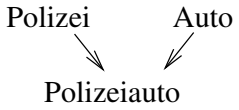
4.4 Schnittstellen

Unter der **Schnittstelle einer Klasse** verstehen wir die Menge aller anwendbaren Merkmale dieser Klasse. Falls die Schnittstelle keine Attribute enthält, entspricht dies einem ADT, da nur die Operationen sichtbar sind. Die Vererbung entspricht dann der Erweiterung von Schnittstellen.

Mehrfachvererbung bedeutet, dass aus mehreren Oberklassen vererbt wird. Dies ist manchmal dadurch motiviert, dass man Funktionalität aus verschiedenen vorhandenen Klassen gemeinsam nutzen will. Hierbei können allerdings einige Probleme auftreten:

- Bei namensgleichen Merkmalen, die direkt in Oberklassen oder deren Oberklassen definiert sind, entstehen Vererbungskonflikte, d.h. es ist dann in der Unterklassen nicht klar, welches Merkmal welcher Oberklasse gemeint ist.
- Mehrfachvererbung kann leicht zu einer falschen Problemmodellierung führen:

  Dies ist eine falsche Modellierung, denn ein Polizeiauto ist keine Polizei, sondern nur ein Auto.

Um mit diesen Problemen umzugehen, werden in objektorientierten Programmiersprachen unterschiedliche Ansätze verfolgt:

- Smalltalk: Hier ist nur Einfachvererbung erlaubt, sodass die oben genannten Probleme nicht auftreten können.
- Java: Hier ist bei Klassen auch nur Einfachvererbung erlaubt. Bei Schnittstellen (s.u.) ist dagegen Mehrfachvererbung zugelassen, was aber unproblematisch ist, da bei Schnittstellen keine Funktionalität vererbt werden kann.
- C++, Eiffel: Hier ist Mehrfachvererbung erlaubt, allerdings müssen mögliche Konflikte explizit gelöst werden.
- Python: Hier ist ebenfalls Mehrfachvererbung erlaubt, wobei die Reihenfolge der angegebenen Oberklassen verwendet wird, um mögliche Konflikte aufzulösen.

4.4.1 Abstrakte Klassen

Wichtig ist bei einem guten Entwurf, logisch zusammengehörige Teile zusammenzufassen und diese möglichst wiederzuverwenden. Häufig trifft man dabei auf folgende Situation: Verschiedene Module/Klassen haben gemeinsame Teile, sind aber nicht Untertyp einer gemeinsamen realen Oberklasse. Hier kann man sich durch das Entwurfsprinzip der **Faktorisierung** behelfen: Verschiebe gemeinsame Anteile in eine gemeinsame Oberklasse. Falls diese Oberklasse keine Instanzen hat, spricht man von einer **abstrakten Klasse**. Damit ist die Charakteristik abstrakter Klassen, dass einige Merkmale zwar bekannt sind, andere Merkmale aber in Unterklassen definiert werden.

Beispiel 4.12 (Abstrakte Klasse)**.**

Betrachten wir eine Klasse zur Durchführung von Benchmarks. Die betrachtete Klasse `Benchmark` ist abstrakt, da ein konkreter Benchmark noch unbekannt ist. Wir können in dieser Klasse aber schon die allgemeine Funktionalität der wiederholten Durchführung von Benchmarks implementieren.

```
abstract class Benchmark {   // abstract: enthaelt mind. eine
                             //           abstrakte Methode
  abstract void benchmark(); // abstract: der Rumpf ist unbekannt

  public long repeat(int count) {
    long start = System.currentTimeMillis();
    for (int i = 0; i < count; i++) benchmark ();
    return (System.currentTimeMillis() - start);
  }
}
```

Beachte: Der Aufruf „`new Benchmark()`“ ist nicht erlaubt! Als Beispiel betrachten wir nun einen konkreten Benchmark zur Zeitmessung für Methodenaufrufe:

```
class MethodBenchmark extends Benchmark {
  void benchmark() { } // leerer Rumpf, d.h. nur Aufruf wird gemessen

  public static void main(String[] args) {
    int count = Integer.parseInt(args[0]);
    long time = new MethodBenchmark().repeat(count);
    System.out.println("msecs: " + time);
  }
}
```

Damit ergeben sich die folgenden Merkmale abstrakter Klassen:

- Einige Teile der Implementierung sind bekannt, andere nicht.
- Es existieren keine Objekte dieser Klasse.
- Falls alle Implementierungsteile unbekannt sind, aber die Merkmalsnamen/-typen festgelegt sind, dann entspricht dies einem Interface in Java, was wir als Nächstes erläutern.

4.4.2 Interfaces in Java

Interfaces sind eine Technik zur besseren Abstraktion/Modularisierung bei der Programmierung. Außerdem erlauben sie die ansonsten nicht unterstützte Mehrfachvererbung in Java.

Konzeptuell enspricht ein **Interface** einer abstrakten Klasse, die nur abstrakte Methoden (und Konstanten) enthält. Jede Klasse kann *beliebig viele* Interfaces implementieren, d. h. sie muss für jede Interface-Methode eine Methode gleichen Namens und Typs enthalten.

Beispiel 4.13 (Interface für Tabellen)**.**

```
interface Lookup {  // statt abstract class
  Object lookup(String name);
}
interface Insert {
  void insert(String name, Object value);
}
```

Interfaces können wie abstrakte Klassen benutzt werden:

```
void processValues (String[] names, Lookup table) {
    // table ist Objekt einer Klasse, die Lookup implementiert
    ...
    table.lookup(...);
    ...
}
```

Konkrete Implementierung einer Tabelle:

```
class MyTable implements Lookup, Insert {
  private String[] names;
  private Object[] values;
  ... // Implementierung von lookup() + insert()
}
```

Da Interfaces nichts implementieren, werden die Konflikte, die bei Mehrfachvererbung auftreten können, vermieden (die ansonsten, wie z.B. in Eiffel oder C++, explizit aufgelöst werden müssen).

4.5 Generizität

Wir haben gesehen, dass Klassen und Module ADTs entsprechen. In Kapitel 2.2.6 haben wir als Verallgemeinerung auch *parametrisierte ADTs* kennengelernt. Damit stellt sich die Frage, ob es auch ein ähnliches Konzept für parametrisierte Klassen oder Module in Programmiersprachen gibt. Tatsächlich sind entsprechende Konzepte zur **Generizität** in verschiedenen Programmiersprachen entwickelt worden.

Als einfachsten Ansatz zur Generizität in OO-Programmiersprachen kann man die Tatsache ausnutzen, dass jede Klasse ein Untertyp von `Object` ist und man dadurch Typparameter als Typ `Object` modellieren kann. Betrachten wir dazu Listen mit beliebigen

Objekten als Elemente:

```
class List {
  Object elem;
  List next;
}
```

Da der Elementtyp `Object` ist, können die Listen beliebige Objekte als Elemente enthalten. Der Nachteil ist allerdings, dass damit keine *uniformen Listen* (z. B. `Integer`- oder `Character`-Listen, etc.) garantiert werden können, wodurch es leicht zu Programmierfehlern kommen kann. Um dies zu vermeiden, gibt es in existierenden Programmiersprachen verschiedene Ansätze, die, ähnlich wie parametrisierte ADTs, explizite Typparameter verwenden.

- In Ada und Modula-3 gibt es Module mit Parametern.

  ```
  GENERIC MODULE List(Elem)
    ⋮
    ... Elem.type ...
    ⋮

  MODULE IntList = List(IntegerElement)
  END IntList;
  ```

- Eiffel erlaubt Klassen mit Parametern

  ```
  class List[T] ...
  ```

  wobei `T` ein Typparameter ist.

- Java hat ab Version 5 eine Erweiterung um Generizität (**generics**).

  ```
  class List<etype> {
      etype elem;
      List<etype> next;
  }
  ```

  Bei der Instanzbildung muss der konkreten Typ angegeben werden:

  ```
  List<String> names;
  ```

- Funktionale Programmiersprachen wie Haskell oder SML erlauben parametrischen Polymorphismus bei Datentypen und Funktionen (vgl. Kapitel 5).

4.6 Pakete (packages)

Wie wir gesehen haben, können logisch zusammengehörige Teile zu Modulen oder Klassen zusammengefasst werden. Bei größeren Anwendungen benötigt man für einen Anwendungsbereich eventuell mehrere Module und Klassen, die auch logisch zusammengehören und die man daher als Einheit kenntlich machen will. Zu diesem Zweck werden in Programmiersprachen oft weitere Strukturierungsmechanismen angeboten. Z.B. werden in Java Pakete zur Zusammenfassung von Klassen zu größeren Einheiten verwendet.

- Konzeptional gehört jede Klasse zu einem Paket, welches zu Beginn der Programmdatei angegeben wird.

  ```
  package mh.games.tetris;
  ```

- Jede Klasse eines Pakets ist ansprechbar über ihren Paketnamen (womit Namenskonflikten zwischen Klassen in unterschiedlichen Paketen vermieden werden).

  ```
  java.util.Date now = new java.util.Date();
  ```

  Hierbei ist `java.util` der Paketname und `Date` der Name der Klasse in diesem Paket.

- Durch den Import einzelner Klassen oder kompletter Pakete können die Klasse oder die im Paket enthaltenen Klassen sichtbar gemacht werden.

  ```
  import java.util.Date; // oder java.util.*: alle Klassen
  ⋮
  Date now = new Date();
  ```

- In der Java API sind viele Pakete vorhanden, z. B.
  - `java.io`: Ein/Ausgabe
  - `java.util`: viele Hilfsklassen, z.B. Datenstrukturen
  - `java.awt`: Graphik, Fenster etc.
  - `java.net`: Netzwerkprogrammierung
  - `java.rmi`: verteilte Programmierung mittels Remote Method Invocation