Kapitel 8

Generische Klassen

- Ziel: Zusammensetzen von Software-Bausteinen

  Oft probiert, nie erreicht! sprachliche Mittel fehlten

- In C++, ADA, Eiffel, Java: Klassen, die mit anderen Klassen parametrisiert sind:

```
class C<T>{ // generische Klasse
  T x;
  void f(T y){...}
  ...
}

C<Student> p; Student s;
C<Hiwi> q; Hiwi h;
p.f(s); // OK
p.f(h); // Typfehler
```

- Eine generische Klasse kann man als Funktor ($\rightsquigarrow$ Kategorientheorie) auffassen: sie bildet einen Typ (i.e. Klasse) in einen Typ ab.

$$C : \mathcal{TYP} \to \mathcal{TYP};\ \ T \mapsto C\langle T \rangle$$

- Eiffel/C++: Parameterklasse kann irgendwas sein
  $\Rightarrow$ generische Definition nicht isoliert typcheckbar, sondern nur konkrete Instanzen
  $\Rightarrow$ Bibliotheken nicht statisch typprüfbar
- Java et al: *Bounded Polymorphism*: Parameter muß Unterklasse einer gewissen Klasse sein

  $\Rightarrow$ Verfügbarkeit von Methoden kann auch in generischen Klassen statisch geprüft werden

$\Rightarrow$ Klassen/Module höherer Ordnung („Funktoren“)

$\Rightarrow$ Wiederverwendung, Allgemeinheit, Bibliotheken, Komponentenretrieval werden enorm verbessert!

8.1 Generische Klassen in Java

- Bis Java 1.4. mussten generische Container und polymorphe Funktionen mittels Elementen vom Typ `Object` und Downcasts realisiert werden, was sehr fehleranfällig ist:

```
class Stack {
        Object[] elements;
        void push(Object x) { ... };
        Object top() {...}
}

Object id(Object x) { return x;}

Point q = (Point) id (new Point()); // OK
Point q = (Point) id (new Student());
  // statisch korrekt,
  // aber illegaler Downcast zur Laufzeit
Point q = (Point) top(push(new Student()));
  // dito
```

- in Java 1.5. bzw „Java 5“: *Typschranken* für Typparameter generischer Klassen/Interfaces erlauben Typprüfung von isolierten Klassendefinitionen

  Objekte mit generischem Typ dürfen Methoden der Typschranke verwenden

  NB: keine Typschranke = Typschranke `Object`

- Beispiele:

```
class NumberPair<X extends Number, Y extends Number> {
  // Typschranke=Klasse
  X a; Y b;
  public X first(NumberPair<X,Y> p){return p.a;}
}
NumberPair<Integer,Float> np =
  new NumberPair<Integer,Float> ();

interface Comparable<T> {
  int compareTo(T o)
}

class Time implements Comparable<Time> {
  int compareTo(Time o) {...}}
  // Verwendung generischer Interfaces

class Pair<X extends Comparable, Y extends Comparable>
      implements Comparable<Pair<X,Y>> {
  // Typschranke = generisches Interface
  X a; Y b;
  int compareTo (Pair<X,Y> other) {
  ... a.compareTo(other.a) ...
  ... b.compareTo(other.b); ...}
}

class NP2 <X extends Number & Comparable,
           Y extends Number & Comparable>
           implements Comparable<NP2<X,Y>> { ...}
  // mehrere Typschranken
```

- *generische Methoden*: Typparameter steht vor Methodensignatur, zB

```
static <U extends T>  U foo(U x) {...}

class UT extends T {...}
T x = foo(new T()); // OK
UT y = foo(new UT()); // OK
```

  Diese Flexibilität kriegt man auch durch Kontravarianz ($\rightsquigarrow$ Typsysteme) nicht hin

- *Implementierung generischer Klassen:* da die JVM nicht geändert wurde, setzt der Compiler nach der Typprüfung für alle Typparameter `Object` ein und ersetzt generische Klassen durch ihren nichtgenerischen „Rohtyp“. Illegale Downcasts sind nun aber ausgeschlossen!

- Es gibt Refaktorisierungstools (zB von Frank Tip für Eclipse), die automatisch Programme ohne Generizität in Programme mit generischen Klassen umsetzen

8.2 Wildcards

Kann man zwischen generischen Klassen oder Instanzen eine Vererbungsbeziehung definieren? Bsp:

```
class OrderedList<Data> extends List<Data> {...}  // OK
class OrderedPair<X extends Comparable, Y extends
  Comparable>
      extends Pair<X,Y> {...}   // OK

class Amsel extends Vogel {...}
class Strauss extends Vogel {...}
class Käfig<Art extends Vogel> { Art insasse; ...}

void f(Käfig<Vogel> k) {...}

f(new Käfig<Vogel>);  -- OK
f(new Käfig<Amsel>);  -- TYPFEHLER
```

- `Käfig<Amsel>` *nicht* Unterklasse von `Käfig<Vogel>`!
- Generell kann es keine Unterklassenbeziehung zwischen generischen Klassen geben, wenn diese zuweisbare Unterkomponenten/Members des Typparameters enthalten (wie `insasse`)

  NB auch zwischen Arrays kann es eigentlich keine Unterklassenbeziehung geben, obwohl Java dies zulässt ($\rightsquigarrow$ „Array-Anomalie“)

  Begründung für beides im Kapitel Typsysteme
- Um diese Einschränkung zu umgehen, gibt es „Wildcards“

  de facto *anonyme Typparameter*

Bsp:

```
class Käfig<? extends Vogel> {...}

f(new Käfig<Vogel>);  -- OK
f(new Käfig<Amsel>);  -- OK!
```

- Unterklassenbeziehung `Käfig<Amsel>` zu `Käfig<Vogel>` funktioniert, da man in `Käfig` kein zuweisbares Member vom Typ `?` deklarieren kann.

  Trotz dieser Einschränkung sind Wildcards nützlich!

- Beispiele: `Collections.copy/binarySearch`

```
static <T> void copy(List<? super T> dst, List<?
  extends T> src )


static <T extends Comparable<? super T>> int
  binarySearch(List<? extends T> lst, T k) )
```

  Achtung: mehrere `?` stehen für verschiedene anonyme Parametertypen

  `? super T` gibt eine *untere Typschranke* an

- Diese Formulierungen sind flexibler als

```
static<T extends Comparable<T>> int binarySearch(List<
  T> lst, T K)

static <U> void copy(List<? super U> dst, List<U> src)
```

  da man nicht zB in einer Liste von Studenten nach einer Person suchen kann. Dies geht hingegen in der obe-

ren Formulierung (wenn `Person` `Comparable` implementiert). Es ginge sogar, wenn nur `Lebewesen` `Comparable` implementiert.

Beim Kopieren kann die obere Formulierung eine Liste von Studenten (elementweise) in eine Liste von Lebewesen kopieren, die untere kann nur eine Liste von Personen in eine Liste von Lebewesen kopieren; ohne `super` müssten beide Listen exakt den gleichen Typ haben

8.3 Generische Programmierung in C++

weiteres Beispiel: MVC in C++ mit Templates und Function Pointern ⇝ Kapitel 10.4