Eleganz, Klarheit, Präzision durch funktionale Programmierung mit Java 5

Die Bürger erster Klasse

VON CHRISTIAN NEUMANN

In funktionalen Programmiersprachen sind Funktionen „first class citizens“, d.h., Funktionen sind ganz normale Variablen. Funktionen höherer Ordnung akzeptieren Funktionen als Argumente und/oder produzieren Ergebnisse, die selbst wieder Funktionen sein können. Deswegen sind funktionale Programme oft kürzer und abstrakter als ihr Äquivalent in imperativen/objektorientierten Programmiersprachen.

Da liegt die Idee nahe, dass in Java jede Funktion durch eine Klasse implementiert wird, und zwar so, dass jedes Objekt der Klasse diese Funktion repräsentiert. Ein solches Funktionsobjekt übernimmt die Rolle eines Funktionszeigers in C (*void (*fp) (void)*), eines Delegate in C# oder einer Funktionsvariablen in Python. Funktionsobjekte sind nichts anderes als die Benutzung des GoF Command-Musters: Funktionen als ganz normale Objekte mit der einzigen Methode *execute*.

Java-Schnittstellen sind ein wichtiges Hilfsmittel, damit man annähernd funktional mit Java programmieren kann. Die neue Java-Version bietet die Möglichkeit, generische Schnittstellen zu definieren und diese typsicher zu implementieren. Generics gewährleisten einerseits Typsicherheit zur Compile-Zeit, nicht erst zur Laufzeit, und vermeiden andererseits die zahlreichen mit der generischen Java-Programmierung verbundenen Cast-Methoden.

Wir befassen uns mit der Frage: Welchen Mehrwert hat der funktionale Programmierstil? Dazu stellen wir einige Beispiele für den Einsatz von funktionalen Hilfsmethoden vor und zeigen wiederkehrende Aufgaben, die sich damit beson-

ders gut lösen lassen. Das Zusammenspiel von funktionalen Programmteilen und der Objektorientierung kann uns auch die tägliche Arbeit erleichtern. Unsere Überlegungen zeigen, dass die funktionale Programmierung mit Java 5 kein theoretischer Ballast ist, sondern dass wir damit einen weiteren Schritt in Richtung Wiederverwendung, Änderbarkeit und Softwarequalität gehen.

Die Sprachdesigner haben bei der Entwicklung von Java 5 ein Hauptaugenmerk auf die Qualität der Programmiersprache gelegt: Die Programmierung soll einfacher, die Performance verbessert werden. Funktionale Programmteile sind neben der Programmiersprache, dem objektorientierten Programmierstil und dem Denken in Komponenten und Schnittstellen ein weiterer Ansatz, um qualitativ hochwertige Software zu bauen. Dazu benutzen wir in Java 5 generische Schnittstellen, die die Außensicht auf typsichere Prädikate und Funktionen definieren. Für die Innensicht sind die Benutzer selbst verantwortlich, jeder Benutzer legt einen konkreten Typ und die Implementierung fest. Gewissermaßen als Nebeneffekt zeigt der Artikel einige Beispiele, wo die viel kritisierten Generics zu einfachen und sprechendem, aber vor allem auch zu eleganterem Programmcode führen.

Objekte und Funktionen

Funktionsobjekte besitzen die einzige Methode *execute*. Je nach Verwendung unterscheidet sich diese nur in der Anzahl der Parameter und/oder des Rückgabetyps. Prädikate sind für eine Bedingung immer *wahr* oder *falsch*. Prädikate gibt es als unäre Prädikate (mit einem Parameter), als binäre Prädikate (mit zwei Parametern) und als Prädikate mit variablen Argumentlisten. Unäre und binäre Funktionen (*UnaryFunction*, *BinaryFunction*) bzw. Funktionen mit beliebiger Anzahl von Parametern (*NaryFunction*) liefern dagegen jeweils ein Objekt zurück. Listing 1 ist ein Beispiel für die unäre Funktion *MultiplyBy* sowie für das Zusammenspiel der Schnittstelle *UnaryFunction* und der funktionalen Hilfsmethode *map* in der alten Java-Version.

Aber es ist nicht immer so einfach: Die generische Schnittstelle *UnaryFunction* lässt sich in Java 1.4 und Listing 1 vom

Funktionale Programmierung – die Idee

Es gibt keine allgemein anerkannte Definition, was zu einer funktionalen Sprache gehört und was nicht. In diesem Artikel bezeichnen wir mit der funktionalen Programmierung einen Programmierstil, der den Schwerpunkt auf die Auswertung von Ausdrücken und Funktionen legt – und nicht auf die Ausführung von Befehlen. Funktionale Programme bestehen üblicherweise aus Funktionsdefinitionen, Funktionsanwendungen und Funktionskompositionen.
Die Vorzüge der funktionalen Programmierung lassen sich am Beispiel der Summe von *n* Zahlen besonders gut darstellen. Einen sinnvollen Einsatz zeigen wir im Artikel am Beispiel eines Warenkorbs.
In einer reinen funktionalen Programmiersprache (wie Haskell [9]) kann das Ergebnis durch die Auswertung dieses Ausdrucks berechnet werden:

```
sum [20, 10, 5, 99, 28]
```

In imperativen Sprachen, wie C oder Java, drücken wir diese Funktion als Schleife aus und erhöhen bei jedem Durchlauf den Schleifenzähler und die Summe.

```
int sum(int[] a) {
    int result = 0;
    for (int i = 0; i < a.length; i++) {
      result += a[i];
    }
    return result;
  }
// Execution
sum(anyArray);
```

Funktionale Programmierung vereinheitlicht auch die lästigen Schleifen über Arrays/Behälter. Wir müssen nicht mehr überlegen, ob der Index bei 0 oder 1 anfängt und bis kleiner oder kleiner/gleich der Länge des Arrays läuft.
Das Denken in einfachen mathematischen Begriffen lässt uns das Beste von funktionaler Programmierung auch in Java nutzen; funktionale Programmteile sind nahezu optimal wiederverwendbar – damit verbessert sich die Softwarequalität.

Benutzer nicht auf eine Menge von Typen einschränken. Sie ist zwar generisch, nicht aber typsicher, denn in Java ist jedes Objekt von der Basisklasse *Object* abgeleitet. Es kann niemand garantieren, dass nur *Integer*-Objekte in der Liste sind.

Typsichere Prädikate und Funktionen

Durch die Benutzung der Java Generics meldet schon der Compiler Konflikte bei den Prädikaten oder Funktionen, nicht erst die Laufzeitumgebung. Java Generics sind aber auch die Ausnahme gegenüber der Leichtigkeit des Seins. Sie enthalten Komplexität, die zu verstehen selten Spaß macht und immer Aufwand verursacht [1]. Anders formuliert: Typsicherheit bei Behältern mit Generics führt manchmal auch zu Sackgassen und Irrwegen [2]. Dass das nicht immer so sein muss, zeigen unsere Beispiele: Generics und funktionale Programmierung führen zu Programmcode mit den Eigenschaften Eleganz, Klarheit, Präzision. Wir betrachten als Beispiel die Transformation einer Liste. Der Programmcode in Listing 2 ist einfach zu verstehen: Es geht um eine neue Liste mit den *String*-Repräsentationen aller *Customer*-Objekte der Ursprungsliste. Was man dazu braucht, ist die unäre Funktion *ToString* (Bedenke: *toString()* wird nur auf einem Objekt aufgerufen). Des Weiteren benutzt der Anwender die Implementierung der funktionalen Hilfsmethode *map*, die auf jedes Element der Ursprungsliste die unäre Funktion *ToString* ausführt. *ToString* funktioniert für alle Objekttypen; der Rückgabetyp ist immer ein String – das ist durch die Implementierung der Schnittstelle festgelegt. Hier zeigt sich ein Vorteil der Generics: *new ToString<Customer>()* ist so prägnant, dass jeder Softwareingenieur sieht, dass es sich um eine *String*-Repräsentation von *Customer*-Objekten handelt. *ToString* benutzt in Listing 2 die Standardmethode *toString()*; es ist aber auch denkbar, dass in dieser unären Funktion eine andere *String*-Repräsentation implementiert wird.

> Generics und funktionale Programmierung führen zu Code mit den Eigenschaften Eleganz, Klarheit, Präzision.

Argumente für diesen Programmierstil

Zuständigkeiten sollen möglichst gut getrennt sein [3] – das ist einer der Leitgedanken dieses Artikels. In Listing 2 zeigen wir die Vorteile der funktionalen Programmierung mit Java 5 und warum dieses Beispiel zur Trennung von Zuständigkeiten passt; hier die Details:

1. Die unäre Funktion *ToString* ist bereits zur Compile-Zeit typsicher: *ToString* ist nur ein einziges Mal zu implementieren und wiederverwendbar, Änderungen an *ToString* sind nur an einer Stelle durchzuführen.
2. Für die funktionale Hilfsmethode *map* gilt Regel 1 entsprechend.

Listing 1

Funktionale Programmierung mit Java 1.4

```
public interface UnaryFunction {
  Object execute(Object object);
}

public abstract class FP {

  // Applying a function to each element of a list
  public static List map(UnaryFunction function,
                                                List list) {
    List result = new ArrayList(list.size());

    for (Iterator iterator = list.iterator(); iterator.
                                            hasNext();) {
      result.add(function.execute(iterator.next()));
    }

    return result;
  }
}

// Practice, Test
List numbers = ... // fill with objects of type integer

List result = FP.map(new MultiplyBy(2), numbers);
// Result is a new list in which each element is
// multiplied by 2
```

Listing 2

Transformation einer Liste

```
// 0. The interface is given
public interface UnaryFunction<R, T> {

  R execute(T t);
}

// 1. Reusable UnaryFunction with the action
public class ToString<T> implements
                                UnaryFunction<String, T> {

  public String execute(T t) {
    return t.toString();
  }
}

// 2. Reusable implementation of the map-method
public static <R, T> List<R> map(UnaryFunction<R, T>
                                function, List<T> list) {
  List<R> result = new ArrayList<R>(list.size());

  for (T t : list) {
    result.add(function.execute(t));
  }

  return result;
}

// 3. Practice, Test
List<Customer> customers = ... // fill with objects of
                                // type customer

List<String> result = FP.map(new
                ToString<Customer>(), customers);
// result contains all customers as string representation
```

Listing 3

Transformation einer Liste

```
// 0. Uses the same interface UnaryFunction

// 1. Reusable UnaryFunction with the action
public class ToCustomer<T extends String>
                implements UnaryFunction<Customer, T> {

  public Customer execute(T number) {
    return new Customer(number);
  }
}

// 2. Reusable implementation of the map-method
// Uses the same map-method

// 3. Practice, Test
List<String> customerNumbers = ... // fill with
                                //objects of type string

List<Customer> list = FP.map(new
    ToCustomer<String>(), customerNumbers);
// result contains all customerNumbers as
// customer objects
```

3. Den offensichtlichen Vorteil hat der Benutzer von (1) und (2). Er muss sich keine Gedanken um die Implementierung machen; in nur einer Programmzeile bekommt er das Ergebnis der funktionalen Hilfsmethode. Der Programmcode ist kürzer und klarer (nur eine sprechende Programmzeile!). Jeder Softwareingenieur kann *map* benutzen. Diese Methodenaufrufe verteilen sich zwar über den globalen Zustandsraum der Softwareanwendung, nicht aber die Implementierung – die gibt es nur an einer einzigen Stelle. *map* ist fehlerfrei und steht der ganzen Java-Welt zur Verfügung.

Eine Transformation einer Liste von Strings in eine Liste von Customers funktioniert – unserer Forderung entsprechend – wie in Listing 3. Hier ist nur die Implementierung der unären Funktion *ToCustomer* zu tun.

Funktionale Programmierung mit Java 5 beschränkt sich nicht nur auf Listen und Maps. Prädikate können z.B. bei der Datenbank-Programmierung verwendet werden: *IsAvailable* überprüft eine Ergebnismenge (z.B. ein *ResultSet*) und liefert im Zusammenspiel mit der funktionalen Hilfsmethode *filter* die Teilmenge zurück, die genau diesem Prädikat entspricht. Der Aufruf wäre schlicht: *FP.filter(new IsAvailable<RentalCar>(), cars)*.

Listing 4 zeigt eine weitere *map*-Implementierung, die elegant ist. Die transformierten Elemente werden nicht in eine neue Liste kopiert; *map* speichert lediglich die Vorschrift der Berechnung und führt die *get*-Methode *lazy* aus. Wir benutzen dazu das GoF Template-Muster. Dieses Muster eignet sich vor allem für schnelle Operationen auf Listen. Einzige Einschränkung: Die neue Liste hat eine feste Größe; der Benutzer kann keine weiteren Elemente hinzufügen. In Listing 4 ist aber auch Vorsicht angebracht. Man sieht erst auf den zweiten Blick, dass diese Implementierung nur eine Sicht auf die Ursprungsliste ist – und das macht Seiteneffekte möglich. In reiner funktionaler Programmierung sind keine Seiteneffekte erlaubt; das sollte man bei den Implementierungen in *FP* berücksichtigen.

Best Practice

Wir plädieren dafür, funktionale Programmierung mit sehr gut definierten Schnittstellen zu betreiben und dann in einer abstrak-

Listing 4

map-Implementierung mit Seiteneffekt

```
public static <R, T> List<R> map(final UnaryFunction
                        <R, T> function, final List<T> list) {
  return new AbstractList() {

    public R get(int index) {
      return function.execute(list.get(index));
    }

    public int size() {
      return list.size();
    }
  };
}
```

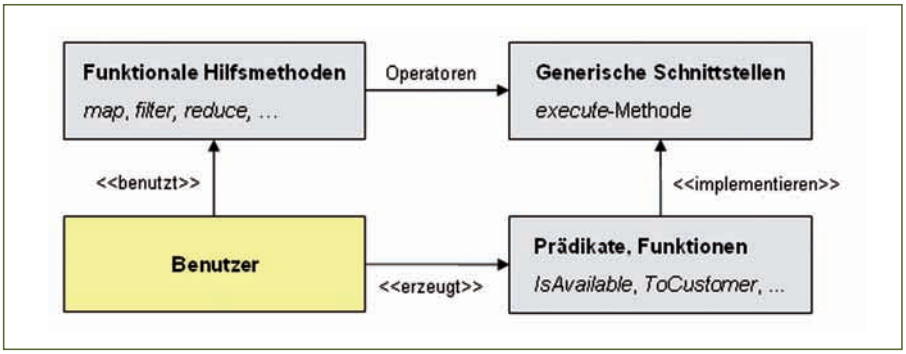

Abb. 1: Funktionale Programmierung mit Java 5

ten Klasse *FP* mit statischen Hilfsmethoden von ihren Vorzügen zu profitieren.

Besonderes Augenmerk gilt dabei den Abhängigkeiten zwischen Funktionsobjekten: Diese kommunizieren nur und ausschließlich über Schnittstellen. Das hört sich an wie eine Selbstverständlichkeit, definiert aber gleichzeitig auch das Verfahren, wie man innerhalb der Objektorientierung die Qualität des Programmcodes verbessert. Hier drei Regeln, wie man sehr schnell funktionale Programmteile schreibt und integriert:

1. Prädikate und Funktionen sind voneinander unabhängig. Sie werden gegen eine generische Schnittstelle implementiert. Jede Schnittstelle definiert die Methodensignatur und den Rückgabetyp der *execute*-Methode. Jedem Experten ist es freigestellt, ein eigenes Prädikat oder eine Funktion zu schreiben. Diese Implementierung ist als Erstes zu tun. Die Schnittstellen sind gegeben.
2. Funktionale Hilfsmethoden (wie *map*) befinden sich in einer abstrakten Klasse *FP*. Ist die passende Methode noch nicht verfügbar, dann kann diese Klasse erweitert werden. Die neue Methode steht dann der ganzen Softwareanwendung zur Verfügung. *FP* ist gut dokumentiert (wie das JDK). Mit den neuen *static imports* kann man diese Hilfsmethoden noch einfacher benutzen.
3. Die Anwendung (z.B. ein JUnit-Testtreiber) bringt (1) und (2) zusammen – das ist nur eine Programmzeile, und zwar der Aufruf einer funktionalen Hilfsmethode und die Übergabe eines Prädikats/Funktion sowie eine Ursprungsliste/-Array.

Listing 5

Generische Schnittstellen

```
public interface UnaryPredicate<T> {
  boolean execute(T t);
}

public interface UnaryFunction<R, T> {
  R execute(T t);
}

public interface BinaryPredicate<T> {
  boolean execute(T t1, T t2);
}

public interface BinaryFunction<R, T> {
  R execute(T t1, T t2);
}

public interface NaryPredicate<T> {
  boolean execute(T... t);
  boolean execute(List<T> list);
}

public interface NaryFunction<R, T> {
  R execute(T... t);
  R execute(List<T> list);
}
```

Bei diesem Vorgehen sind zwei Probleme denkbar: Einerseits das Tangling (Vermischung von Zuständigkeiten) und andererseits das Scattering (Verteilung von Zuständigkeiten). Beides sind Standardprobleme der Objektorientierung, die man auch bei der funktionalen Programmierung mit Java 5 diskutieren kann. Vermischung von funktionalen Anforderungen und funktionalen Hilfsmethoden kommt nicht in Frage, denn: In der Methode, in der die funktionale Hilfsmethode aufgerufen wird, steht ohne der funktionalen Programmierung noch mehr Programmcode. Mehr Programmcode – das ist die Implementierung von *map*, also mehrere Programmzeilen, die für jedes Element einer Liste eine Funktion ausführen.

Normalerweise ist die gleiche Funktion an *n* Stellen in der Softwareanwendung implementiert (als Beispiel sei *n* = 1000); eine potenzielle Fehlerquelle die sich über das komplette Softwareprojekt verteilt! Wir implementieren diese funktionale Hilfsmethode an nur einer Stelle, ähnlich einiger Hilfsmethoden im JDK, also Operationen auf *Collections*, *Arrays* oder *Character*. Die statische Methode *min* der *Collections*-Klasse ermittelt beispielsweise das kleinste Element einer Liste – der Algorithmus ist fehlerfrei. Darüber herrscht Konsens.

Das zweite Problem, die Verteilung von Zuständigkeiten existiert auch nicht: Egal ob man die funktionalen Hilfsmethoden, Prädikate oder Funktionen betrachtet, alle diese Implementierungen befinden sich an einer einzigen Stelle. Änderungen sind nur an dieser Stelle durchzuführen und nicht über die Softwareanwendung verteilt. Die Signaturen der funktionalen Hilfsmethoden in *FP* ändern sich nicht. Es ist kein Problem, wenn sich diese Methodenaufrufe über den globalen Zustandsraum der Softwareanwendung verteilen.

Der wichtigste Gedanke hinter dieser Ebene ist die Einfachheit. Funktionale Programmierung macht qualitativ hochwertigen Programmcode; *map* ist fehlerfrei, erzeugt keine Seiteneffekte und steht der ganzen Java-Welt zur Verfügung. Warum gibt es eine abstrakte Klasse *FP* mit funktionalen Hilfsmethoden im JDK noch nicht?

Prinzip der minimalen Annahme

Damit der Benutzer Prädikate und Funktionen überhaupt implementieren kann, muss für jedes Prädikat und für jede Funktion eine passende und generische Schnittstelle definiert werden sowie eine oder mehrere Klassen, die diese Schnittstelle exportieren. Das Prinzip der minimalen Annahme enthält zwei Regeln:

1. Mache in der definierten Schnittstelle keine unnötige Annahme über Parameter und Rückgabewerte, es sei

denn, es ist ausdrücklich gewünscht. Schnittstellen zur funktionalen Programmierung mit Java 5 sind immer generisch. Dazu genügt die Verwendung des Typparameters *T* (auch Typ-Variable). Die *execute*-Methode ist mit *T* parametrisiert. Dieses *T* ist ein Platzhalter für alle Java-Klassen. Erst der Benutzer entscheidet sich für einen bestimmten Typ; *T* wird durch diesen Typ ersetzt.

2. Wähle unter allen passenden Schnittstellen die einfachste (z.B. *UnaryFunction* statt *NaryFunction*, *UnaryPredicate* statt *NaryPredicate*, wenn die Methoden der einfacheren Schnittstelle genügen). Die Parametertypen sind so allgemein wie möglich und so speziell wie nötig. Beispiel: *List<T>* statt *Collection<T>* oder *Iterable<T>*, falls die Reihenfolge der Elemente eine Rolle spielt.

Hier noch eine Ergänzung: Die Verwendung von variablen Argumentlisten (*T...*) anstatt Arrays (*T[]*) minimiert die Annahme über die Anzahl der Parameter. Intern ist eine variable Argumentliste als Array organisiert.

Die Verwendung von Varargs gesteht dem Benutzer jedoch zwei Möglichkeiten zu: Entweder er übergibt alle Parameter einzeln oder er verpackt sie zusammen in einem Array. Mit variablen Argumentlisten können wir schließlich auch einen optionalen Parameter in einer Methodensignatur (wie bei *reduce* in Listing 7) realisieren.

Generische Schnittstellen

In Java 5 lassen sich generische Schnittstellen vom Benutzer einschränken. Er entscheidet, für welche Menge von Typen er die Schnittstellen benutzen will. Für diese Menge gewährleistet Java 5 Typsicherheit schon zur Compile-Zeit, nicht erst zur Laufzeit. In Listing 5 zeigen wir sechs Vorschläge für Schnittstellen zur funktionalen Programmierung mit Java 5.

Zum Typparameter *T* ist der Typparameter *R* des Rückgabetyps hinzugekommen. *R* wird genauso wie *T* vom Benutzer eingeschränkt und definiert. Auch für *R* gilt Typsicherheit zur Compile-Zeit. Die Schnittstellen der Prädikate sind so definiert, dass sie jeweils den primitiven booleschen Wert zurückliefern. Hier könnte auch das Objekt *Boolean* stehen, dann wäre aber oft ein unnötiges Autoboxing des primitiven Datentyps zur Objektrepräsentation notwendig. Neue Objekte zu erzeugen geht zu Las-

Listing 6

Funktionen höherer Ordnung

```
public static <R, T> void forEach(UnaryFunction<R, T> function, List<T> list) {
  // Implementation of forEach
}

public static <R, T> List<R> map(UnaryFunction<R, T> function, List<T> list) {
  // Implementation of map
}

public static <T> List<T> filter(UnaryPredicate<T> predicate, List<T> list) {
  // Implementation of filter
}

public static <T> boolean some(UnaryPredicate<T> predicate, List<T> list) {
  // Implementation of some
}

public static <T> boolean every(UnaryPredicate<T> predicate, List<T> list) {
  // Implementation of every
}

public static <T> T reduce(BinaryFunction<T, T> function, List<T> list, T... initialValue) {
  // Implementation of reduce
}
```

ten der Performance – man verwende in Java 5 primitive Datentypen, wann immer es geht!

Funktionen höherer Ordnung

Wir unterscheiden Funktionen erster Ordnung und Funktionen höherer Ordnung. Funktionen höherer Ordnung akzeptieren andere Funktionen als Argumente und/oder produzieren Ergebnisse, die selbst wieder Funktionen sein können. Einige Programmiersprachen unterstützen funktionale Konzepte sehr gut (wie Python [4]) und bringen eine ganze Reihe solcher Funktionen mit. Listing 6 zeigt sechs Vorschläge für Funktionen höherer Ordnung. Alle diese Funktionen sind in der abstrakten Klasse *FP* als statische Methoden implementiert.

Solche Methoden gibt es in zahllosen Variationen, aber hier kommt die Erklärung der Beispiele: *forEach* führt die unäre Funktion für jedes Listenelement aus. Das Ergebnis kann die unäre Funktion in ihrer Implementierung speichern oder auch nicht; *list* ist unverändert. *map* funktioniert wie *forEach*, kopiert aber die Ergebnisse der Funktion in eine neue Liste. *filter* liefert eine neue Liste zurück, in der alle Elemente von *list* sind, für die das unäre Prädikat *wahr* ist. *Some* überprüft, ob mindestens ein Element der Liste die Bedingung des unären Prädikats erfüllt; das Ergebnis von *every* ist *wahr*, wenn diese Bedingung für alle Elemente der Liste gilt. *reduce* wendet die binäre Funktion kumulativ auf alle Elemente der Liste an, sodass sich die Liste zu einem Wert reduziert. *reduce* kommt zum Einsatz, wenn man z.B. die Summe einer Liste von Zahlen summieren will (wie beim Warenkorb in Listing 7).

Der *reduce*-Algorithmus erwartet zwei oder drei Parameter, aber nicht endlich viele. Der Experte für den *reduce*-Algorithmus implementiert die Fehler- und Ausnahmebehandlung, was den Benutzer von *reduce* nicht weiter kümmert. Für weitere Ideen empfehlen wir die Dokumentation einer (echten) funktionalen Programmiersprache.

Design by Contract

Eine weitere Neuerung in Java 5 sind die erwähnten variablen Argumentlisten, welche die Parameterübergabe komfortabler gestalten. Hier das Beispiel von *map* mit einer zweiten Signatur:

```
public static <R, T> List<R> map(UnaryFunction
                                 <R, T> function, T... t) {
  return map(function, Arrays.asList(t));
}
```

Die meisten Methoden in der abstrakten Klasse *FP* haben diese zweite Signatur. Der Benutzer der *map*-Methode hat dann zwei Möglichkeiten: Entweder er übergibt alle Parameter einzeln oder zusammen in einem Array verpackt. *T...* garantiert uns, dass nur Objekte des gleichen Typs übergeben werden. Die Implementierung von *map* ist kein weiteres Mal zu tun, wir benutzen jeweils die statische Methode *asList* der *Arrays*-Klasse.

An dieser Stelle ist die Entscheidung zum Design der Methoden in *FP* bekannt zu machen: Es ist nicht möglich, in der *FP*-Klasse neue generische Arrays zu erzeugen und an den Aufrufer zurückzuliefern (das kann Java 5 nicht). In unserem Vertrag steht: Alle funktionalen Hilfsmethoden, die ein Array oder eine variable Argumentliste als letzten Parameter in der Methodensignatur haben, liefern eine typsichere Liste mit den Ergebnissen der Funktion zurück. Dem Softwareingenieur ist es freigestellt, die neue Liste zu verwenden oder mithilfe *Collections.toArray()* in nur einer weiteren Zeile daraus ein ebenfalls typsicheres Array zu machen (Beispiel: *String[]* oder *Customer[]*).

Zusammenfassung

Die funktionale Programmierung mit Java 5 ergänzt die Objektorientierung mit der „Eleganz, Klarheit, Präzision" der Mathematik. Für Projekte, die diesseits der Machbarkeitsgrenze operieren, ist das Denken in einfachen mathematischen Begriffen zwar noch lange nicht die Garantie für den Projekterfolg, aber immerhin eine weitere Grundlage zur Absicherung der Softwarequalität. Der funktionale Programmierstil ist einerseits gewöhnungsbedürftig und eine weitere Abstraktion zur Objektorientierung, führt aber andererseits zu besserem Programmcode mit weniger Fehleranfälligkeit.

Wir unterscheiden Prädikate, Funktionen und funktionale Hilfsmethoden (wie *map*, *filter*, *reduce*), die die Operationen auf Funktionsobjekte implementieren. Der Programmcode wird kürzer

Listing 7

Beispiel für den reduce-Algorithmus

```
// 1. Reusable BinaryFunction with the action
public class Sum<T extends Euros>
                    implements BinaryFunction<T, T> {

  public Euros execute(Euros first, Euros second) {
    return new Euros(first.getAmount() + second.
                                    getAmount());
  }
}

// 2. Reusable implementation of the map-method
public static <T> T reduce(BinaryFunction<T, T>
    function, List<T> list,  T... initialValue) {
  if (list.isEmpty() && initialValue.length == 0) {
    throw new RuntimeException ("reduce of empty
                            list with no initial value");
  }

  if (initialValue.length > 1) {
    throw new RuntimeException
    ("reduce expected at most 3 arguments, got " +
                            (2 + initialValue.length));
  }

  Iterator<T> iterator = list.iterator();
  T result = initialValue.length > 0 ? initialValue[0] :
                                    iterator.next();

  while (iterator.hasNext()) {
    result = function.execute(result, iterator.next());
  }

  return result;
}

// 3. Practice, Test
List<Euros> shoppingCart = ...
                    // fill with objects of type Euros

// totalAmount without forwarding charges
totalAmount = FP.reduce(new Sum<Euros>(),
                                    shoppingCart);

// totalAmount with forwarding charges
totalAmount = FP.reduce(new Sum<Euros>(),
                    shoppingCart, forwardingCharge);
```

und klarer, der Benutzer kommt in nur einer Programmzeile an Informationen, die er sich normalerweise mühsam zu beschaffen hat (Fehleranfälligkeit, Zeitaufwand). Die gleichen Implementierungen verteilen sich ohne *FP* über den globalen Zustandsraum der Softwareanwendung. Die Wartbarkeit und Änderbarkeit nimmt mit dieser Verteilung stark ab.

Es ist kein Problem, verschiedene Bausteine (Funktionen, Prädikate) zu kombinieren und wieder zu verwenden. Der funktionalen Programmierung sind keine Grenzen gesetzt, egal ob man eine Liste mit einem Prädikat filtert oder in einer Datumsklasse weitere Prüfungen mit Prädikaten einbaut: Funktionale Programmierung mit Java 5 funktioniert in der Praxis – den Nutzen spüren und sehen wir auch ohne wissenschaftliche Messungen!

Viele Programmierer verstehen schon nach kurzer Zeit ihren eigenen Programmcode nicht mehr. Funktionale Programmierung ist auch hier hilfreich, denn der Programmcode ist übersichtlich und sprechend (Beispiel: *while(some(watching, dogs)) wait()*). Die Abstraktion hinter der funktionalen Programmierung müssen nicht alle Softwareingenieure verstehen. Dabei ist aber zu beachten, dass man wenigstens die Benutzung der funktionalen Hilfsmethoden verstanden haben muss. Das ist aber überhaupt kein Problem. Hat man diese Methoden einmal benutzt, dann sind die funktionalen Hilfsmethoden so einfach zu handhaben wie die statischen Hilfsmethoden im JDK (Beispiel: *Arrays.equals()*). Der Softwareingenieur soll es leicht haben, im besten Fall sogar Spaß.

Aber wo Licht ist, ist auch Schatten: Generics können zu weniger gut lesbaren Programmcode führen. Das ist der Preis für Typsicherheit zur Compile-Zeit. Wegen der Abwärtskompatibilität zu Java 1.4 hat der Benutzer die Möglichkeit, alle diese Elemente auch ohne Generics zu verwenden. Man befindet sich nur dann auf der typsicheren Seite, wenn man die Vermischung von rohen Typen (wie *List*) und parametrisierten Typen (wie *List<String>*) vermeidet.

Hier noch ein Tipp aus dem Pragmatischen Programmierer [5]: Eine weitere Hilfe ist es, wenn man jedes Jahr eine neue Sprache lernt. Dies erweitert den Horizont und macht Abstraktionen verständlicher. Vielleicht fällt die Entscheidung auf eine Programmiersprache mit funktionalen Konzepten (wie Python [4], Groovy [6] oder Ruby [7]). Weitere Informationen zur funktionalen Programmierung, Beispiele oder „Why Functional Programming Matters“ findet man mit den Links [10] und [11]. Interessant in diesem Kontext ist auch die kleine „Erfolgsgeschichte“ von Paul Graham [12].

Zum Schluss geht noch ein Dank an Prof. Dr. Siedersleben (T-Systems Enterprise Services), Gerd Beneken (TU München) für die Motivation und an die Dozenten und Studenten der FH Rosenheim, deren Seminar „J2SE 5.0“ [8] die Grundlage für diesen Artikel geliefert hat.

Christian Neumann (Christian.Neumann@QAware.de) ist Softwarearchitekt bei QAware und Dozent an der Fachhochschule Rosenheim. Sein Hauptinteresse gilt dem qualitativ hochwertigen Software-Engineering, der modernen Softwarearchitektur und dem professionellen Programmieren.

Links & Literatur

[1] Johannes Nowak: Fortgeschrittene Programmierung mit Java 5, dpunkt, 2004

[2] Josef Adersberger: Java 5 Pitfalls: Fallstricke und Best Practices bei der Entwicklung mit Java 5, in *Java Magazin* 8.2005

[3] Johannes Siedersleben: Moderne Softwarearchitektur, dpunkt, 2004

[4] www.python.org

[5] Andrew Hunt, David Thomas: Der Pragmatische Programmierer, Hanser, 2003

[6] www.groovy.codehaus.org

[7] www.ruby-lang.org

[8] FH Rosenheim, Seminar „J2SE 5.0“: www.fh-rosenheim.de/~siedersleben/

[9] www.haskell.org

[10] www.cs.chalmers.se/~rjmh/Papers/whyfp.html

[11] groups.google.de/group/de.comp.lang.funktional

[12] Paul Graham: Beating the Averages: paulgraham.com/avg.html