

# Objekt-Orientierte Programmiersprachen

Martin Gasbichler, Holger Gast

3. November 2005

---

## Bisher: Grundtechniken OO Design

- Client/Server Beziehung  
→ Eine Kunde nutzt die Dienste eines Anbieters
- Parnas' Geheimnisprinzip  
→ Der Anbieter ist der Experte, der Kunde muß die Details nicht kennen
- Design by Contract  
→ Verträge müssen detailliert sein (Vor-, Nachbedingungen, Invarianten)
- Modellierung des Anwendungsgebietes  
→ Programmstruktur bildet die Realität ab
- Hierarchien  
→ Spezialfälle werden Subklassen

---

## Plan für die erste Semesterhälfte

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

---

## Bisher: OO-Design und Java

- Designansätze und Umsetzung in Java
  - Geheimnisprinzip → private, protected, public
  - Invarianten → Initialisierung, Konstruktoren, super()
  - Modellierung → Felder und Methoden
  - Hierarchien → Vererbung
- These: Ohne Verständnis für Design kein Verständnis für Java
  - Sprachkonstrukte sind für Design-Ansätze eingeführt
  - Richtige Verwendung der Sprache durch Verständnis der Absicht
  - Einschränkungen von Java begreifen und akzeptieren

---

## Erinnerung: Klassenhierarchien

- Vererbung unterstützt Wiederverwendung von Code
- Adaption von allgemeingültigen Klassen für Spezialfälle
- Erweiterung einer bestehenden Anwendung
- Beispiel: Klassen `Picture`, `PictureEditor` für Bildanzeige
- Neues Beispiel jetzt: Einträge im Popup-Menü

---

## Das Action Interface

```
void    actionPerformed(ActionEvent e)
boolean isEnabled()
void    setEnabled(boolean b)
Object  getValue(String key)
void    putValue(String key, Object value)
void    addPropertyChangeListener(PropertyChangeListener l)
void    removePropertyChangeListener(PropertyChangeListener l)
```

- Drei Aufgaben in einer Schnittstelle:
  - Eigentlich wichtig: `actionPerformed`
  - Eigenschaft `enabled`: Auswählbar (oder grayed out)?
  - Properties: `putValue`, `getValue`, `addPropertyChangeListener`, `removePropertyChangeListener`
- Ein ziemlich kompliziertes Interface für einen Menüeintrag!

---

## Ergänzung: Anonyme Klassen

```
public JPopupMenu getPopup() {
    JPopupMenu ret = super.getPopup();
    ...
    ret.add(new AbstractAction("Bild laden") {
        public void actionPerformed(ActionEvent e) {
            pic.force();
        }
    });
    return ret;
}
```

- Action-Objekte bündeln ...
    - den Text eines Menü-Eintrags (auch Toolbar, ...)
    - das Icon des Eintrags (optional)
    - den auszuführenden Code bei Auswahl des Eintrags
- ⇒ `JPopupMenu.add()` erwartet Instanz von `Action`
- ⇒ Klassisch brauchen wir eine neue Klasse für jeden (!) Menüeintrag

---

## Properties von Action

- Properties bieten uniformen Zugriff auf Informationen.
- Namen von Properties für Zugriff in `putValue`, `getValue` sind Strings
- Action definiert Konstanten
  - `ACCELERATOR_KEY` Tastaturkürzel für zum Auslösen der Aktion
  - `ACTION_COMMAND_KEY` Identifier für generierte `ActionEvents`
  - `LONG_DESCRIPTION` Textangabe für interaktive Hilfefunktionen
  - `SHORT_DESCRIPTION` Textuelle Kurzbeschreibung für Tooltips
  - `NAME` In Menüs und Buttons angezeigter Name der Aktion
  - `SMALL_ICON` Icon, das in Toolbars für die Aktion angezeigt wird

---

## Mit AbstractAction halb zum Ziel

- AbstractAction implementiert administrative Aufgaben
    - Eigenschaft enabled
    - Properties und PropertyChangeListeners
  - nur actionPerformed ist (natürlich) noch nicht implementiert
- ⇒ actionPerformed bleibt abstract  
→ AbstractAction ist abstrakte Klasse
- ⇒ Zur Verwendung müssen wir
- Eine abgeleitete Klasse definieren
  - Die Methode actionPerformed implementieren
  - Eine Instanz der neuen Klasse für JPopupMenu.add() erzeugen
- Genau für diese Situation gibt es anonyme Klassen
  - vgl. These: Programmkonstrukte werden für Designkonstrukte eingeführt

---

## Verwendung von Anonymen Klassen

```
ret.add(new AbstractAction("Bild laden") {  
    public void actionPerformed(ActionEvent e) {  
        pic.force();  
    }  
});
```

- Definition einer anonymen Subklasse von AbstractAction
- Methode actionPerformed überschrieben
- Neue Instanz mit Argument "Bild laden"
- Aufruf Konstruktor von AbstractAction
- Hinweise:
  - Zugriff auf Feld pic der umgebenden Klasse → Closures
  - Der Konstruktor AbstractAction(String) wird offenbar an die anonyme Klasse vererbt (!)

---

## Anonyme Klassen sind auch nur Klassen – wirklich?

- Bei Überraschungen lohnt ein Blick in die Java Sprachspezifikation
- Anonyme Klassen als Klassen
  - 15.9.5 Anonymous Class Declarations
  - An anonymous class declaration is automatically **derived** from a class instance creation expression by the compiler. An anonymous class is **never abstract** (§8.1.1.1). An anonymous class is **always an inner class** (§8.1.2); it is **never static** (§8.1.1, §8.5.2). An anonymous class is always **implicitly final** (§8.1.1.2).
- Die Definition bezieht sich auf vorher definierte Konzepte
- Wir kennen alle Konzepte von normalen Klassen
- Und wie ist die Sache mit dem Konstruktor?

---

## Konstruktoren Anonymer Klassen

### 15.9.5.1 Anonymous Constructors

An anonymous class **cannot have an explicitly declared constructor**. Instead, the **compiler must automatically provide** an anonymous constructor for the anonymous class. The form of the anonymous constructor of an anonymous class C with **direct superclass S** is as follows:

[ . . . ] The actual arguments to the class instance creation expression are used to determine a constructor **cs of S**, using the same rules as for method invocations (§15.12). The type of each **formal parameter of the anonymous constructor** must be identical to the corresponding **formal parameter of cs**.

The **body of the constructor consists of an explicit constructor invocation (§8.8.5.1) of the form super(...)**, where the actual arguments are the formal parameters of the constructor [ . . . ]

## All neat and orderly again (Poirot)

- Die anonyme Klasse erbt *nicht* die Konstruktoren der Superklasse
- Stattdessen tut der Compiler drei übliche Schritte
  - Er stellt fest, welchen Konstruktor `cs` der Superklasse der Benutzer gemeint hat
  - Er generiert einen neuen Konstruktor `ca` in der anonymen Klasse, der diesen Konstruktor mittels `super()` aufruft
  - Er erzeugt für das `new` einen Aufruf des neu generierten Konstruktors `ca`, um die Instanz anzulegen
- Damit ist das unbekannte Konzept auf bekannte zurückgeführt
- Die Erklärung ist nur dann hilfreich, wenn man die Konzepte von objekt-orientierten Sprachen kennt
- (vgl. These der Vorlesung)

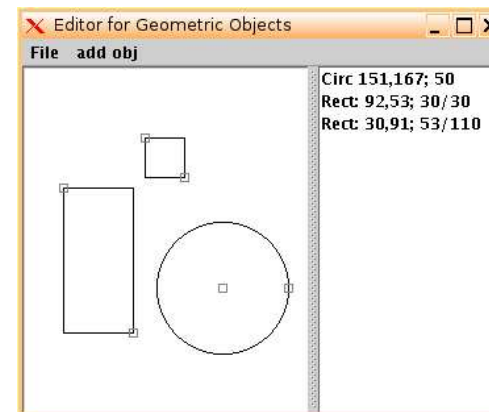
## Fortgeschrittenes OO Design

- Responsibility Driven Design
  - Betrachte Gruppen von Objekten statt Paaren
  - Strategische Aufgabenverteilung
- Refactoring
  - Das erste Design ist meist nicht das beste
- Design Patterns
  - Wiederverwendbares, getestetes Design
  - Von den Experten lernen

## Responsibility Driven Design (RDD)

- Vorgeschlagen von Rebecca Wirfs-Brock (1989, 2003)
- Metapher: Objekte bilden Gruppen und erreichen gemeinsam Ziele
  - Gruppen teilen Aufgaben untereinander auf
  - Jedes Mitglied ist für seine Aufgaben verantwortlich
  - Jedes Mitglied delegiert fremde Aufgaben an Kollegen (*collaborators*)
- ⇒ Grundfrage: Wie verteilt man Teilaufgaben möglichst geschickt auf die Objekte, so daß sie gemeinsam die Gesamtaufgabe lösen?
- Bisherige Ansätze waren
  - bilateral (Sie betrachten nur zwei Objekte gleichzeitig)
  - gerichtet (Client benutzt Server)
  - unsymmetrisch (Server ist nur Helfer für den wichtigen Client)

## Erinnerung: Der Editor



- EditorPane links
- ShapeEdits für Kreise und Rechtecke
- Reaktion auf Maus:
  - Dragging
  - Cursor-Änderung
  - Popup Menüs

---

## Beispiel: Mauseingaben im Editor

- Im EditorPane soll
  - der Maus-Cursor sich ändern, wenn er über einer DragMark steht
  - eine angewählte DragMark gezogen werden
  - ein Popup-Menü für Befhle an einzelne Objekte zur Verfügung stehen
- Drei Aufgaben, drei Objekte
  - CursorAdapter achtet nur darauf, ob sich der Cursor ändern soll
  - DragAdapter wartet darauf, dass der Benutzer eine DragMark zieht
  - PopupAdapter zeigt bei Bedarf Popup-Menü an
- Verwaltung der einzelnen Editoren erledigt EditorPane

⇒ EditorPane kann sich auf seine "Kernkompetenz" konzentrieren

---

## CursorAdapter mit Kernkompetenz

```
public class CursorAdapter ... {
    private EditorPane editor;
    public CursorAdapter(EditorPane editor) {
        this.editor = editor; // editor also Kollege
    }
    public void mouseMoved(MouseEvent ev) {
        if (editor.findWouldDrag(ev.getX(), ev.getY()) != null)
            editor.setCursor(new Cursor(Cursor.MOVE_CURSOR));
        else
            editor.setCursor(null);
    }
    public void mouseDragged(MouseEvent e) { }
}
```

- Einzige Aufgabe: Bei Mausebewegungen Cursor ändern
- EditorPane wird Kollege: Gibt es eine DragMark an der Mausposition?

---

## EditorPane mit Kernkompetenz

```
private ShapeEditFactory editFactory;
private Drawing drawing;
private HashMap edits;

public EditorPane(Drawing drawing) {
    ...
    DragAdapter da = new DragAdapter(this);
    addMouseListener(da);
    addMouseMotionListener(da);

    addMouseMotionListener(new CursorAdapter(this));

    addMouseListener(new EditPopupAdapter(this));
}
```

- Eigene Aufgaben: Verwaltung von Zeichnung und Editoren
- Delegiert: Reaktion auf Benutzereingaben

---

## DragAdapter mit Kernkompetenz

```
public class DragAdapter ... {
    private EditorPane editor;
    private ShapeEdit dragEdit;
    public void mousePressed(MouseEvent ev) {
        dragEdit = editor.findWouldDrag(ev.getX(), ev.getY());
        if (dragEdit != null)
            dragEdit.startDrag(ev.getX(), ev.getY());
    }
    public void mouseDragged(MouseEvent ev) {
        if (dragEdit != null) {
            dragEdit.dragTo(ev.getX(), ev.getY());
        } } }
}
```

- Aufgabe: Verwaltung des aktuell selektierten Objekt
- Kollegen:
  - editor für Suche nach selektiertem Objekt
  - dragEdit für die eigentliche Änderung der Daten

---

## EditPopupAdapter für Menüs

```
public class EditPopupAdapter extends MouseAdapter {
    protected EditorPane pane;
    ...
    public void mouseClicked(MouseEvent e) {
        if (e.isPopupTrigger()) {
            ShapeEdit edit = pane.findWouldDrag(e.getX(), e.getY());
            if (edit != null) {
                JPopupMenu popup = edit.getPopup();
                ...
                popup.add(new ShapeEditAction("Löschen", edit) {
                    public void actionPerformed(ActionEvent e) {
                        pane.removeByEdit(edit);
                    }
                });
                popup.show(pane,e.getX(), e.getY());
            } } } }
```

- EditPopupAdapter wartet auf Popup-Event (Rechte Maustaste)
- Menü aus speziellen und gemeinsamen Aktionen

---

## Refactoring im Beispiel

- Refactoring = Verbesserung der Code-Struktur bei gleicher Funktionalität
- Ursprünglich ...
  - hatte EditorPane Methoden
    - mousePressed, mouseReleased → Anfang und Ende
    - mouseDragged → Verschieben einer Marke
    - mouseMoved → Cursor-Änderung
    - die alle auf drawing direkt zugegriffen
  - mussten diese Methoden gleichzeitig zwei Aufgaben erledigen
  - war EditorPane ein "Einzelkämpfer"
- Jetzt ...
  - ist DragAdapter nur für Dragging zuständig
  - ist CursorAdapter nur für Cursor zuständig
  - ist EditPopupAdapter nur für Menüs zuständig

---

## Sprachdesign für RDD

- Objekte sind klein und eigenständig
- Zyklen im Objektgraph sind natürlich (Objekte bilden keine Bäume)
- Auswirkungen auf das Speichermanagement
  - Klassisches "Client gibt Server frei" funktioniert nicht
  - Daher unklar: Wann kann ein Kollege zerstört werden?
  - Zyklen: Reference counting nicht ausreichend⇒ Brauchen echten Garbage Collector
- Nicht benötigt werden
  - Klassen
  - Vererbung
  - (Anonyme Klassen)⇒ Diese sind nur Hilfsmittel zur Implementierung

---

## Responsibilities nach Refactoring

- Die Aufgaben sind verteilt und klein
- Jedes Objekt übernimmt eine einzige, klar definierte Teilaufgabe
- Invarianten für einzelne Objekte überschaubar
- Gemeinsam erledigen sie die Gesamtaufgabe
- Eleganz
  - Jedes Objekt erledigt das, was es aufgrund seiner Felder gut kann
  - Für die anderen Aufgaben bittet es Kollegen um "Amtshilfe"
  - Jedes Objekt hat eine knappe, klare Beschreibung
  - Die Methoden sind kurz und offensichtlich
  - Die Komplexität ist durch Zusammenarbeit aufgelöst

---

## Bezug zu klassischen Techniken

- Objekte sind abwechselnd Client und Server
- Objekte schließen Verträge mit verschiedenen Partnern, die Verträge ergänzen und bedingen sich
- Modellierung hilft nur anfänglich, letztlich muß die Software in sich stimmig sein
- Klassenhierarchie nur Implementierung für Wiederverwendung von Code

---

## Warum Design Patterns in dieser Vorlesung?

- Design Patterns beschreiben häufig verwendete Designs
- Eine OO-Sprache muß also Patterns gut unterstützen
- Patterns bieten viele Rollen
- (Allgemeinbildung)

---

## Responsibilities und Rollen

- Objekte interagieren mit verschiedenen Partnern
- Sie erfüllen verschiedene Rollen
- Jede Rolle erfordert nur eine Teilmenge der Schnittstelle
  - ⇒ Deklaration von Teilen der Schnittstelle
  - ⇒ Sprachkonstrukt: Interfaces
- Unabhängigkeit der Rollen → gut für Wiederverwendung & Erweiterung
- Substitutability
  - Bisher: Subklassen-Instanzen für Superklassen-Instanzen
  - Jetzt: Jede Instanz, die ein Interface implementiert

---

## Design Patterns

- Design patterns beschreiben erprobte Lösungen
  - (Aussagekräftiger) Name
  - Problembeschreibung
  - Lösungsbeschreibung (Klassen und Objekte)
- Gamma, Helm, Johnson, Vlissides: Design Patterns, 1995 genannt GoF-Patterns (gang of four)
- Vorteile
  - Details ergeben sich aus Anwendung eines Patterns
  - Irrwege vermieden
  - Kommunikation erleichtert

---

## Heute: Drei Patterns

- Observer: Benachrichtigung über Änderungen
- MVC: *Das* Pattern für GUI-Anwendungen
- Composite: Algebraische Datentypen in Java

---

## Observer Pattern

- Objekte ändern dynamisch ihren Zustand
- Observer Pattern erlaubt Reaktion auf Änderungen
- Subject = Beobachtetes Objekt
  - Hält Liste von aktuellen Observern
  - Ruft Methode in allen Observern bei Zustandsänderung auf
- Die Beobachtung ist nur Nebenaufgabe des Observers
  - Vererbung wird aber für Hauptaufgabe eingesetzt
  - Verwendung von Interfaces statt Vererbung
  - Die Rolle als Observer
- Variante für zusammengesetzte Objekte
  - Registrierung bei Teilobjekten ineffizient
  - ⇒ Oft Teilobjekte implizit beobachtet durch Observer für Gesamtobjekt

---

## Beispiel: Editor

```
interface ShapeListener {
    void shapeChange(Drawing list, Shape s);
    void shapeAdd(Drawing list, Shape s);
    void shapeRemove(Drawing list, Shape s);
}
```

- Ein Observer für graphische Objekte wird benachrichtigt über
  - Änderung der Daten (Position, Größe)
  - Neues Objekt in Zeichnung
  - Löschen eines Objektes in Zeichnung
- In der Praxis
  - Mehr Details über Änderungen für Effizienz
  - Ein Event-Objekt als Parameter

---

## EditorPane als Observer

```
class EditorPane extends JComponent
implements ShapeListener {
    ...
    public void setDrawing(Drawing drawing) {
        if (this.drawing != null)
            this.drawing.removeShapeListener(this);
        drawing.addShapeListener(this);
        ...
    }
    public void shapeChange(Drawing d, Shape s) {
        ((ShapeEdit)edits.get(s)).refresh();
        repaint();
    }
    ...
    public void shapeAdd(Drawing d, Shape s) { ... }
    public void shapeRemove(Drawing d, Shape s) { ... }
}
```

- Invariante: EditorPane ist als Listener bei drawing registriert

---

## Model-View-Controller

- Die Daten sollen unabhängig von Anzeige sein
  - Komplexität verkleinern
  - Unabhängige Programmierung & Tests
  - Portierung der Anzeige für andere Umgebung
- Die Benutzereingaben werden getrennt von der Ausgabe behandelt

⇒ Responsibility Driven Design

---

## Beispiel: Editor

- Modell: Shape, Circle, Rectangle, Picture, Drawing
- View+Controller:
  - ShapeEdit, CircleEdit, RectangleEdit, PictureEdit
  - EditorPane als Koordinator
- Zweiter View: TextView

---

## MVC-Struktur

- Modell
    - Daten der Anwendung
  - View
    - Darstellung von Daten in graphischer Form
    - Nachführung der Anzeige bei Änderung der Daten (→ Observer)
  - Controller
    - Annahme von Benutzereingaben (events)
    - Ableich mit graphischer Darstellung (View)
    - Interpretation als Änderung der Daten
- ⇒ Nachführung der Anzeige immer indirekt
- ⇒ Verschiedene Anzeigen sind automatisch synchron
- Kopplung View—Controller → “View+Controller = UI-delegate”

---

## Die TextView Klasse

```
class TextView extends JList
implements ShapeListener {
    public TextView(Drawing l) {
        super(new DefaultListModel());
        l.addShapeListener(this);
    }
    public void shapeChange(Drawing list, Shape obj) {
        DefaultListModel m = (DefaultListModel)getModel();
        int i = m.indexOf(obj);
        m.setElementAt(obj,i); // trigger ChangeEvent für Anzeige
    }
    public void shapeAdd(Drawing list, Shape obj) { ... }
    public void shapeRemove(Drawing list, Shape obj) { ... }
}
```

- Vererbung für Hauptaufgabe Anzeige
- Interface für Observer-Rolle
- Delegation an DefaultListModel

---

## Composite Pattern

- Baumstrukturen mit Objekten darstellen
- Jeder Knoten wird zu einem Objekt
- Jedes Knoten-Objekt kann seine Kind-Knoten liefern
- Vergleich: Algebraische Datentypen

```
type int_tree =  
  Empty  
  | Node of int_tree * int * int_tree
```

---

## Beispiel: Tree als Composite

```
abstract class IntTreeNode {  
  protected Vector children; // ineffizient  
  public boolean hasChildren() {  
    return children != null && children.size()>0;  
  }  
  public IntTreeNode getChild(int i) { // kann fehlschlagen  
    return (IntTreeNode)children.get(i); // bounds-check  
  }  
  public abstract int getValue();  
}  
class IntTreeInner extends IntTreeNode {  
  protected int v;  
  public int getValue() { return v; }  
}  
class IntTreeEmpty extends IntTreeNode {  
  public int getValue() { throw (new RuntimeException("No value"))  
}
```

---

## Fortgeschrittenes OO-Design

- Design betrachtet immer Gruppen von Objekten
- Die Grundtechniken bleiben sinnvoll für Ausschnitte des Designs
- Interfaces
  - Beschreiben Teilmengen von Schnittstellen
  - Definieren Rollen von Objekten
  - Lassen die Vererbung für die Hauptaufgabe der Implementierung
  - Sind gute Alternative zu Multiple Inheritance (→ C++)
- Design Patterns
  - beschreiben getestete Designs von Experten
  - definieren Rollen von Objekten
  - müssen in OO-Sprachen leicht umsetzbar sein