

Zusammenfassung, Vergleich und Ausblick

Martin Wirsing

in Zusammenarbeit mit
Matthias Hölzl, Piotr Kosciuzenko, Dirk Pattinson

06/03

Objektorientierte Programmierung und Software-Entwicklung

- Ein **Objekt** besitzt
 - lokalen Zustand (Attribute)
 - Methoden zur Änderung des Zustandes
- **Modulare Programmierung** durch Bildung von Klassen, Schnittstellen, Kapselung
- Unterstützung von Wiederverwendung und Änderung von Programmen durch **Vererbung**
- Robuste Programmierung durch **selbstdefinierte Ausnahmen**
- **Nebenläufige Programmierung**
- Systematische OO-SW-Entwicklung durch **UML**

Grundlagen der Informatik

- Parameterübergabemechanismen
- Korrektheit
- Komplexität

Vergleich mit funktionaler, imperativer Programmierung

- Ein funktionales Programm beschreibt **Werte**.

```
sumSquares : int -> int
fun sumSquares 0 = 0
  | sumSquares n = n*n + sumSquares(n-1)
```

Vergleich mit funktionaler, imperativer Programmierung

- Ein imperatives Programm beschreibt, wie der **(globale) Speicher modifiziert** wird.

```
s = 0;
i = 0;
while (i < n)
{
    i = i + 1;
    s = i * i + s;
}
```

Vergleich mit funktionaler, imperativer Programmierung

- Ein objektorientiertes Programm beschreibt die **Änderung lokaler Zustände** und berechnet Werte ebenfalls mittels Änderung lokaler Zustände.

```
class Square
{
    int data;
    int sumSquares()
    {
        int s = 0; int i = 0;
        while (i < data) {...}
        return s;
    }
}
```

Programme

- Ein **funktionales Programm** besteht aus Gleichungen, die Funktionen definieren.
- Ein **imperatives Programm** besteht aus Anweisungen, die zusammen mit dem Speicherinhalt betrachtet werden müssen.
- Ein **objektorientiertes Programm** (à la Java) verfeinert die imperative Sichtweise durch Betrachtung lokaler Zustände.

Module

- Module in funktionalen Programmen** bestehen aus der Definition von Datentypen und Funktionen.
- Module in imperativen Programmen** bestehen ebenso aus der Definition von Datentypen und Prozeduren.
- Module in OO-Programmen** bestehen aus Schnittstellen und Klassen (d.h. Aus der Definition von (privaten) Zuständen und Methoden).

Programmverifikation

Funktional:

- Programme **beschreiben direkt ihre Eigenschaften**: $\text{sumSquares } 0 = 0$ oder für $n > 0$: $\text{sumSquares } n = n * n + \text{sumSquares}(n-1)$
- Beweis durch Gleichheitsbeweise oder Induktion: $\text{sumSquares } 1 = 1 * 1 + \text{sumSquares}(1 - 1) = 1 + 0 = 1$

Imperativ:

- Verifikation durch Vor- und Nachbedingungen, die Zustandsmengen beschreiben
- Problem: Seiteneffekte
- Schwierig: Verifikation für nichttriviale Programme

Typen/Klassen

Funktionale Programmierung:

- starkes Typsystem \Rightarrow weniger Laufzeitfehler
- parametrische Polymorphie: polymorphe Typen und Funktionen über polymorphen Typen

Imperative/OO-Programmierung:

- imperativ: schwaches Typsystem OO: starkes Typsystem, ev. Mit Problemen bei Vererbung
- Vererbungspolymorphie für Methoden

Effizienz

Funktionale Programmierung:

- Sehr abstrakte Programme, oft weit entfernt von Maschinenebene
- dadurch oft Analyse der Komplexität schwierig

Imperative/OO-Programmierung:

- Näher an Maschinenebene
- Einfluß auf Repräsentation von Daten in Speicher
- Komplexität schwieriger festzustellen als bei imperativer Programmierung

OO-Programmierung:

- Zusätzliche Abstraktion im Vergleich zu rein imperativer Programmierung durch Methoden und Vererbung, Klassenbibliotheken
- Komplexität schwieriger festzustellen als bei imperativer Programmierung

Fazit

Aus Komplexitätsgründen oft imperative Sprachen bevorzugt

- aber Unterschiede sind oft nicht so groß
- Faktoren wie DB-Anbindung, I/O, GUI, Klassenbibliothek spielen oft eine größere Rolle

Ausblick: Neue Entwicklungen in Programmiersprachen

Neue Entwicklungen in Programmiersprachen

- Aspekt-orientierte Programmierung (AspectJ, HyperJ, ...)
 - „Einweben“ von Aspekten:
z.B. Logging, Zusicherungen,
Synchronisation beim Einbetten sequentieller in nebenläufige
Programme
- Agenten-orientierte Programmierung (Jade, Aglets, ...)
 - Autonome, reaktive und proaktive Programme
- Mobile Systeme (Ambients)
 - Mobile Computing (mobile Hardware)
 - Mobile Computation (mobiler Code)

Ausblick: Neue Entwicklungen im Software Engineering

- UML
 - Extreme Programming & UML
 - Model-driven Architecture
 - Generieren von Code aus den Modellen
 - Modelchecking von Software Entwürfen
- Software-Engineering für mobile Systeme
 - Agile Projekt (Global Computing Initiative der EU)
- Web-Engineering
 - Entwurf von Web-basierten Systemen
 - ICWE 04 in München an der LMU

Ausblick: Lehre

- Programmierpraktikum: Entwicklung eines verteilten Spiels
 - Nebenläufige Programmierung
 - Graphische Benutzeroberfläche
 - Teams mit 4-5 Mitgliedern
- Systempraktikum
- Informatik III
- Software-Engineering (5.Semester)
Requirements Engineering, SW-Architektur, Prozeßmodellierung,
Testen und Validierung, Web-Engineering
- Grundlagen der Systementwicklung
- Modelchecking-Praktikum
- Objekt-orientierte SW-Entwicklung