definition der Black Boxes
Abstraktionsleistung!
Welche Eigenschaften sind relevant für das Problem?

Was ist ein Objekt?

Ein Objekt ist wie eine Black Box: Es
- hat ein Interface (sendet und empfängt Botschaften)
- hat einen Zustand (enthält Daten)
- hat ein Verhalten (kennt Methoden, Algorithmen, Funktionen)

Traditionell wurden Daten (c: structs) und Methoden (c: functions) getrennt gehalten. In OO gehören Daten und Methoden in einem Objekt untrennbar zusammen. Methoden eines Objekts können Daten anderer Objekte nicht ’sehen’ (Encapsulation).

- hat eine Identität (Typ und Name)
- wird entworfen, gebaut, definiert, programmiert von
  <u>einem</u> Designer, Programmierer, Schöpfer ...
- wird benutzt von <u>vielen</u> Usern

Was ist ein Objekt also?

Objekte sind
sowohl die physischen als auch die konzeptionellen Dinge, die uns umgeben / die relevant für das Problem sind.

Ein Modell einer Firma: Angestellte, Gebäude, Abteilungen, Dokumente, Gehälter, ...

Autobau: Reifen, Türen, Motoren, Verbrauch, ...

Simulation chemischer Prozesse: Atomen, Molekülen, Volumina, Temperaturen, ...

Elementarteilchenphysik:
Detektorsignale, Hits, Spuren, Cluster, (identifizierte) Teilchen, 4−Vektoren, Vertizes, Ereignisse, ...

Objekteigenschaften an Beispielen

Ein Objekt hat einen Zustand

Beispiel: Tank: (voll/leer)

Ein Objekt hat ein Verhalten

passiv (statisch): Änderung durch messages von außen

Beispiel: Briefkasten (füllen / leeren)

aktiv: Änderung von selbst

Beispiel: Uhr, Mensch

Ein Objekt hat ein Interface

Ein Objekt hat eine Identität

Beispiel: Mein Computer ist ein Linux−PC und heisst lido.desy.de

(Mehrere Objekte gleichen Typs (Verhalten) können unter verschiedenen Namen und mit gleichem oder unterschiedlichem Zustand existieren.)

Wichtigste OO–Regel:

»Never need to peek inside the box«

Objekte kommunizieren ausschließlich über Messages.
Form und Inhalt der Messages ist definiert durch Interfaces.

Man soll und muß nichts wissen über die Funktionsweise der Objekte, um sie zu benutzen (Encapsulation/Information Hiding)

–> Die Person, die das Objekt betreut, kann Details im Objekt ändern, ohne daß dadurch anderer Code 'kaputt geht'

–> Die Personen, die das Objekt benutzen, sollten sich nicht davon abhängig machen, wie das Objekt funktioniert.

Gute Software

Gutes Design
Lokale Änderungen ohne Auswirkungen auf andere Teile
Die Erfahrung lehrt, daß Software sich ändert.
Wenn sich Software ändert, sind die Autoren oft nicht mehr erreichbar.

Unabhängige Objekte
mit relevantem, korrekten Verhalten
mit guten, robusten, erweiterbaren Interfaces

OO ist eine Disziplin, die Abhängigkeiten so verwaltet, daß durch Änderungen kein 'Software Rot' entstehen muß

Erweiterungen sind möglich ohne Änderungen an altem Code

OO Design Model

Modellierung, Gebrauch und Verbesserung
Zyklischer Prozeß

Evaluation

Requirements Gebrauch

Design Tests

Implementierung

Klassen

Eine Klasse ist eine Beschreibung eines Objekts
einschließlich dem Mechanismus für die Erzeugung des Objekts.

Eine Klasse definiert
- die Daten
- die Methoden
- die Interfaces
  - (input) parameter
  - public
  - inheritance

Ein Objekt ist eine im Speicher erzeugte Klasse.
Ein Objekt ist eine Instanz der Klasse.

Es können viele Objekte der gleichen Klasse (des gleichen Typs)
unter unterschiedlichen Namen erzeugt werden.

Klassen in C++

Syntax am Beispiel einer Klasse für komplexe Zahlen

`comp.h` Deklaration (auch **`.hh`**)

`comp.cpp` Implementierung (auch **`.C`** **`.cc`**)

Die (selbstgeschriebene) Klasse **`comp`** für komplexe Zahlen enthält:
– zwei Zahlen (Re,Im) als Datamembers
– Methoden zum Zugriff auf Re, Im, Betrag und Winkel (rad., deg.) (Accessorfunctions)
– Methoden zur Ausgabe (Print)
– Methoden zur Addition (+) und Multiplikation (*)

comp.h

```
#ifndef __comp
#define __comp

class comp
{
protected:
  double re;          // data members
  double im;

public:
  // memberfunctions
  comp(double r=0, double i=0){re=r;im=i;} //  constructor
  double real() { return re; } ;          // Accessorfunctions
  double imag() { return im; } ;
  double absValue();
  double angle();
  void print();                           // Print Method
  friend comp operator + ( comp, comp ) ;  // overloaded operators
  friend comp operator * ( comp, comp ) ;
};

#endif
```

comp.cpp

```
// implementation of functions prototyped in comp.h

#include "comp.h"
#include <math.h>
#include <iostream.h>

// const double pi = 3.14159265358979323846;

double comp::absValue() {
    return sqrt(pow(imag(),2.)+pow(real(),2.));};

double comp::angle() {
   return atan(imag()/real());}

void comp::print() {
  cout << "Real: " << real() << "   Imaginaer: " << imag() << endl;
  cout << "Winkel: " << angle() << " Betrag: " << absValue() << endl;
}

comp operator + ( comp a1, comp a2) {
return comp(a1.real() + a2.real(), a1.imag() + a2.imag());}

comp operator * ( comp a1, comp a2) {
return comp(a1.real() * a2.real() - a1.imag() * a2.imag(),
                          a1.real() * a2.imag() + a1.imag() * a2.real());}
```

testcomp.cpp

```
#include "comp.h"

main(){
comp a(1.,0.);
comp b(0.,1.);
comp c;

c=a+b;
c.print();
c=a*b;
c.print();

// uebrigens geht auch:
(a*b).print();
}
```

Was geschieht hier also?

- Die komplexen Zahlen a und b werden initialisiert.
- Addition und Multiplikation werden ausgeführt
- Print–Methode zeigt das Ergebnis (selbst eine komplexe Zahl) an.

(Übung: Implementiere eine Methode zur Ausgabe des Winkels in Grad)

Inheritance, Vererbung

Ableitung von der abstrakteren zur konkreten Klasse
(»is a« Relation)

Beispiele:
•Stoppuhren sind Uhren, genau wie Standuhren, Parkuhren, Armbanduhren (nächstes Beispiel)
•Trompeten sind (Blas)instrumente, Klaviere sind auch Instrumente (uebernächstes Beispiel mit Beispielcode)
•4−Vektoren enthalten 3−Vektoren (siehe z.B. root: TLorentzVector−Klasse `http://root.cern.ch/root/html/TLorentzVector.html`)
•Elektronen sind identifizierte, geladene Teilchen genauer: Leptonen (hier kein Beispiel, kommt zu H1 und seht selbst!)

Beispiel: Objekt Stoppuhr

Verhalten/Methoden:
start, stop, reset, timer, showstate (stopped or running), showtime
Zustand: Stopped or running, elapsed time

Wie funktioniert die Stoppuhr?
Spielt keine Rolle, so lange sie nur funktioniert
Encapsulation / Information Hiding:
public data accessible to clients, private data not accessible

Was hat eine Stoppuhr gemein mit anderen Uhren?
Inheritance

Ein Inheritance Tree für Uhren

Uhr

Uhr, die stoppt — Uhr, die immer läuft

Stoppuhr, Parkuhr — Armbanduhr, Wanduhr

Abgeleitete Klasse hat alle Eigenschaften der Basisklasse

- –Daten
- –Methoden
- –(Parameter–) Interfaces

Abgeleitete Klasse kann zusätzliche Dinge hinzufügen

(java: **class StoppUhr extends UhrDieStoppt{**)

C++: **class StoppUhr : <public> UhrDieStoppt {**

Beispiel Instrumente: **TestInstr.cpp**

```
#include "Instruments.h"

void spielen(Instrument& i) {
  i.play();
}

int main() {
  Blaeser blaeser;
  Piano piano;
  Trumpet trumpet;

  cout << " ------ Blaeser --------" << endl;
  blaeser.play();        // Direkter Funktionsaufruf
  spielen(blaeser);      // Upcasting to base class
  cout << " ------ Klavier --------" << endl;
  piano.stimmen();       // Direkter Funktionsaufruf
                         // (zusaetzliche Funktion)
  piano.play();          // Direkter Funktionsaufruf
  spielen(piano);        // Upcasting to base class
  cout << " ------ Trompete --------" << endl;
  trumpet.play();        // Direkter Funktionsaufruf
  spielen(trumpet);      // Upcasting to base class
}
```

Instruments.h

```
#include <iostream>

class Instrument {
public:
  void play() {
    cout << "Instrument::play"  << endl;}
};
// ------------------------------------------------------------------
// Piano erbt von Instrument und implementiert zusätzliche Eigenschaften
class Piano : public Instrument {
public:
void stimmen() {
    cout << "Piano::stimmen"  << endl;
  }
void play() {
    cout << "Piano::play"  << endl;
  }
};
// ------------------------------------------------------------------
// Blaeser erben von Instrument (Implementation spaeter)
class Blaeser : public Instrument {// public members of Instrument are also
                                   // public members of Flute
};
// ------------------------------------------------------------------
class Trumpet : public Blaeser {
private:
// implementation of trumpet simulation to come later
};
```

Polymorphismus

Einer der wichtigsten Aspekte von OO

Instrumentenbeispiel:
– Methode play() ist implementiert in der Klasse Instrumente
– Piano erbt von Instrument,
aber es implementiert die Klasse selbst anders

"Polymorphismus":
- Zwei Objekte mit dem gleichen Interface können die gleiche Message erhalten.
- Der Sender kann den Unterschied zwischen den Objekten nicht erkennen.
- Die gleiche Message (Operation) kann unterschiedliches Verhalten in verschiedenen Klassen, Typen, Objekten erzeugen

Zweck: Fallunterscheidung ohne If–Statements

Dynamic Binding

`virtual void play();`

Basis Klasse (Instrument) nimmt Verhalten der abgeleiteten Klasse an.

Run–time Entscheidung
Die Funktion **`spielen`** braucht nur für die abstrakte Klasse Instrument definiert zu werden (nicht aber extra für alle konkreten Klassen, Klavier, Trompete ...)

Keine Fallunterscheidungen im Code notwendig –> Erweiterbarkeit

Keyword **`virtual`** definiert also nur das Interface und macht die Übernahme der Funktionalität optional. Die abgeleitete Klasse kann ihr eigenes Verhalten implementieren, muss es aber nicht.

Pure virtual functions

`virtual void play() = 0;`   Reines Interface

Basisklasse (Instrument) erzwingt Implementierung der Methode **`play()`** in allen abgeleiteten Klassen.

Keine Implementierung der Funktion in der Basisklasse selbst möglich.

Nur das Interface ist vordefiniert in Basisklasse,
abgeleitete Klassen müssen sich daran halten.

Erweiterungen müssen vorhandene Interfaces verwenden

–> OO ist eine Disziplin, die Erweiterungen ohne Änderungen ermöglichen kann
–> Gutes Design ist Voraussetzung,
Zyklische Softwareentwicklung ist Mittel zu diesem Zweck

Vorschläge zur Übung

Inheritance & Polymorphismus

–Kompiliere **`testInstr.cpp`**

–Modifiziere die Funktion play() der Basisklasse Instrument

1) **`void play(){...`** nach **`virtual void play(){...`**
2) **`virtual void play(){...`** nach **`virtual void play() = 0 ;`**
   Beachte: Pure virtual functions erlauben keine Implementierung in der Base–class und erfordern die Implementierung in der abgeleiteten Klasse
3) Implementiere **`void play(){...`** in der Klasse Blaeser
4) Implementiere **`void play(){...`** in der Klasse Trompete

–Nach jedem Durchgang Kompilieren und Ausführen von testInstr

–Beobachte das Verhalten von **`play()`** beim Upcasting nach der Baseclass Instrument im Vergleich zum direkten Funktionsaufruf.

Beispiel: J/Psi mit 4–Vektoren

Root–File mit je zwei 4–Vektoren pro Event
4–Vektorklasse TLorentzvector verwendet Lorentzmetrik für Operationen
Berechnung der Invarianten Masse und Füllen des Histogramms ist sehr einfach (Korrektheit):

```
// ...
TH1F *histo   = new TH1F("total","Jpsi",80,2.,6.);
for ( int i = 0 ; i<nentries ; i++ ) {
    fourvecs->GetEntry(i);
    TLorentzVector sum=*v1+*v2;
    histo->Fill( sum.M() );
}
gStyle->SetOptStat(111111);
histo->Draw();
}
```

siehe: **http://www.desy.de/~ameyer/c++/oo/jpsi_macro.C**

Vorschläge zur Übung mit Root

–Lade `vec.root` und `jpsi_macro.C` von der Webseite
–Starte `root`
–Führe das Macro aus
–Fülle zusätzliche Histogramme mit
Energie, Transversalimpuls, Polarwinkel
der beiden Spuren und des J/Psi
(Hinweis: Benutze dazu die Accessorfunctions
`E(),Pt(),Theta()`;
Beschreibung der Klasse:
`http://root.cern.ch/root/html/TLorentzVector.html`)
Wieviel Ereignisse gibt es mit einem Pt > 1 GeV/c?

Zusammenfassung

OO

Objekte haben:
Zustand Daten
Verhalten Funktionen
Interface Messages definierter Form und Inhalt
Identität Typ und Name

Daten und Methoden zusammen:	Übersichtlich, Unabhängig
Encapsulation/Data Hiding	Korrektheit, Unabhängigkeit
Abstract Data Types	Objekt–Modellierbarkeit
Operator Overloading	Einfachheit
Inheritance	Konkretes erbt vom Abstrakten
Polymorphismus	Veränderbarkeit, Erweiterbarkeit
Interfaces	

Bemerkungen