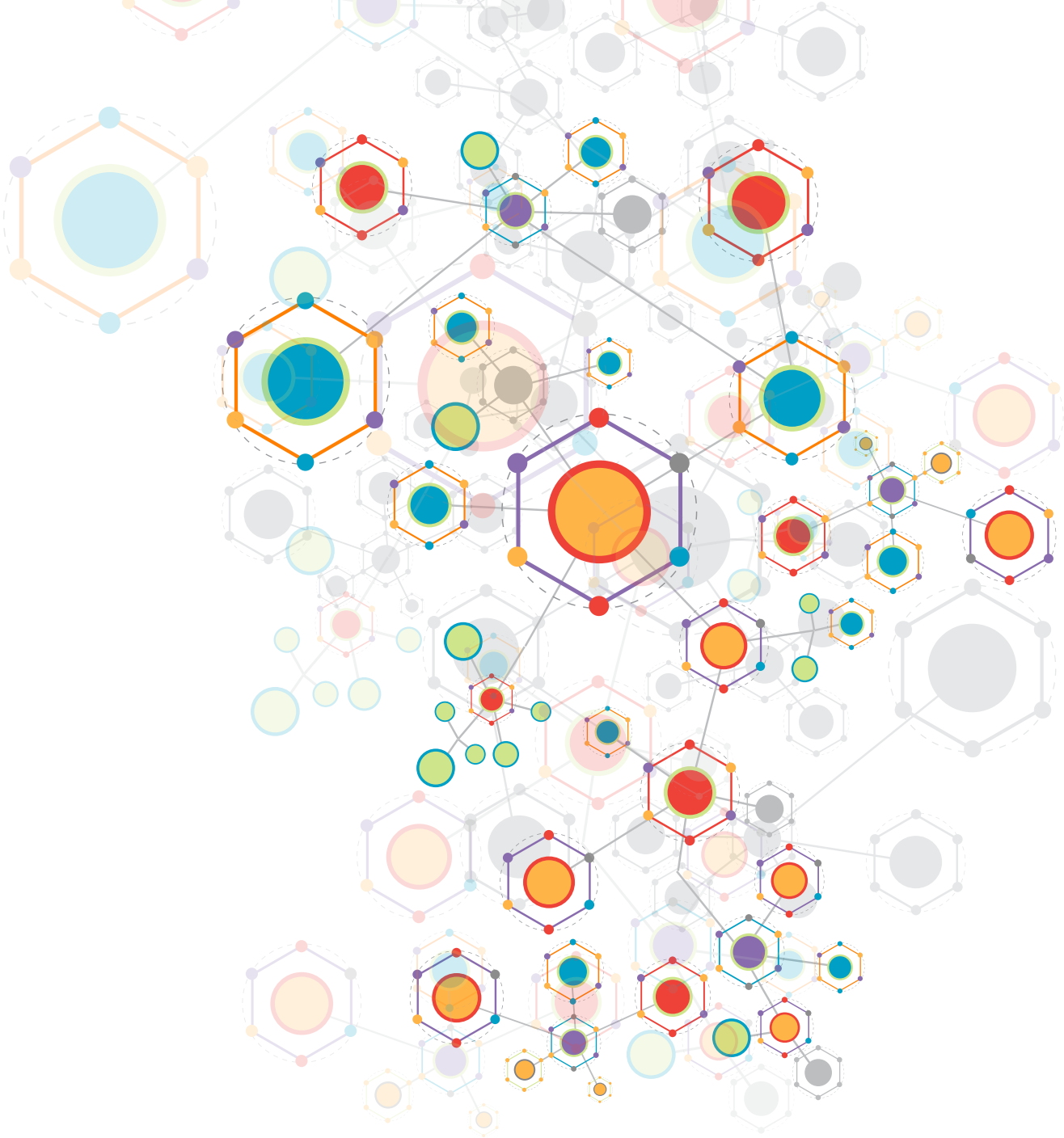

Kombinator als funktionales Entwurfsmuster in Java

Gregor Trefs, Freiberufler

Neben Streams und Optionals bietet Java 8 weitere Möglichkeiten, um Problemstellungen funktional zu lösen. Das Kombinator-Entwurfsmuster kombiniert kleine fachliche Funktionen situationsgerecht zu komplexer Fachlogik. Vorteile sind, neben der klaren Trennung von Verantwortlichkeiten, die Erweiterbarkeit, die Wiederverwendbarkeit und der deklarative beziehungsweise domänengetriebene Ansatz. Dieser Artikel beschreibt anhand praktischer Beispiele, wie das Kombinator-Entwurfsmuster funktioniert und wann der Einsatz sinnvoll ist.

```
int addOne(int i){
    return i + 1;
}
```

Listing 1

```
Function<Integer, Integer> addOne = i -> i + 1;
int compute(Function<Integer, Integer> f, int value){
    return f.apply(value);
}
compute(addOne, 1); // ergibt 2
```

Listing 2

```
Function<Integer, Integer> addOne = i -> i + 1;
Function<Integer, Integer> addTwo;
addTwo = i -> addOne.apply(addOne.apply(i));
Function<Integer, Integer> addThree = addTwo.
compose(addOne);
```

Listing 3

Das Kombinator-Entwurfsmuster, kurz Kombinator, folgt dem „Teile und Herrsche"-Prinzip. Eine Menge von Primitiven beschreibt die simpelsten Elemente einer Domäne und wird mit einer Menge von Kombinatoren zu komplexerer Logik kombiniert. Kombinator ähnelt also dem Kompositum-Entwurfsmuster. Der Unterschied liegt in den Dingen, die miteinander kombiniert werden. Im Kompositum sind es Objekte, die in einer Baumstruktur zusammengeführt werden; beim Kombinator sind es Funktionen, die mit Funktionen zu neuen Funktionen kombiniert werden.

Im Umkehrschluss sollten wir zunächst verstehen, was eine Funktion ist und was Funktionen höherer Ordnung sind. Danach betrachten wir das Kombinator-Entwurfsmuster am Beispiel einer Benutzervalidierung und diskutieren, welcher Rückgabewert in solch einer Domäne sinnvoll ist und wie dieser die Gestaltung unserer Primitiven und Kombinatoren beeinflusst. Zuletzt werden Vorteile, Herausforderungen und das Einsatzgebiet vorgestellt.

Funktionen und Funktionen höherer Ordnung

Java ist keine funktionale Sprache. Jedoch ist es möglich, Programme funktional zu gestalten. Betrachten wir dazu die Methode „addOne" in *Listing 1*. Die Logik ist simpel: Jeder übergebene Wert wird um 1 erhöht und zurückgegeben. Allerdings hat „addOne" zwei interessante Eigenschaften: Zum einen hängt der Rückgabewert ausschließlich vom Übergabeparameter „i" ab. Zum anderen gibt es, außer dem Rückgabewert, keinen anderen von außen wahrnehmbaren Effekt auf die Ausführung des Programms, „addOne" hat keine Seiteneffekte. Eine Methode mit diesen beiden Eigenschaften wird als „pure Funktion" bezeichnet. In diesem Artikel nehmen wir an, dass mit Funktion immer eine pure Funktion gemeint ist. Durch ihre Zustandslosigkeit sind Funktionen unter anderem einfach testbar, parallel ausführbar und ihr Verhalten ist nachvollziehbarer.

Funktionen sind Werte. Sie können anderen Funktionen übergeben und/oder von diesen zurückgegeben werden. Funktionen, die auf anderen Funktionen arbeiten, werden „Funktionen höherer Ordnung" genannt. Um das zu verdeutlichen, sehen wir in *Listing 2* die Funktion „addOne" in Lambda-Syntax und die höhere Funktion „compute".

Die Logik von „addOne" hat sich nicht geändert. Die Funktion „compute" nimmt eine Funktion „f" und einen Wert „value" entgegen und wendet „f" auf „value" an. Das Ergebnis mit den Parametern „addOne" und „1" ist „2".

Funktionen können auch miteinander zu neuen Funktionen verknüpft werden. Dabei beschreiben der Eingabe- und Ausgabetyp einer Funktion eine Schnittstelle und bestimmen die Verknüpfungsmöglichkeiten *(siehe Listing 3)*.

In Funktion „addTwo" wird „addOne" mit sich selbst verknüpft. Zuerst wird „addOne" auf den Parameter „i" angewendet und danach nochmals auf den Rückgabewert. Das Ergebnis ist „i", um „2" erhöht. Diese Art der Komposition ist als „default"-Methode auf der Schnittstelle „Function" verfügbar und wird für die Funktion „addThree" benutzt, die sich aus den Funktionen „addOne" und „addTwo" zusammensetzt.

Das Kombinator-Entwurfsmuster

Wie die meisten Konstrukte in der funktionalen Programmierung sind Primitive und Kombinatoren Namen für Abstraktionen. In Wikipedia sind Primitive in Programmiersprachen wie folgt beschrieben: „[...] Sprach-Primitive sind die simpelsten Elemente, die in einer Programmiersprache vorhanden sind. Ein Primitiv [...] ist ein atomares Element eines Ausdrucks in einer Sprache. Sie sind Einheiten mit Semantik [...]." Gleichermaßen sind Primitive im Kombinator-Entwurfsmuster die simpelsten Elemente innerhalb einer Domäne, etwa Addition und Multiplikation in der Domäne der Ganzzahlen. Kombinatoren beschreiben, wie Primitive und/oder bereits verknüpfte Strukturen miteinander in noch komplexere Strukturen verknüpft werden können, zum Beispiel die Verknüpfung von Multiplikation und Addition.

```
public class User{
   public final String name;
   public final int age;
   public final String email;

   public User(String name, int age, String email){
     this.name = name;
     this.age = age;
     this.email = email;
   }

   public boolean isValid(){
     return nameIsNotEmpty() && mailContainsAtSign();
   }

   private boolean nameIsNotEmpty(){
     return !name.trim().isEmpty();
   }

   private boolean mailContainsAtSign(){
     return email.contains("@");
   }
}

new User("Gregor", 31, "nicemail@gmail.com").is-
Valid(); // true
```

Listing 4

```
interface UserValidation extends Function<User, Boolean>{}

UserValidation nameIsNotEmpty = u -> !u.name.trim().isEmpty();
UserValidation mailContainsAtSign = u -> u.email.contains("@");
User gregor = new User("Gregor", 31, "nicemail@gmail.com");

nameIsNotEmpty.apply(gregor) &&
  mailContainsAtSing.apply(gregor); // true
```

Listing 5

```
interface UserValidation extends Function<User, Boolean>{
    UserValidation nameIsNotEmpty = u ->
       !u.name.trim().isEmpty();
    UserValidation mailContainsAtSign = u ->
       u.email.contains("@");

    default UserValidation and(UserValidation other) {
        return user -> this.apply(user) && other.apply(user);
    }
}

User gregor = new User("Gregor", 30, "nicemail@gmail.com");
nameIsNotEmpty.and(mailContainsAtSign).apply(gregor); // true
```

Listing 6

```
List<User> users = findAllUsers()
    .stream().parallel()
    // Konvertierung von UserValidation zu Predicate
    .filter(nameIsNotEmpty.and(mailContainsAtSign)::
apply)
    .collect(Collectors.toList());
```

Listing 7

Die Validierung mit Kombinatoren

Stellen wir uns vor, wir sind beauftragt, die Benutzer einer Applikation zu validieren. Benutzer sind berechtigt, einen Dienst zu nutzen, wenn der Name nicht leer ist und die E-Mail Adresse ein „@"-Zeichen enthält. Ein direkter Weg, dies zu modellieren, wären Abfragemethoden in der entsprechenden Entität *(siehe Listing 4)*.

Obwohl sich das gut liest, hat diese Lösung ein paar Nachteile. Was passiert, wenn sich die Validierungsregeln ändern? Zum Beispiel kann es sein, dass Benutzer zur Nutzung von bestimmten Diensten älter als 14 sein müssen. In diesem Fall ist die Methode „isValid" anzupassen, was eine Verletzung des Single-Responsibility-Prinzips ist.

Bevor wir mit dem Refactoring beginnen und die Validierung in eine eigene Klasse extrahieren, sollten wir noch einmal darüber nachdenken, was wir eigentlich ausdrücken wollen. Benutzer sind genau dann valide, wenn eine Menge an Regeln erfüllt ist. Eine Regel beschreibt hierbei eine Benutzer-Eigenschaft. Verschiedene Regeln können miteinander zu komplexeren Regeln kombiniert werden. Die Anwendung einer Regel ergibt ein Validierungsergebnis, das den Ausgang der Validierung beschreibt. Zunächst nehmen wir an, dass der boolesche Wert „true" Benutzer als „valide" auszeichnet. *Listing 5* drückt das in Funktionen aus.

Der Code ist etwas langatmig. Warum werden die Regeln „nameIsNotEmpty" und „mailContainsAtSign" manuell erstellt und verknüpft? Sind vier Zeilen Initialisierung für gerade mal eine Anwendungszeile wirklich nötig? Auch wenn der Code etwas wortreich ist, so enthält er doch alle Bausteine für das Kombinator-Entwurfsmuster: „nameIsNotEmpty" und „mailContainsAtSign" sind Primitive und „&&" ist ein Kandidat für einen Kombinator. Seit Java 8 ist es möglich, Primitive und Kombinatoren in den „UserValidation"-Typ zu schieben *(siehe Listing 6)*.

Primitive sind als statische Variablen definiert und Kombinatoren als „default"-Methoden. Beide Primitive „nameIsNotEmpty" und „mailContainsAtSign" sind ein Lambda-Ausdruck mit „UserValidation" als Ziel-Typ. Im Unterschied dazu ist der „and"-Kombinator eine Funktion höherer Ordnung, die zwei „UserValidation" mit dem „&&"-Operator verknüpft. Hinzu kommen zwei Beobachtungen: Zum einen werden „UserValidation" während der Konstruktion einer Benutzervalidierung nicht ausgeführt. Zum anderen haben sie keinen eigenen Zustand. Unter anderem bedeutet das, dass eine

```
interface ValidationResult{
    static ValidationResult valid(){
      // Gibt immer die selbe Instanz zurück
      // mit isValid gleich true
      // und getReason gleich Optional.empty()
      return ValidationSupport.valid();
    }

    static ValidationResult invalid(String reason){
        // Gibt neue Instanz mit
        // isValid gleich false
        // und getReason gleich Optional.of(reason)
        return new Invalid(reason);
    }

    boolean isValid();

    Optional<String> getReason();
}
```

Listing 8

```
interface UserValidation extends Function<User, ValidationResult>{
    UserValidation nameIsNotEmpty =
       user -> !user.name.trim().isEmpty()?valid():invalid("Name is empty.")

    UserValidation mailContainsAtSign =
       user -> user.email.contains("@")?valid():invalid("Missing @-sign.");

    default UserValidation and(UserValidation other) {
        return user -> {
            final ValidationResult r = this.apply(user);
            return r.isValid() ? other.apply(user):r;
        };
    }
}

UserValidation v = nameIsNotEmpty.and(mailContainsAtSign);
User gregor = new User("", 31, "mail@mailinator.com");

ValidationResult r = v.apply(gregor);
r.getReason().ifPresent(System.out::println); // Name is empty.
```

Listing 9

```
UserValidation ext = nameIsNotEmpty.and(user -> … );
```

Listing 10

Benutzer-Validierung parallel auf mehreren Benutzern ausgeführt werden kann *(siehe Listing 7)*.

Aus einer anderen Perspektive betrachtet, ist der Typ „UserValidation" ein eigener Kontext mit Elementen aus der Domäne der Benutzer-Validierung. Der Compiler garantiert, dass bei der Beschreibung einer Benutzer-Validierung nur Elemente aus dem „UserValidation"-Kontext benutzt werden. Dies bedeutet den Aufbau einer eingebetteten domänenspezifischen Sprache. Der Code wird deklarativer und orientiert sich an der Domäne.

Das Validierungsergebnis

„Boolean" ist keine gute Wahl, um Validierungsergebnisse zu repräsentieren. In Java ist es nur schwer möglich, fremden Code anzupassen. Der Typ „Boolean" kann nicht um Abfrage-Methoden erweitert werden, um festzustellen, welche Regel das Ergebnis invalidiert. Auch die Bedeutung eines booleschen Werts ist implizit und kontextspezifisch. Zum Beispiel kann die Annahme, dass „true" im Validierungskontext einen validen Benutzer auszeichnet, in einem anderen Kontext falsch sein. Aus diesen Gründen wird ein neuer Typ „ValidationResult" benötigt, der beschreibt, warum ein Benutzer invalide ist *(siehe Listing 8)*.

Ein „ValidationResult" ist entweder „valid" oder „invalid". Die Gleichheit zweier „ValidationResult" ist bestimmt durch die Gleichheit ihrer Felder. Zum Beispiel sind zwei „Invalid"-Instanzen genau dann gleich, wenn sie den gleichen Invalidierungsgrund haben. Im Gegensatz dazu sind valide Ergebnisse nicht anhand ihres Zustands unterscheidbar; hier reicht eine einzelne anonyme Instanz. Das bedeutet, „valid" und „invalid" sind Wert-Objekte. *Listing 9* zeigt, wie „UserValidation" für die Nutzung von „ValidationResult" angepasst wird.

Die Primitiven „nameIsNotEmpty" und „mailContainsAtSign" wurden entsprechend geändert. Der „and"-Kombinator berechnet das Ergebnis von „other" nur, falls „this" valide ist. Zum Beispiel wird die E-Mail eines Benutzers nur geprüft, falls der Name vorhanden ist. Aus Entwicklerperspektive übermittelt die Änderung am API mehr Informationen als ein einfacher boolescher Wert. Jedoch gibt es Makel am Design. Zum Beispiel wäre es in einem Web-Kontext hilfreich, alle Validierungsregeln für einen ausführlichen Bericht zu evaluieren. Der „and"-Kombinator beendet jedoch die Validierung, sobald eine Regel nicht erfüllt wurde. Die Implementierung eines entsprechenden „all"-Kombinators ist dem Leser überlassen.

Vorteile und Herausforderungen

Die Vorteile des Kombinator-Entwurfsmusters sind der domänenspezifische Ansatz, explizit modellierte Informationen, die Trennung von Verantwortlichkeiten, Erweiterbarkeit und Wiederverwendbarkeit.

- **Domänenspezifischer Ansatz**
  Das Beispiel der Benutzervalidierung hebt den domänenspezifischen Ansatz hervor. Regeln und deren Verknüpfungsmöglichkeiten werden in einem Kontext zusammengefasst, eine eingebettete domänenspezifische Sprache wird aufgebaut.

```
Comparator<User> byAge = Comparator.comparing(u -> u.age);
Comparator<User> byName = Comparator.comparing(u -> u.name);
Comparator<User> byAgeAndName =
                 byAge.thenComparing(byName);
Random r = new Random(12);
IntStream.range(0,100)
  .mapToObj(i -> new User("user: "+i,r.nextInt(i+1), "m"+i))
  .sorted(byAgeAndName);
```

Listing 11

- **Explizit modellierte Informationen**
  Das Ergebnis einer Validierung wird explizit modelliert und trägt mehr Informationen als ein integrierter Datentyp wie beispielsweise „Boolean". Zudem sind die einzelnen Validierungsregeln und Verknüpfungsmöglichkeiten klar benannt.
- **Trennung der Verantwortlichkeiten**
  Die Verantwortlichkeiten zwischen Primitiven und Kombinatoren sind klar aufgeteilt. In der Benutzer-Validierung sind Primitive die simpelsten Regeln, die einen Benutzer beschreiben. Kombinatoren sind die Verknüpfungsmöglichkeiten, wie diese Regeln zu komplexeren Regeln kombiniert werden können.
- **Erweiterbarkeit**
  Durch die Verwendung des Kontexts als Zieltyp kann die Benutzervalidierung einfach um eigene Regeln erweitert werden, die nicht im eigentlichen Kontext definiert sind *(siehe Listing 10)*.
- **Wiederverwendbarkeit**
  Einmal beschrieben, kann eine Funktion vielfach angewendet werden. Zum Beispiel können komplexe Validierungsregeln in verschiedenen Diensten verwendet werden.

Die Herausforderungen des Kombinator-Entwurfsmusters sind das Verständnis über Funktionen und die Bestimmung von Primitiven und Kombinatoren in einer Domäne:

- **Verständnis der funktionalen Programmierung**
  Seiteneffekte wie beispielsweise das Werfen von Ausnahmen sind in der funktionalen Programmierung nicht erlaubt. Die Frage ist also, wie eine funktionale Fehlerbehandlung gestaltet werden kann. Solche Fragestellungen treiben die Entwicklung eines funktionalen API und erfordern daher ein funktionales Verständnis.
- **Bestimmung von Primitiven und Kombinatoren**
  Es gibt verschiedene Herangehensweisen, Primitive und Kombinatoren in einer Domäne zu bestimmen. Eine davon nennt sich „algebraisches Design". Hierbei werden auf einer abstrakteren Ebene Typen, Funktionen und deren Eigenschaften bestimmt beziehungsweise diskutiert, bis sie den Anforderungen entsprechen; dann wird die eigentliche Implementierung geschrieben. Eine weitere Herangehensweise ist die „testgetriebene Entwicklung". Hierbei beschreiben Tests das erwartete Verhalten von Funktionen und helfen bei der Bestimmung von Primitiven und Kombinatoren durch Herausarbeitung der Namen und Eigenschaften.

Wann der Einsatz sinnvoll ist

Das Kombinator-Entwurfsmuster hilft bei der Gestaltung eines modularen API. Der Typ „Comperator" ist beispielsweise seit Java 8 auf diese Art aufgebaut und ermöglicht so eine Vielzahl von Kombinationen. *Listing 11* zeigt, wie Benutzer nach Name und Alter sortiert werden.

Um den gleichen Funktionsumfang ohne Kombinator zu erreichen, müsste jede mögliche Kombination an „Comparator" als eigene Methode definiert werden. Unter Berücksichtigung der Unterstützung von primitiven Datentypen wären das mehr als hundert Funktionen. Mit Kombinator sind es hingegen nur 16.

Allgemein ist das Kombinator-Entwurfsmuster sinnvoll, wenn das grundlegende Konzept eine Funktion ist. Ein Beispiel wäre das Parsing. Ein Parser ist eine Funktion, die Zeichen in eine Struktur überführt. Parser für beliebige Grammatiken werden aus primitiven Parser-Funktionen kombiniert. So erkennt eine Parser-Funktion einen Buchstaben und eine andere erkennt Zahlen. Mit einem Kombinator für ein logisches Oder und einem für Vielfache können beliebige alphanumerische Zeichen erkannt werden.

Fazit

Ein gut umgesetztes Kombinator-Entwurfsmuster hat einige Vorteile. Hervorzuheben ist hier die Trennung von Verantwortlichkeiten in einzelne Funktionen, die das Single-Responsibility-Prinzip stärkt. Besonders bei der Gestaltung eines API ist das sinnvoll. Allerdings kann die funktionale Denkweise auch herausfordernd sein. Wie wird beispielsweise Fehlerbehandlung funktional gestaltet? Wie werden die richtigen Primitiven und Kombinatoren bestimmt?

Auch wenn dieser Artikel Lösungsvorschläge für diese Fragestellungen präsentiert hat, liegt es immer noch am Team, wie das Entwurfsmuster eingesetzt wird und ob Alternativen im aktuellen Kontext nicht besser geeignet sind. Dennoch zeigt das Kombinator-Entwurfsmuster, dass es in Java möglich ist, Programme funktional zu gestalten, auch wenn Java selbst keine funktionale Sprache ist. Weitere Informationen unter *„https://github.com/gtrefs/javaaktuell_combinator"*.

Gregor Trefs
gregor.trefs@gmail.com

Gregor Trefs' erstes Programm war ein in BASIC geschriebenes Text-Adventure. Inzwischen ist er Mitorganisator der Java User Group Mannheim (majug) und schreibt regelmäßig auf seinem Blog. Sein persönliches Code-Highlight 2016 war sein Beitrag zu „vavr".