Objekt

- Objekt kapselt Variablen und Routinen
- Interaktionen zwischen Objekten durch Senden von Nachrichten und Reagieren auf empfangene Nachrichten
- Eigenschaften jedes Objekts:
  - Identität (identisch = mehrere Namen für ein Objekt)
  - Zustand (gleich $\neq$ identisch)
  - Verhalten (Reaktion auf Nachrichten)
- Software-Objekt simuliert und abstrahiert reales Objekt

Beispiel eines Objekts

Objekt: einStack

nicht öffentliche (private) Variablen:

`elems`:	`"a"`	`"b"`	`"c"`	`null`	`null`
`size`:	3				

öffentlich aufrufbare Routinen:

`push`:	Implementierung der Routine
`pop`:	Implementierung der Routine

Objektschnittstelle, Datenabstraktion

- Schnittstelle beschreibt, was von außen sichtbar ist
- data hiding (black box, grey box)
- Datenabstraktion = data hiding + Kapselung
- Datenabstraktion wichtig für Wartung
- mehrere unterschiedliche Schnittstellen pro Objekt
  = Datenabstraktionen für unterschiedliche Sichtweisen

Klasse

- Klasse beschreibt Struktur ihrer Instanzen (= Objekte)
- Konstruktoren zur Erzeugung aller Instanzen
- alle Instanzen einer Klasse haben dieselben Schnittstellen und dasselbe Verhalten
- nicht-identische Instanzen haben unterschiedliche Instanzvariablen, möglicherweise unterschiedliche Zustände
- jedes Objekt ist Instanz genau einer Klasse

Beispiel einer Klasse (1)

```
public class Stack {
    private String[] elems;
    private int size = 0;
    // constructor initializes new stack of size sz
    public Stack (int sz) {
        elems = new String[sz];
    }
    // push puts elem onto the stack if it is not full
    public void push (String elem) {
        if (size < elems.length) {
            elems[size] = elem;
            size = size + 1;
        }
    }
    ...
```

Beispiel einer Klasse (2)

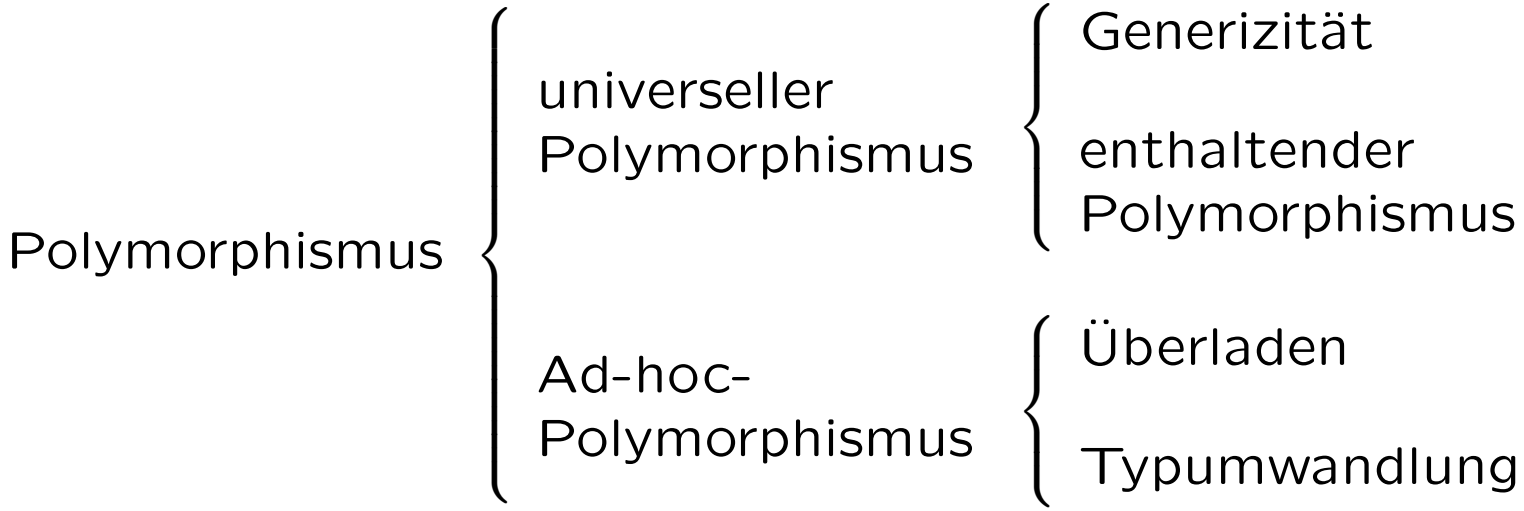
```
    ...
    // class Stack continued

    // pop returns element taken from the stack if it
    // is not empty; otherwise pop returns null
    public String pop() {
        if (size > 0) {
            size = size - 1;
            return elems[size];
        }
        else
            return null;
    }
}
```

Beispiel einer Klasse (3)

```
class StackTest {
    // main executes tests on a new instance of Stack
    public static void main (String[] args) {
        Stack s = new Stack(5);
        int i = 0;
        while (i < args.length) {
            s.push(args[i]);
            i = i + 1;
        }
        while (i > 0) {
            i = i - 1;
            System.out.println(s.pop());
        }
    }
}
```

Polymorphismus

Objekt hat gleichzeitig mehrere Typen ($\approx$ Schnittstellen)



Enthaltender Polymorphismus

- Untertyp beschreibt Instanzen genauer als Obertyp
- Instanzenmenge des Obertyps enthält jene des Untertyps

  Beispiel: Instanz von `Student` ist auch Instanz von `Person`
- Ersetzbarkeitsprinzip = Def. von Untertypbeziehungen:

  U ist Untertyp von T wenn Instanz von U überall verwendbar ist, wo Instanz von T erwartet wird
- Variable hat deklarierten und dynamischen Typ
- dynamischer Typ oft spezieller als deklarierter Typ
- dynamisches Binden

Vererbung

- Ableitung neuer Klassen aus existierenden Klassen
- Änderungsmöglichkeiten:
  - Erweiterung um Methoden und Variablen
  - Überschreiben von Methoden
- Zusammenhang mit enthaltendem Polymorphismus:

  Vererbungsbeziehung $\approx$ Untertypbeziehung

  (in Java, C#, C++, . . . , aber NICHT GENERELL)

Beispiel für Vererbung (1)

```
public class CounterStack extends Stack {
    private int counter;
    // constructor initializes new stack of size sz
    // and a counter with initial value c
    public CounterStack (int sz, int c) {
        super(sz);
        counter = c;
    }
    // push puts elem onto the stack if it is not full
    // and increments the counter by 1
    public void push (String elem) {
        counter = counter + 1;
        super.push(elem);
    }
    ...
```

Beispiel für Vererbung (2)

```
    ...
    // class CounterStack continued

    // pop inherited from Stack

    // count puts the current value of the counter
    // onto the stack if it is not full
    // and increments the counter by 1
    public void count() {
        push (Integer.toString(counter));
    }
}
```

Qualität von Programmen

- Brauchbarkeit:
  - Zweckerfüllung
  - Bedienbarkeit
  - Effizienz des Programmablaufs
- Zuverlässigkeit
- Wartbarkeit:
  - Einfachheit
  - Lesbarkeit
  - Lokalität
  - Faktorisierung

Programmerstellung und Wartung

- Schritte in Softwareentwicklungsprozessen:
  - Analyse
  - Entwurf
  - Implementierung
  - Verifikation und Validierung
- Arten von Softwareentwicklungsprozessen:
  - Wasserfallmodell
  - zyklische Prozesse mit schrittweiser Verfeinerung
- ständige Anpassung von Entwicklungsprozessen wichtig
- Wartung ist wesentlicher Kostenfaktor

Rezept für gute Programme

- Softwareentwicklung ist sehr komplex (unvollständiges Wissen, widersprüchliche Ziele, Zeitdruck)

  $\Rightarrow$ einfache Rezepte scheitern

- gezielter Einsatz von **Erfahrung** erhöht Erfolgsaussichten

  $\Rightarrow$ kollektive Erfahrung in Software-Entwurfsmustern

- objektorientierte Programmierung bietet besonders viele Möglichkeiten zur **Faktorisierung** von Programmen

  $\Rightarrow$ erleichtert gezielten Einsatz von Erfahrung

  $\Rightarrow$ überfordert Anfänger

Verantwortlichkeiten einer Klasse

- definiert durch drei w-Ausdrücke (Ich ist Instanz):
  - was ich weiß (Zustand der Instanzen)
  - was ich mache (Verhalten der Instanzen)
  - wen ich kenne (sichtbare Objekte, Klassen)
- EntwicklerInnen der Klasse zuständig für Änderungen in den Verantwortlichkeiten der Klasse

Klassen-Zusammenhalt

- Klassen-Zusammenhalt (class coherence) = Grad der Beziehungen zwischen den Verantwortlichkeiten der Klasse
- hoch, wenn
  - Variablen und Methoden eng zusammenarbeiten
  - und durch Klassenname gut beschrieben
- Klassen-Zusammenhalt soll hoch sein

  $\Rightarrow$ Hinweis auf gute Faktorisierung
  (wenig Änderungen nötig)

Objekt-Kopplung

- Objekt-Kopplung (object coupling)
  = Abhängigkeit der Objekte voneinander

- stark, wenn
  - viele sichtbare Methoden und Variablen
  - viele Nachrichten im laufenden System
  - viele Parameter in Methoden

- Objekt-Kopplung soll schwach sein

  $\Rightarrow$ Hinweis auf gute Kapselung
  (wenig unnötige Beeinflussung bei Änderungen)

Frühe Bewertung von Alternativen

Klassenzusammenhalt und Objektkopplung

- hängen oft zusammen (beides gut oder beides schlecht)
- sind bereits in früher Entwicklungsphase abschätzbar
- helfen bei der Bewertung von Alternativen

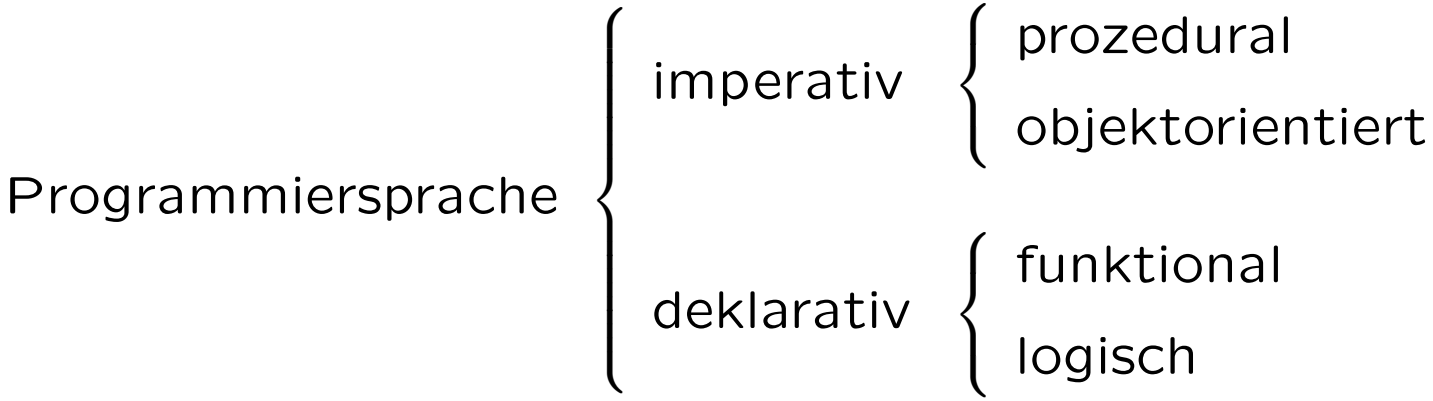
Refaktorisierung

- erste Faktorisierung oft nicht optimal
- Refaktorisierung = Änderung der Programmstruktur (ohne Änderung der Funktionalität)
- Refaktorisierung in Anfangsphase oft billig
- wenige gezielte Refaktorisierungen führen zu stabiler Zerlegung in Objekte (braucht nicht mehr geändert werden)
- refaktorisieren bevor Probleme sich über das ganze Programm ausbreiten

Codewiederverwendung

- geringere Kosten durch Codewiederverwendung wenn
  - EntwicklerInnen ausreichend erfahren
  - Zeit in Wiederverwendbarkeit investiert
- Fehlentscheidungen $\Rightarrow$ lange Entwicklungszeiten, Scheitern
- im Zweifelsfall weniger in Wiederverwendbarkeit investieren, bei Bedarf Refaktorisierungen nachholen
- nötige Refaktorisierungen nicht verzögern

Paradigmen der Programmierung



Eignung der Programmierparadigmen

- wenn Komplexität des Systems von Komplexität der Algorithmen dominiert

  $\Rightarrow$ prozedurale oder deklarative Programmierung

- wenn Komplexität des Systems deutlich höher als Komplexität der einzelnen Algorithmen (großes System)

  $\Rightarrow$ objektorientierte Programmierung