Praktikum „SEP: Java-Programmierung“ SS 2019

Modularisierung

Karlheinz Friedberger und Thomas Bunk

Modularisierung

- Module gliedern Programme in einzelne, überschaubare und wiederverwendbare Bausteine
- Eine einzige, klar festgelegte Aufgabe pro Modul
- Klare Schnittstellendefinition
- Interner Zustand und Implementierungsdetails geheim (Geheimnisprinzip)
- Unterscheidung zwischen
  - ADO: Abstraktes Datenobjekt (einzigartig)
  - ADT: Abstrakter Datentyp (instanzierbares Modul mit Verhaltensdefinition)

Begriffsdefinition für objektorientierte Sprachen

Klasse	Instanzierbarer Bauplan mit konkreter Verhaltensdefinition (Implementierung)
Typ	Abstrakte Verhaltensspezifikation
Objekt	Instanz einer Klasse mit eigenem Zustand
Geheimnis	Interna der Klasse
Zugriff	Benutzung und ggf. Modifikation von Objekten
Kohäsion	Eine Klasse hat genau eine Aufgabe
Kopplung	Abhängigkeit verschiedener Klassen voneinander

Zugriffsmodifikatoren in Java

- Zwiebelmodell

`private`	Zugriff nur innerhalb der Klasse
„nichts“	+ im selben Paket
`protected`	+ in Unterklassen in anderen Paketen
`public`	+ überall sonst

- Weiterer wichtiger Modifikator:

`final`	Verhindert Schreibzugriff auf Attribute, Erstellung von Unterklassen, Neudefinition von Methoden

Objektorientes Programmieren in JAVA

- Programmaufteilung in Klassen
- Klasse: „Blaupause“ für Objekte
- Objektorientierung: Objekte arbeiten miteinander („dynamisch“), nicht Klassen („statisch“/`static`)
- Programmeinstieg immer statisch
- Wechsel von statisch zu dynamisch über Konstruktoraufruf mit `new`: Instanzierung von Klassen zu Objekten

Verwendung von `static`

Konstanten

Immer `static` und `final`

```
private static final ERROR_PREFIX = "Error! ";
```

Methoden

`static` genau dann, wenn keine Klasseninstanz verwendet wird

Utility-Klassen

Klassen ohne Objekte

- Nur statische Methoden
- Konstruktor ist `private` (und explizit angegeben)
- `final` Klasse verhindert Unterklassen

Attribute

Objektzustand kann nur nicht-statisch (sinnvoll) repräsentiert werden

Lokale Variablen

Immer nicht-statisch

Vererbung

- Darstellung von Subtypen-Beziehungen („is-a“-Relation)

```
class Bird extends Animal  ...
class Mammal extends Animal  ...
class Bunny extends Mammal  ...
class Cat extends Mammal  ...
```

- Mathematisch z.B. $\mathbb{N} \subset \mathbb{Z}$
- Liskovsches Substitutionsprinzip (= Konformanz)
  - Objekt des Untertyps haben mindestens alle Eigenschaften des Obertyps
  - Objekte des Untertyps sind überall dort verwendbar, wo Objekte des Obertyps verwendbar sind
  - In der Praxis oft mit Schwierigkeiten verbunden (z.B. `equals`)
- Empfehlung:
  - Flache Vererbungshierarchien bevorzugen
  - Komposition statt Vererbung
  - Oberklassen `abstract` (nicht instanzierbar) machen

Interfaces

```
public interface Trie {
    ...
}

public class SimpleTrie implements Trie {
    ...
}
```

- Schlüsselwort `interface` statt `class`
- Abstrakte „Oberklasse“
  - ohne eigenen Zustand
  - ohne eigene Methodenimplementierungen (außer `default`)
- Ausschließlich Schnittstellendefinition
- Einzige Möglichkeit für „Mehrfachvererbung“ in JAVA

Unterschied zwischen Methodenüberladung und -überschreibung

Methodenüberladung:

- Verschiedene Methoden mit gleichem Namen aber
- unterschiedlichen formalen Parametern und daher
- unterschiedlicher Signatur
- Haben nicht zwingend miteinander zu tun
  (bei gutem Stil aber meist schon)
- Setzen keine Vererbung voraus; innerhalb der selben Klasse möglich
- Bei statischen und dynamischen Methoden möglich

Methodenüberschreibung

- Nur im Kontext von Vererbungsbeziehungen
- Nur bei dynamischen Methoden
- Bezieht sich auf Methoden mit der gleichen Signatur (Name+Parameter) in Ober-/Unterklasse (auch transitiv)
- Überschreibende Methode in der Unterklasse redefiniert das

Dynamische Bindung

- Mächtiges Feature objektorientierter Programmiersprachen
- Liegt vielen gängigen Programmiermustern zu Grunde
  - (Abstract-)Factory-Pattern
  - Strategy-Pattern
  - Observer-Pattern
  - Visitor-Pattern (aka Double-Dispatch-Pattern)
  - ...
- Prinzip: Verhalten unterscheidet sich je nach (zur Compile-Zeit unbekanntem) Laufzeittyp des Objekts, auf dem eine Methode aufgerufen wird