

Objekt-Orientierte Programmiersprachen

Martin Gasbichler, Holger Gast

27. Oktober 2005

Plan für die erste Semesterhälfte

- ✓ Einführung
- ✓ OCaml Einführung
- Objekt-orientierte Programmierung (in Java)
- Konzepte von Programmiersprachen (in OCaml)
- Eine objekt-orientierte Sprache im Detail (in OCaml)
- Klassenbasierte Sprachen

Warum OOP in dieser Vorlesung?

- Programmiersprachen bieten Ausdrucksmittel für Programmierer
- ⇒ Objekt-orientierte Sprachen unterstützen objekt-orientierte Programmierung
- ⇒ Um die Sprachen zu verstehen, müssen wir OOP kennenlernen
- Plan
 - 27.10.: Sprachkonstrukte Java und grundlegende OOP Techniken
 - Übung: Einfache Erweiterung eines graphischen Editors
 - 3.11.: Fortgeschrittene Techniken, Design Patterns
 - Übung: Refactoring im graphischen Editor
- Mit OOP könnte man auch eine komplette Vorlesung bestreiten

Naiver Ansatz zu OO Design

- Naive Feststellung: Ein Java Programm besteht aus Klassen

⇒ Naive Fragen

- Welche Klassen werden benötigt?
- Welche Felder sollen die Objekte haben?
- Welche Methoden müssen verfügbar sein?

- Probleme mit naivem Ansatz

- Es gibt nicht **die** Sammlung von Klassen, nicht **die** Lösung
- Fragen orientiert an der Sprache, nicht am Problem
- Daten zur Laufzeit sind (größtenteils) Objekte, nicht Klassen

⇒ Wir brauchen Konzepte um über OO Programme zu sprechen

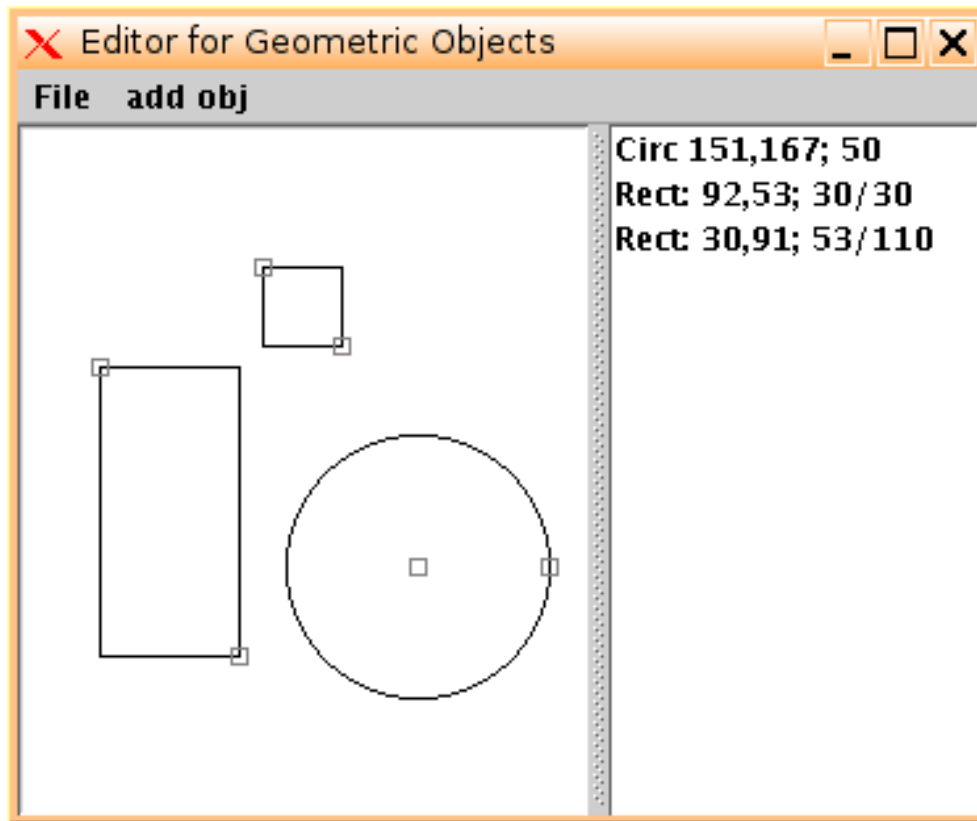
Grundtechniken OO Design

- Client/Server Beziehung
 - Parnas' Geheimnisprinzip
 - Design by Contract
 - Modellierung des Anwendungsgebietes
 - Hierarchien
- ⇒ Überlappungen
- ⇒ Verschiedene Sichtweisen & Fragestellungen
- Nächste Stunde: Responsibility Driven Design, Design Pattern

“Design” beinhaltet dabei nicht . . .

- Allgemeine Strategiefragen
 - Wie allgemeingültig soll meine Lösung sein?
 - Welche Teile der Lösung schreibe ich zuerst, welche später?
 - Systemdesign
 - Anforderungen an das System
 - Benutzerschnittstelle
 - Sprachfragen
 - Prozedural oder objekt-orientiert?
 - Low-level C oder high-level OCaml oder Prolog?
- ⇒ Im wesentlichen also “Design” = “Programmstruktur”

Durchgehendes Beispiel: Der Editor



- Interaktive Bedienung
- Kreise und Rechtecke
 - erzeugen
 - löschen
 - ändern
- Anzeige von Koordinaten
- Zentral: Direkte Reaktion

Struktur des Editors

- Daten (Modell)
 - Einzelne Zeichnungsobjekte
 - Die gesamte Zeichnung
- Graphische Anzeige
 - Anzeige für einzelne Zeichnungsobjekte
 - Koordinator: Editor für eine Zeichnung
 - Standard-Komponenten (Menüs, Dialoge, etc.)
- Verbindung
 - Anzeige fragt Daten ab und gibt entsprechende Linien/Kreise aus
 - Benutzereingaben ändern die entsprechenden Daten
 - Jede Änderung an den Daten muss angezeigt werden

Klassen des Editors

- Datenebene
 - Shape, Circle, Rectangle
 - Drawing Eine Zeichnung
 - Anzeigeebene
 - ShapeEdit Grundfunktionalität für Dragging
 - CircleEdit, RectangleEdit Anzeige eines Circle, Rectangle
 - EditorPane Anzeige und Editor für ein Drawing
 - TextView Koordinatenanzeige
 - AppWindow Hauptfenster mit Menüs und EditorPane, TextView
 - Main Hauptprogramm öffnet ein AppWindow
- ⇒ Anzeige erfordert (wie üblich) den Großteil des Aufwandes

Client/Server Beziehung

- Metapher: Dienstleistung
 - Objekt a ruft Methode m von Objekt b auf.
 - b erledigt also in m eine Aufgabe für a .
 - b ist Anbieter (server) einer Dienstleistung, a ist Kunde (client)
- Client **delegiert** seine Aufgaben an den Server
- Client muß die Interna des Servers nicht kennen
- Wiederverwendung fertiger Funktionalität
- Leicht prozedural gedacht: Das geht auch in Pascal und C

Beispiel: Die Drawing Klasse

- Drawing muß Listen mit Kreisen, Rechtecken verwalten
- Vector Instanz verwaltet die Kreise & Rechtecke

```
class Drawing {  
    private Vector shapes;  
    ...  
    public void add(Shape s) {  
        shapes.add(s);  
        ...  
    }  
    ...  
}
```

Parnas' Geheimnisprinzip

- Metapher: Der Anbieter ist der Experte.
 - Legt nur seine **Schnittstelle** für die Benutzung offen
 - Kümmert sich um die “technischen Details”
- Der Kunde **braucht nicht** zu wissen, wie der Anbieter arbeitet.
 - Reduktion der Komplexität
 - Parallele Programmierung
 - **Separation of concerns** (Dijkstra)
- Der Kunde **darf nicht** wissen, wie der Anbieter den Dienst erbringt.
 - Verhindert Mißbrauch
 - Schützt den Anbieter
 - Sichert Austauschbarkeit des Anbieters

Beispiel: Geheimnisprinzip

- Drawing weiss nicht, wie Vector implementiert ist
- EditorPane verwendet HashMap und Drawing
- ShapeEditFactory wählt zum Objekt passenden Editor aus

```
class EditorPane ... {
    private ShapeEditFactory editFactory;
    private Drawing drawing;
    private HashMap edits;
    private void addEditorFor(Shape shape) {
        edits.put(shape, editFactory.getEditor(shape));
    }
    public void removeByEdit(ShapeEdit edit) {
        drawing.remove(edit.getShape());
    }
}
```

Design by Contract

- Vorgeschlagen von Bertrand Meyer (→Eiffel)
 - Metapher: Kunden und Anbieter schließen Verträge
 - Kein Dienst ist umsonst, der Kunde muß auch etwas “zahlen”
 - Genaue Spezifikation: Welchen Dienst bietet der Anbieter eigentlich an und zu welchen Konditionen?
 - Der Anbieter kann seine Aufgabe nicht in jedem Fall erfüllen
 - Unzulässige Parameter
 - Zu lesende Datei nicht vorhanden
 - Früherer Fehler nicht behoben
 - Der Anbieter kann nur ein spezielles Ergebnis zusichern
- ⇒ Wenn der Kunde seine Pflichten erfüllt,
dann muß der Anbieter die seinen erfüllen

Spezifikation

- Wie schreibt man Verträge auf?
 - Man braucht Zusicherungen & Anforderungen an Objekte
 - Anforderungen an erlaubte Werte von Parametern
 - Zusicherungen für Rückgabewerte
 - Zustand von Feldern / Objekten
 - Zustand von Feldern ändert sich eventuell durch Zuweisungen
 - Brauchen Aussage: “Zu diesem Zeitpunkt gilt sicher”
- ⇒ Zentrale Idee der Hoare Logik
(*An Axiomatic Basis for Computer Programming*, CACM 1969)

Vor- und Nachbedingungen

- Relevante Zeitpunkte: Vor und nach Ausführung einer Anweisung
- Jede Anweisung bekommt eine Vorbedingung und eine Nachbedingung
 - Sie beschreiben den Zustand aller (sichtbaren) Variablen vor bzw. nach Ausführung der Anweisung
 - Erweiterung auf Methoden: Vor- & Nachbedingung des Rumpfes
- Korrektheit eine Anweisung: **Wenn** vor Ausführung der Anweisung die Vorbedingung galt, **dann** muß die Anweisung die Nachbedingung sicherstellen
 - Vorbedingung = Voraussetzung für erfolgreiche Ausführung
 - Nachbedingung = Zugesichertes Ergebnis der Anweisung
- Lies “Bedingung” wie “Wetterbedingung”, nicht als “Voraussetzung”

Beispiel: Vor- und Nachbedingungen

```
int maxOfIntArray(int a[]) {  
    // a.length ≠ 0  
    int res = a[0];  
    int i = 1;  
    int n = a.length;  
  
    // n = a.length ∧ res = max(a0...ai-1) ∧ i ≤ n  
    while (i != n) {  
        if (a[i] > res) { res = a[i]; }  
        i++;  
    }  
  
    // res = max(a0...an-1)  
    return res;  
}
```

Invarianten

- Invariante ist ebenfalls Aussage über sichtbare Variablen
 - Sie gilt **immer wenn** die Programmstelle erreicht wird, an der sie notiert ist
 - Sie gilt in der Regel **nicht** “zu jedem Zeitpunkt”
- Anwendungen
 - Klassisch: Schleifeninvariante
 - Aussage über Objektzustand:
“Immer wenn eine Methode aufgerufen wird, gilt . . .”

Objekt-Invarianten

- Methoden arbeiten auf Feldern
- ⇒ Sie müssen Annahmen über deren Inhalt machen
- Aussage: “Immer wenn eine Methode aufgerufen wird, gilt”
- ⇒ Das ist eine Invariante über die Objekt-Felder
- Häufig: Konsistenzbedingungen
 - Konstruktoren stellen Invarianten durch Initialisierung her

Beispiel: EditorPane Invarianten

```
class EditorPane ... {  
    private Drawing drawing;  
    private HashMap edits;  
    public void paintComponent(Graphics g) {  
        ...  
        Shape shapes[] = drawing.getObjs();  
        ...  
        ((ShapeEdit)edits.get(shapes[i])).paint(g);  
    }  
    ...  
}
```

- Feld drawing ist nicht null
- HashMap edits ist nicht null
- edits enthält für jedes Objekt in drawing einen Editor
- Alternative: Tests in jeder Methode

Beispiel: Gewährleistung von Invarianten in EditorPane

```
public EditorPane(Drawing drawing) {
    // Vorbedingung: drawing != null
    ...
    setDrawing(drawing);
}
public void setDrawing(Drawing drawing) {
    // Vorbedingung: drawing != null
    ...
    this.drawing = drawing;
    edits = new HashMap();
    Shape os[] = drawing.getObjs();
    for (int i=0; i!=os.length; ++i) {
        addEditorFor(os[i]);
    }
}
public void shapeAdd(Drawing d, Shape s) {
    if (d == drawing) {
        addEditorFor(s);
        repaint();
    }
}
```

Zusammenfassung Design by Contract

- Explizite Verträge zwischen Anbieter und Kunde
 - Kunden haben auch Verpflichtungen
 - Anbieter darf annehmen, daß diese Verpflichtungen erfüllt sind
 - ⇒ Anbieter ist geschützt und wird weniger komplex
- Technisch: Aussagen über Variablen
 - Vorbedingungen
 - Nachbedingungen
 - Invarianten
- Initialisierung stellt Invarianten sicher
- Alle Methoden müssen Objekt-Invarianten aufrechterhalten

Modellierung

- Programmstruktur soll die Realität abbilden
 - Leichter verständlich für Programmierer
 - Immer klarer Bezug zur Anwendung
 - Kleine Änderungen der Realität bedingen kleine Anpassungen
- Klassisch (Booch, Coad, Yourdon)
 - Jedes Objekt repräsentiert ein Objekt der Realität
 - Seine Felder beschreiben die Eigenschaften des realen Objektes
- Simplizistisch
 - Objekte = Nomen der Problembeschreibung
 - Methoden = Verben der Problembeschreibung

Hierarchien

- Erweiterung der Modellierung
- Spezialfälle von realen Objekten werden durch Subklassen abgebildet.
 - Superklasse (Basisklasse) enthält allgemeingültige Daten & Methoden
 - Spezialfälle fügen noch neue Details hinzu
 - Spezialfälle ändern Details ab
- **Substitutability** (“Ersetzbarkeit”)
 - Spezialfälle können für allgemeine Fälle eingesetzt werden
 - Instanzen der Subklasse können wie Instanzen der Superklasse verwendet werden
 - Die **is-a** Beziehung zwischen Klassen

Vererbung als Sprachkonstrukt

- `class A extends B`

⇒ A

- übernimmt alle Felder und Methoden von B
- kann neue Felder hinzufügen
- kann neue Methoden hinzufügen
- kann den Rumpf von B-Methoden ersetzen (überschreiben)

- Die Schnittstelle von A

- bietet sicher mehr Funktionalität als die von B
- ist also “abwärtskompatibel”

⇒ Substitutability: Alle A Instanzen können als B-Instanzen benutzt werden

Beispiel: Daten von Shape → Circle, Rectangle

```
class Shape {
    protected int x,y; // Position des Zeichnungsobjektes
    protected Drawing drawing; // Zeichnung, die Objekt enthält
    ...
}
class Circle extends Shape {
    protected int r; // Radius
}

class Rectangle extends Shape {
    protected int w,h; // Ausdehnung
    ...
}
class Drawing {
    private Vector shapes;
    public void add(Shape s) { // für Rectangle, Circle
        shapes.add(s);
    }
    ...
} }
```

Beispiel: Methoden von Shape

```
abstract class Shape {  
    ...  
    protected void fireChange() {  
        if (drawing!=null)  
            drawing.childChanged(this);  
    }  
    public void moveTo(int x, int y) {  
        this.x = x;  
        this.y = y;  
        fireChange();  
    }  
}
```

- fireChange ist Hilfsmethode für abgeleitete Klassen
- moveTo ist gemeinsame Methode zur Positionsänderung
- Benutzt in Rectangle, Circle

Initialisierung bei Vererbung

- Initialisierung stellt Invarianten sicher
 - Die Superklasse initialisiert ihre Felder, da sie ihre Invarianten gekennt
 - Die abgeleitete Klasse initialisiert die neu hinzugekommenen Felder
- Aufruf `super(...)` als erstes Statement im Konstruktor übergibt Parameter für Konstruktor der Superklasse
- Wenn der `super()`-Aufruf fehlt, setzt der Compiler `super()` ein
(. . . auch wenn Eclipse glaubt, man müsse `super()` hinschreiben)

Beispiel: Initialisierung von Shape

```
abstract class Shape {
    protected int x,y;
    public Shape(int x, int y) {
        this.x = x;
        this.y = y;
    } ... }
class Rectangle extends Shape {
    protected int w,h;
    public Rectangle(int x, int y, int w, int h) {
        super(x,y);
        this.w = w;
        this.h = h;
    } ... }
```

- Aufruf des `super()`-Konstruktors zur Initialisierung
- Typisch: Konstruktor der Subklasse hat mehr Argumente

Beispiel: Daten & Methoden von ShapeEdit

```
abstract class ShapeEdit {
    protected DragMark dragMark[];
    public void paint(Graphics g) {
        for (int i=0; i!=dragMark.length; ++i)
            dragMark[i].paint(g);
    }
    protected DragMark curDrag;
    public boolean startDrag(int x, int y) { ... }
    public boolean wouldDrag(int x, int y) { ... }
    public void endDrag() { ... }

    public abstract void dragTo(int dx, int dy);
}
```

- Gemeinsam: Verwaltung und Anzeige von DragMarks
 - Subklassen implementieren spezielle Reaktion auf Dragging
- ⇒ Abstrakte Klassen sind noch nicht vollständig

Vererbung für Bild-Elemente

- Ziehmarken und Größenbeschreibung ähnlich zu Rectangle
- Zusätzlich Dateiname und geladenes Bild für Anzeige
- Bei Erstellung Dialog für Dateiauswahl

⇒ Funktioniert sofort

- Anzeige Umriss und DragMarks
- Reaktion auf Benutzereingaben
- Fehlen
 - Laden von Bildern
 - Auswahl eines Bildes

Picture **is-a** Rectangle

```
public class Picture extends Rectangle {
    protected String file;
    protected ImageIcon icon;

    public Picture(int x, int y, int w, int h, String file) {
        super(x,y,w,h);
        this.file = file;
    }
    public void force() {
        if (icon == null) loadFromDisk();
    }
    protected void loadFromDisk() {
        icon = new ImageIcon(file);
    } }
}
```

- Verwaltung x, y, w, h ganz in Superklasse
- icon bleibt zunächst null
- erst ein force erzeugt das (vielleicht grosse) Speichericon

PictureEdit **is-a** RectangleEdit

```
public class PictureEdit extends RectangleEdit {
    protected Picture pic;
    public PictureEdit(Picture pic) {
        super(pic);
        this.pic = pic;
    }
    public void paint(Graphics g) {
        if (pic.isAvailable())
            g.drawImage(pic.getImage(),...);
        else g.drawString(pic.getFilename(),...);
        super.paint(g);
    }
    public JPopupMenu getPopupMenu() { ... }
```

- Funktionalität des Rechtecks vollständig in Superklasse
- Nur paint für Darstellung und getPopupMenu für Befehle überschrieben
- paint benutzt Funktionalität der Superklasse

Umsetzung von Design-Ansätzen in Java

- Designansätze und Umsetzung in Java

Geheimnisprinzip	→	<code>private, protected, public</code>
Invarianten	→	Initialisierung, Konstruktoren, <code>super()</code>
Modellierung	→	Felder und Methoden
Hierarchien	→	Vererbung

- Andere Sprachen setzen gleiche Design-Ansatz anders um

- Sinnvolle Verwendung der Sprache nur bei Wissen um Design

- Wofür hat der Erfinder von Java ein Konstrukt vorgesehen?
- Wozu kann ich ein Konstrukt gut, wozu gar nicht einsetzen?
- Zwinge ich Java zu Verrenkungen oder arbeiten wir zusammen?
- Respektiere ich Einschränkungen oder versuche ich sie zu umgehen?

- Sprechen über Java setzt Verständnis von Design voraus