Zusammenfassung und Ausblick

Martin Wirsing

in Zusammenarbeit mit
Michael Barth, Philipp Meier und Gefei Zhang

02/05

Zusammenfassung: Objekt-orientierte Programmierung und Software-Entwicklung

Objekt-orientierte Programmierung und Software-Entwicklung

- Ein **Objekt** besitzt
  - lokalen Zustand (Attribute)
  - Methoden zur Änderung des Zustandes
- **Modulare Programmierung** durch Bildung von Klassen, Kapselung
- Unterstützung von Wiederverwendung und Änderung von Programmen durch **Vererbung**
- Robuste Programmierung durch **selbstdefinierte Ausnahmen**
- Software-Entwurf mit **UML-Klassendiagrammen**

Zusammenfassung: Objektorientierte Programmierung und Software-Entwicklung

Grundlagen der Informatik

- Syntax von Programmiersprachen (EBNF)
- Parameterübergabemechanismen
- Komplexität

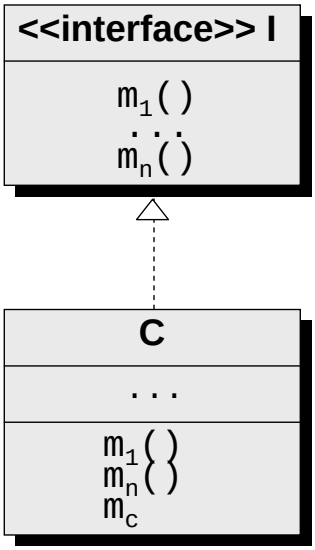
Zusammenfassung: Objektorientierte Programmierung und Software-Entwicklung

In der Vorlesung NICHT behandelt:

- **Einsatz von Schnittstellen** zur modularen Programmierung
- Abstrakte Klassen und Pakete
- **Nebenläufige Programmierung**
- **Systematische OO-SW-Entwicklung** mit UML

Schnittstellen

- Eine **Schnittstelle** in Java (Schlüsselwort „**`interface`**“) deklariert eine Menge von Methoden (ohne Angabe eines Rumpfs) und Konstanten (**aber** keine Attribute). Man nennt eine Methode ohne Rumpf „abstrakte Methode“. Im Gegensatz zu Klassen ist Mehrfachvererbung erlaubt, d.h. eine Schnittstelle kann Erbe mehrerer Schnittstellen sein.
- Eine Klasse `C` **implementiert** eine Schnittstelle `I`, wenn alle Methoden der Schnittstelle in `C` mit ihrer exakten Funktionalität implementiert werden, und zwar durch „öffentliche“ Methoden.



Beispiel: Schnittstelle für Listen (UML)

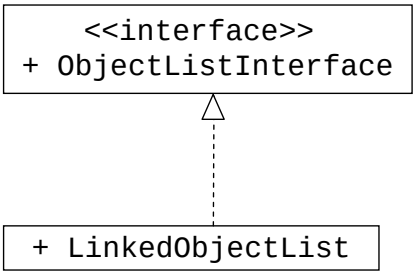
```
<<interface>>
+ ObjectListInterface
```

```
void clear()
void addFirst(Object o)
Object removeFirst() throws NoSuchElementException
boolean isEmpty();
int size();
. . .
```

Beispiel: Schnittstelle für Listen (Java)

```
public interface ObjectListInterface
{/** Removes all of the elements from this list. */
  void clear();

  /** Inserts the given Element at the beginning of this list. */
  void addFirst(Object o);

 /** Removes and returns the first element from this list. */
  Object removeFirst() throws NoSuchElementException;

/** Returns true if the list contains no elements. */
  boolean isEmpty();

/** Returns true if this list contains the specified element. */
  boolean contains(Object o);

/** Returns the number of elements in this list. */
  int size();
. . .
}
```


Beispiel: Implementierung von Listen




```
public class LinkedObjectList implements ObjectListInterface

{ private LinkedObjectListElem anchor;

  ...

}
```

Eine Schnittstelle für Listen II

```
/** Returns true if the list contains no elements. */
  boolean isEmpty();
/** Returns true if this list contains the specified element. */
  boolean contains(Object o);
/** Returns the number of elements in this list. */
  int size();
  /** Destructively reverses the list. */
  void reverse();
   ... <<Weitere Operationen wie etwa clone, iterate siehe später>>
}
```


Nebenläufige und verteilte Systeme

- Programme, wie wir sie in der Vorlesung geschrieben haben, arbeiten sequentiell, d.h. Aktivitäten werden nacheinander ausgeführt.
- Unter einem **System** versteht man eine von seiner Umgebung abgegrenzte Anordnung von Komponenten.
- Können in einem System mehrere Aktivitäten gleichzeitig stattfinden, man von einem **parallel ablaufenden (nebenläufigen) System**.
- Ist das System aus räumlich verteilten Komponenten aufgebaut, spricht man von einem **verteilten System**.

Threads

- In Java wird Nebenläufigkeit durch **Threads** realisiert. Ein nebenläufiges Java-Programm besteht aus **mehreren Threads**, die über **gemeinsame Objekte** miteinander kommunizieren.
- Ein **Thread** ist ein Teil eines Programms, das unabhängig von von anderen Teilen des Programms ausgeführt werden kann. Es repräsentiert eine einzige Sequenz von Anweisungen, d.h. einen sequentiellen Kontrollfluss, der nebenläufig zu anderen Threads ausgeführt werden kann, und der Daten gemeinsam mit anderen Threads benutzt.
- Aus Betriebssystemsicht wird ein **Prozess** verstanden als eine **abstrakte Maschine, die eine Berechnung ausführt**. Da ein Thread im Allgemeinen zusammen mit anderen Threads auf der gleichen abstrakten Maschine läuft und Ressourcen mit anderen Threads teilt, ist ein Thread ein „**leichtgewichtiger Prozess**“.

Software Engineering

- Software Engineering ist die Disziplin der systematischen Software-Entwicklung,

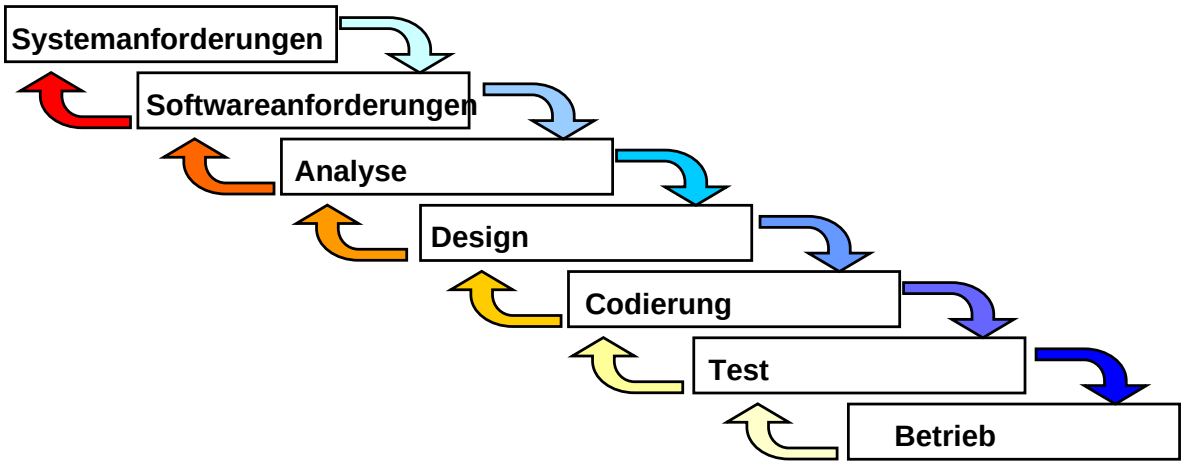
  dies bedeutet

  > die Bereitstellung und systematische Anwendung von Methoden, Verfahren und Werkzeugen zur Entwicklung, Betrieb und Wartung von Software. [***IEEE Std. 610.12 (1990)***]

- Software Engineering umfasst neben SW-Entwicklungstechniken auch Fragen der Managements, der Kosten, der Qualität und der Teamarbeit.

Software Engineering

- Eine klassische (heute vielfach kritisierte) Vorgehensweise ist das **Wasserfallmodell** mit den folgenden Phasen, die in den meisten Vorgehensweisen vorkommen:



- In der Vorlesung wurden nur Codierung sowie etwas Design und Test betrachtet.

Ausblick 1: Objekt-orientierte Programmierung und andere Programmierparadigmen

- Ein objekt-orientiertes 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;
      }
```

Beschreibt lokalen Zustand eines Square-Objekts auf der Halde

Variable im Keller; Keller ist global für das gesamte Programm

Vergleich mit imperativer Programmierung

- Imperative Programmierung basiert auf dem Konzept der **Anweisung.**
- Ein imperatives Programm arbeitet auf einem globalen Speicher und beschreibt, wie der **globale Speicher modifiziert** wird.
- Imperative Programmiersprachen sind Pascal, C, FORTRAN.
- **Beispiel:**

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

Variable im Keller; Keller ist global für das gesamte Programm

Vergleich mit funktionaler Programmierung

- Funktionale Programme basieren auf dem Konzept der **mathematischen Funktion**.
- Ein funktionales Programm beschreibt den **Zusammenhang von Argument und Wert einer Funktion**. Berechnungen hängen **NICHT** vom Zustand des Systems ab.
- Funktionale Programmiersprachen sind SML, Haskell, Gofer, CAML.
- **Beispiel:**

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

Ausblick 2: Neue Entwicklungen in Programmiersprachen

Neue Entwicklungen in Programmiersprachen

- **Aspekt-orientierte Progammierung** (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
- **Dienst-orientierte Programmierung**
  - Programme bestehen aus Diensten, die (im Web) veröffentlicht, gesucht und dynamisch kombiniert werden können

Ausblick 3: Neue Entwicklungen im Software Engineering

- **UML**
  - Extreme Programming & UML
  - Model-driven Architecture
    - Generieren von Code aus den Modellen
    - Modelchecking von Software Entwürfen
- **Web-Engineering und Dienstentwicklung**
  - Entwurf von Web-basierten Systemen
  - Entwicklung von Web-Diensten
  - ICWE 04 in München an der LMU

Ausblick: Lehre

- Einführung in die Informatik: Systeme und Anwendungen
  - Relationale Datenbanksysteme
  - Grundlagen der Betriebssysteme
  - Grundlagen des Internet: Netze, XML

Weiterführende Veranstaltungen:

- Datenbanksysteme (Vertiefung)
- Methoden der Software-Engineering
- Objekt-orientierte Software-Entwicklung (WS 05/06)
- . . .

Vielen Dank für Ihre Aufmerksamkeit und Mitarbeit!

Michael Barth, Philipp Meier, Gefei Zhang und ich wünschen Ihnen

viel Erfolg in der Klausur und

eine **angenehme vorlesungsfreie Zeit!**

Auf Wiedersehen!