Erweiterbare Fluent Interface DSLs mit Java entwickeln

Flüssiger Baukasten

In vielen Projekten werden Domain Specific Languages für die Beschreibung fachlicher oder technischer Strukturen oder Abläufe verwendet. Oft kommen für die Implementierung solcher DSLs mittlerweile flexible Sprachen, wie Groovy oder Scala, zum Einsatz. Aber auch in Java ist die Umsetzung mächtiger, erweiterbarer DSLs möglich.

von Konstantin Diener und Valentino Pola

Als beispielhaftes Anwendungsszenario für eine Fluent Interface DSL [1] betrachten wir ein Redaktionssystem, in dem nach Themen gruppiert einzelne kurze Nachrichten veröffentlicht werden können (**Abb. 1**). Listing 1 und 2 enthalten die Implementierung der entsprechenden beiden Klassen *Topic* und *NewsSnippet*.

Das Thema (Klasse *Topic*) hat einen Namen und kann beliebig viele Kurznachrichten aufnehmen. Diese Kurznachrichten haben einen Titel, eine kurze Beschreibung (optional) und einen Inhalt (optional) sowie eine Reihe von Schlagwörtern (optional), die das Auffinden der Nachrichten per Suche erleichtern sollen. Werden die Nachrichten nachträglich geändert, erfolgt eine Historisierung des ursprünglichen Inhalts.

In **Abbildung 2** ist ein Beispiel dargestellt, in dem die oberste Kurznachricht zweimal geändert wurde. Gültig ist nur noch die linke Version, die rechts davon angeordneten haben keine Verbindung mehr zum *Topic*-Objekt und dokumentieren lediglich frühere Fassungen. Anhand der Attribute *validFrom* und *validTo* ist erkennbar, über welchen Zeitraum ein Eintrag gültig war oder noch ist – der Wert 31.12.9999 steht dabei für „Unendlich“, um beispielsweise BETWEEN-Abfragen in Sprachen wie SQL oder JP-QL zu erleichtern. Unveränderlich ist allein das Attribut *title*, das redundant in allen Versionen der Nachricht abgelegt wird, um das Beispiel für diesen Artikel möglichst kompakt zu halten. Das in **Abbildung 2** dargestellte Beispiel würde man klassisch wie in Listing 3 implementieren.

Die Implementierung ist an dieser Stelle sehr technisch orientiert: Anhand von Hilfsklassen wie *SimpleDateFormat* werden Datumswerte erzeugt und um dem *newsItem*-Objekt Tags zuzuweisen, wird immer wieder die Methode *addTag* aufgerufen, was sehr unübersichtlich ist. Des Weiteren können durch diese Implementierung fachlich ungültige Objekte erzeugt werden, da der Entwickler bspw. mit der Initialisierung der Instanz *news-*

Listing 1

```
public class Topic {
  String name;
  List<NewsSnippet> newsSnippets = new ArrayList<NewsSnippet>();

  public Topic(String name) {
    this.name = name;
  }
}
```

Listing 2

```
public class NewsSnippet {
  Topic topic;
  String title;
  String description;
  String content;
  Date validFrom;
  Date validTo;
  List<String> tags = new ArrayList<String>();
}
```

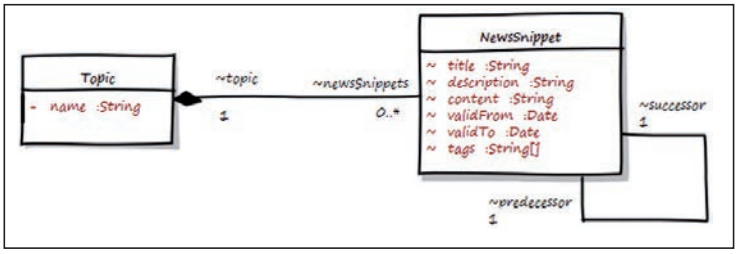

Abb. 1: Domänenmodell des Redaktionssystems

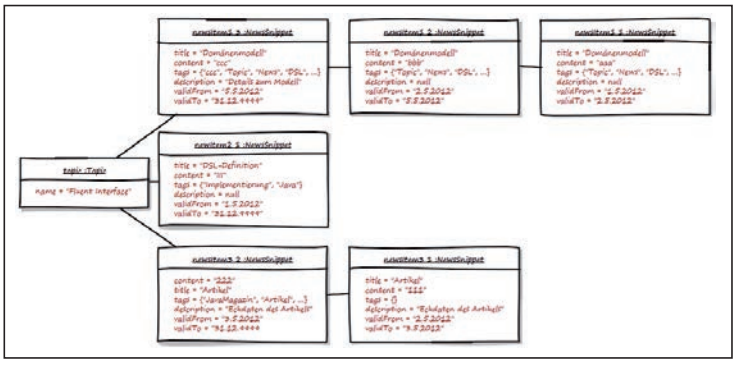

Abb. 2: Beispielhafte Domänenobjekte

Listing 3

```
Topic fluentInterface = new Topic("Fluent Interface");
NewsSnippet newsItem1 = new NewsSnippet("Domänenmodell");
newsItem1.setContent("aaa");
newsItem1.setValidFrom(new SimpleDateFormat().parse("05/01/2012"));
newsItem1.setValidTo(new SimpleDateFormat().parse("05/02/2012"));
newsItem1.addTag("Topic");
newsItem1.addTag("News");
newsItem1.addTag("DSL");
...
newsItem1.setTopic(fluentInterface);
```

Listing 4

```
public class NewsSnippetBuilder {

  private final NewsSnippet objectUnderConstruction;
  private final Topic topic;
  public NewsSnippetBuilder(Topic topic, String title) {
    this.topic = topic;
    this.objectUnderConstruction = new NewsSnippet();
    this.objectUnderConstruction.title = title;
    this.objectUnderConstruction.validFrom = new Date();
    Calendar calendar = Calendar.getInstance();
    calendar.set(9999, 11, 31);
    this.objectUnderConstruction.validTo = calendar.getTime();
  }

  public NewsSnippetBuilder containing(String contents) {
    this.objectUnderConstruction.content = contents;
    return this;
  }

  public NewsSnippetBuilder taggedBy(String... tags) {
    for (String tag : tags) {
      if (!this.objectUnderConstruction.tags.contains(tag)) {
        this.objectUnderConstruction.tags.add(tag);
      }
    }
    return this;
  }

  public void add() {
    this.objectUnderConstruction.topic = topic;
    topic.newsSnippets.add(this.objectUnderConstruction);
  }
}
```

Listing 5

```
Calendar calendar = Calendar.getInstance();
calendar.set(9999, 11, 31);

new Topic("Fluent Interface").newNewsSnippet("Domänenmodell")
  .containing("aaa")
  .taggedBy("Topic", "News", "DSL")
  .validFrom(new Date()).to(calendar.getTime())
  .add();
```

Item1 aufhören kann, ohne es einem *Topic* zuzuweisen. Um diesen beiden Kritikpunkten entgegenzuwirken, wird das Beispiel nachfolgend schrittweise über das Erbauer-Muster [2] in Verbindung mit *Method Chaining* (Kasten: „DSL Patterns“) überführt und somit ein Fluent Interface für die Erzeugung von Kurznachrichten erstellt. Mehr zum Fluent Interface steht im gleichnamigen Kasten.

Als Erstes erstellen wir ein Thema „Fluent Interface“ und fügen ihm die im Diagramm dargestellte Kurznachricht mit dem Titel „Domänenmodell“ hinzu:

```
new Topic("Fluent Interface").newNewsSnippet("Domänenmodell")
  .containing("aaa")
  .taggedBy("Topic", "News", "DSL")
  .add();
```

Damit der Erbauer zum Anlegen einer neuen Kurznachricht in der Klasse *Topic* zur Verfügung steht, erhält die Klasse die Methode *newNewsSnippet*:

```
public NewsSnippetBuilder newNewsSnippet(String title) {
  return new NewsSnippetBuilder(this, title);
}
```

Die Methode erzeugt den Erbauer mit einer Referenz auf *Topic*, zu dem die neue Kurznachricht gehören soll, und dem gewählten Titel. Damit ist sichergestellt, dass der Titel als einziges nicht optionales Element der Nachricht immer mit einem Wert belegt ist. Der *NewsSnippetBuilder* (Listing 4) erzeugt im Konstruktor eine neue Instanz der Klasse *NewsSnippet* und belegt deren Attribute mit Standardwerten.

An dieser Stelle zeigt sich auch, weshalb die Attribute der Klasse *NewsSnippet* alle package scoped sind: Der Erbauer liegt im selben Package wie das Domänenobjekt und kann als einziger die Attribute setzen, Klas-

Listing 6

```
public class DateRangeBuilder {

  private final Date from;

  private final NewsSnippetBuilder newsSnippetBuilder;

  public DateRangeBuilder(Date from, NewsSnippetBuilder
                                                  newsSnippetBuilder) {
    this.from = from;
    this.newsSnippetBuilder = newsSnippetBuilder;
  }

  public NewsSnippetBuilder to(Date to) {
    this.newsSnippetBuilder.setValidationPeriod(this.from, to);
    return this.newsSnippetBuilder;
  }
}
```

sen in anderen Packages ist dies nicht möglich. Damit entspricht die Klasse nicht dem Bean-Standard und zur Integration bspw. in JSF sind weitere Schritte nötig, die aber nicht Gegenstand dieses Artikels sein sollen. Die beiden DSL-Methoden *containing* und *taggedBy* setzen den Inhalt und die Tags der Kurznachricht und liefern jeweils wieder den Erbauer selbst zurück. Die Methode *add* beendet die Methodenkette und fügt die neue Nachricht endgültig dem *Topic*-Objekt hinzu.

Komponentenbasierte DSL

Bislang war das *NewsSnippet*-Objekt nach jedem Methodenaufruf der DSL in einem konsistenten Zustand. Beim Definieren der Gültigkeitsperiode (*validFrom*, *validTo*) sind im Gegensatz dazu zwei Methodenaufrufe notwendig, um einen konsistenten Zustand herzustellen – sowohl Start- als auch Enddatum müssen gesetzt werden (Listing 5).

Damit ein konsistenter Zustand gewährleistet wird, erfolgt die Erstellung der Zeitperiode in einem separaten Erbauer (Listing 6), der nach der Methode *to* wieder den *NewsSnippetBuilder* zurückliefert. Diese Ausgliederung von DSL-Elementen wird als „Object Scoping“ bezeichnet (Kasten: „DSL Patterns“).

Damit der *DateRangeBuilder* nach dem Aufruf der Methode *to* den Kontrollfluss wieder an den aufrufenden Erbauer zurückliefern und die Werte zurückschreiben kann, enthält die Klasse im Moment eine enge Kopplung zum *NewsSnippetBuilder*. In der Praxis wird die Erstellung von Datumsperioden vermutlich nicht nur in einer DSL bzw. einem Erbauer Verwendung finden, weshalb die Verbindung zwischen *NewsSnippetBuilder* und *DateRangeBuilder* durch ein Callback entkoppelt wird:

Listing 7

```
public class DateRangeBuilder<T extends DateRangeBuilderCallback> {

  private final Date from;
  private final T callback;

  public DateRangeBuilder(Date from, T callback) {
    this.from = from;
    this.callback = callback;
  }

  public T to(Date to) {
    this.callback.setDateRange(this.from, to);
    return this.callback;
  }
}
```

```
public interface DateRangeBuilderCallback {
  void setDateRange(Date from, Date to);
}
```

Die Klasse *DateBuilder* wird in eine parametrisierte Klasse umgewandelt (Listing 7), die mit dem Typ des „umschließenden Erbauers“ parametrisiert wird. Die Übertragung des Start- und Enddatums erfolgt über die Methode *DateRangeBuilderCallback.setDateRange*. Der *NewsSnippetBuilder* implementiert diese Schnittstelle (Listing 8), wodurch allerdings die Schnittstelle der Fluent Interface DSL verunreinigt wird: Bei der Verwendung des *NewsSnippetBuilders* ist die Callback-Methode jetzt ebenfalls aufrufbar (**Abb. 3**).

Um diese Verunreinigung zu vermeiden, wird im *DateRangeBuilder* die Referenz auf das Callback von der Referenz auf den „umschließenden Erbauer“ getrennt (Listing 9): Der *NewsSnippetBuilder* kann das Callback in einer anonymen Klasse implementieren und die DSL-Schnittstelle sauber halten (Listing 10).

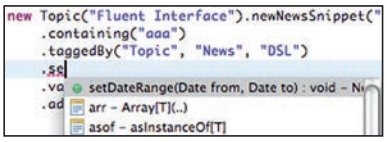

Abb. 3: „Verunreinigte DSL-Schnittstelle“

DSL Patterns

In seinem Buch „Domain Specific Languages“ [3] definiert Martin Fowler eine ganze Reihe von DSL Patterns [4]; eine Auswahl davon enthält folgende Liste:

- Method Chaining: Als Method Chaining bezeichnet man einen Ansatz, bei dem jeder Methodenaufruf auf einem Objekt eine Referenz auf das Objekt selbst zurückliefert, sodass in beliebiger Folge Methoden auf ein und demselben Objekt ausgeführt werden können. Zwar bricht Method Chaining gängige Paradigmen der Objektorientierung, erhöht bei zielgerichtetem Einsatz aber die Lesbarkeit des Quellcodes und ist die Grundlage für das Design von Fluent Interfaces, da sich Aufrufketten konstruieren lassen, die der natürlichen Sprache recht ähnlich sind.
- Nested Functions: Nested Functions werden in den meisten Fällen dazu eingesetzt, das Manko fehlender Named-Parameter (bspw. in Java) auszugleichen. Dazu werden statische Hilfsmethoden bereitgestellt und als Argumente übergeben. Soll beispielsweise ein Kreditobjekt mit einer Laufzeit von zwölf Monaten, einem Betrag von 1000 Euro und einem Zins von 3 Prozent erzeugt werden, dann würde im klassischen Sinn wie folgend ein Konstruktor aufgerufen werden:

  ```
  new Kredit(12,3,1000);
  ```

- Durch den Einsatz des Nested Functions Pattern wird an der Stelle die Lesbarkeit des Codes erhöht, indem bspw. folgender Code geschrieben werden würde:

  ```
  new Kredit(Laufzeit(12), Zins(3), Betrag(1000));
  ```

- Object Scoping: Ein starker Nachteil von Nested Functions ist die nicht an Objekte gebundene Definition von Methoden. Innerhalb eines Namensraums kann es so schnell zu Namenskonflikten kommen. Object Scoping behebt diesen Nachteil, indem DSL-Methoden nicht statisch implementiert, sondern an ein Objekt gebunden werden. Um dann das gleiche Ziel wie Nested Functions (nämlich bessere Lesbarkeit) zu erreichen, werden die Nested Functions innerhalb einer eigenen Erbauerklasse definiert.

Flüssige Konstanten

Durch die bislang implementierte Fluent Interface DSL ergibt sich ein recht gut lesbarer Fluss von Methodenaufrufen, der nur durch die Erzeugung der Datumswerte für den *DateRangeBuilder* unterbrochen wird:

```
Calendar calendar = Calendar.getInstance();
calendar.set(9999, 11, 31);

...
  .validFrom(new Date()).to(calendar.getTime())
```

Wünschenswert wäre, dass sich die Formulierung der beiden Datumswerte ebenfalls nahtlos in die DSL einfügt und für gängige Zeitpunkte wie „heute“, „gestern“, „morgen“ oder „unendlich“ entsprechende Sprachkonstrukte existieren:

```
  .validFrom(today()).to(infinity())
```

Listing 8

```
public class NewsSnippetBuilder implements DateRangeBuilderCallback {

  ...
  public DateRangeBuilder<NewsSnippetBuilder> validFrom(Date date) {
    return new DateRangeBuilder<NewsSnippetBuilder>(date, this);
  }

  public void setDateRange(Date from, Date to) {
    this.objectUnderConstruction.validFrom = from;
    this.objectUnderConstruction.validTo = to;
  }
}
```

Listing 9

```
public class DateRangeBuilder<T> {

  private final Date from;
  private final T caller;
  private final DateRangeBuilderCallback callback;

  public DateRangeBuilder(Date from, T caller,
                          DateRangeBuilderCallback callback) {
    this.from = from;
    this.caller = caller;
    this.callback = callback;
  }

  public T to(Date to) {
    this.callback.setDateRange(this.from, to);
    return this.caller;
  }
}
```

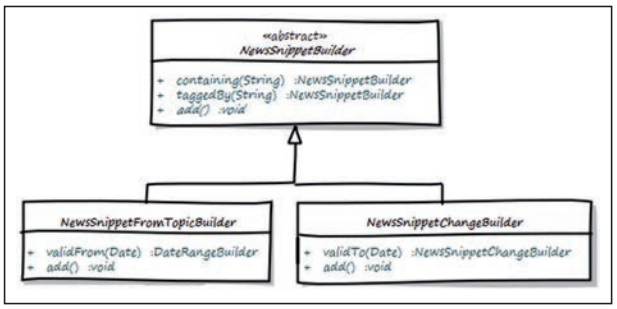

Abb. 4: Klassenhierarchie der „NewsSnippetBuilder“

Für die Umsetzung dieser Anforderung gibt es zwei Alternativen („Nested Functions“ im Kasten „DSL Patterns“):

1. Die Anwendung des Musters „Schablonenmethode“ [2], wobei die DSL eine abstrakte Superklasse und entsprechende Methoden für *today*, *infinity* usw. bereitstellt. Wird eine Klasse erstellt, die von der DSL Gebrauch macht, muss sie von dieser abstrakten Basisklasse abgeleitet werden. Diesen Weg verwendet beispielsweise der Route Builder des Camel Frameworks [5].
2. Die Sprachkonstrukte werden als öffentliche Klassenmethoden umgesetzt und in der verwendenden Klasse mit *import static* sichtbar gemacht. Die Implementierung dieser Alternative ist in Listing 11 und 12 dargestellt.

Spezialflüssigkeiten

In der aktuellen Form gestattet die Fluent Interface DSL bereits die Erstellung neuer Kurznachrichten. Ein zweiter Anwendungsfall sollte die nachträgliche Änderung dieser Nachrichten und eine entsprechende Versionierung sein. Bei dieser Änderung lassen sich alle Eigenschaften der Nachricht mit Ausnahme des Titels und des Gültigkeitsbeginns verändern. Der Gültigkeitsbeginn ist

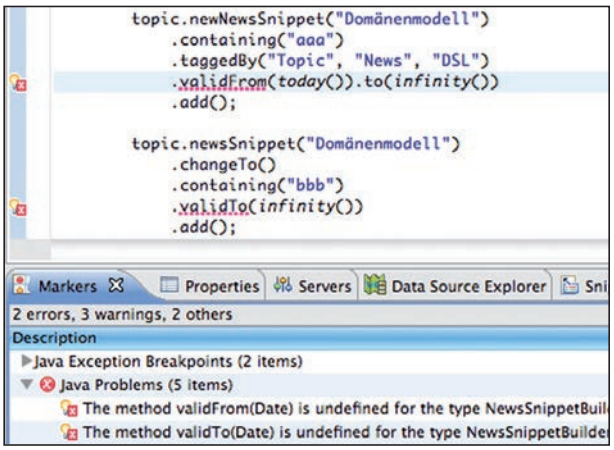

Abb. 5: Fehler bei der Verwendung der DSL

unveränderbar, da die Anwendung in unserem Beispiel davon ausgehen soll, dass eine Änderung sofort wirksam wird (Listing 13).

Für diesen Zweck ist wieder ein Erbauer notwendig, der durch den Aufruf der Methode *changeTo* der Klasse *NewsSnippet* erstellt und zurückgegeben wird. Es fällt auf, dass sich dieser Erbauer von dem Pendant zur Erstellung neuer Kurznachrichten nur geringfügig unterscheidet:

- Der Erbauer zum Verändern von Nachrichten hat statt der Methode *validFrom* eine Methode *validTo*, weil die Änderung sofort gültig wird.
- Die Methode *add* ist abweichend implementiert, da das bislang mit *Topic* verbundene *NewsSnippet*-Objekt abgekoppelt und mit der neuen Version über eine Vorgänger-Nachfolger-Beziehung verknüpft werden muss (Listing 14).

Um die Gemeinsamkeiten der beiden Erbauer an einer zentralen Stelle zu kapseln, wird der *NewsSnippetBuilder* durch ein Refactoring in eine allgemeine Klasse und zwei Spezialisierungen aufgeteilt (**Abb. 4**).

Listing 10

```
public class NewsSnippetBuilder {

  ...
  public DateRangeBuilder<NewsSnippetBuilder> validFrom(Date date) {
    return new DateRangeBuilder<NewsSnippetBuilder>(
                                  date, this, new DateRangeBuilderCallback() {

      public void setDateRange(Date from, Date to) {
        _setDateRange(from, to);
      }
    });
  }

  protected void _setDateRange(Date from, Date to) {
    this.objectUnderConstruction.validFrom = from;
    this.objectUnderConstruction.validTo = to;
  }
}
```

Listing 11

```
public class DateConstants {

  public static Date today() {
    return new Date();
  }

  public static Date infinity() throws ParseException {
    return createDateObject(9999, 11, 31);
  }

  private static Date createDateObject(int year, int month, int day) {
    Calendar calendar = Calendar.getInstance();
    calendar.set(year, month, day);
    return calendar.getTime();
  }
}
```

Listing 12

```
import static de.coinor.dsls.news.DateConstants.infinity;
import static de.coinor.dsls.news.DateConstants.today;

...
new Topic("Fluent Interface").newNewsSnippet("Domänenmodell")
  .containing("aaa")
  .taggedBy("Topic", "News", "DSL")
  .validFrom(today()).to(infinity())
  .add();
```

Listing 13

```
Topic topic = new Topic("Fluent Interface");
topic.newNewsSnippet("Domänenmodell")
...
topic.newsSnippet("Domänenmodell")
  .changeTo()
  .containing("bbb")
  .validTo(infinity())
  .add();
```

Listing 14

```
public void add() {
  objectUnderConstruction.predecessor = oldNewsSnippet;
  oldNewsSnippet.successor = objectUnderConstruction;

  Topic topic = oldNewsSnippet.topic;
  objectUnderConstruction.topic = topic;
  oldNewsSnippet.topic = null;

  topic.newsSnippets.remove(oldNewsSnippet);
  topic.newsSnippets.add(objectUnderConstruction);
}
```

Listing 15

```
public abstract class NewsSnippetBuilder<T extends NewsSnippetBuilder> {

  protected final NewsSnippet objectUnderConstruction;
  public NewsSnippetBuilder(String title) {
    this.objectUnderConstruction = new NewsSnippet();
    this.objectUnderConstruction.title = title;
  }

  public T containing(String contents) {
    this.objectUnderConstruction.content = contents;
    return (T) this;
  }
  ...
}

public class NewsSnippetFromTopicBuilder
extends NewsSnippetBuilder<NewsSnippetFromTopicBuilder> {
...
}
```

Über diese Trennung ist sichergestellt, dass allgemeine Erweiterungen am *NewsSnippet* nur in einem Erbauer realisiert werden müssen und keine Duplikation von Code stattfindet. Versucht man allerdings nach dieser Änderung, die DSL zu verwenden, kommt es zu Fehlern (**Abb. 5**).

Diese Fehler sind durch die Implementierung des abstrakten Erbauers bedingt: Die Rückgabewerte der DSL-Methoden *containing* und *taggedBy* sind vom Typ *NewsSnippetBuilder* und nicht *NewsSnippetFromTopicBuilder* bzw. *NewsSnippetChangeBuilder*. Nach dem ersten Methodenaufruf ist die DSL auf die allgemeine Signatur des abstrakten Erbauers beschränkt. Damit der abstrakte Erbauer in seinen DSL-Methoden den konkreten Typ der Subklasse zurückgeben kann, wird er mit einem Typ-Parameter versehen (Listing 15).

Durch diesen kleinen Kunstgriff haben alle DSL-Methoden, unabhängig davon, ob sie im allgemeinen oder im speziellen Erbauer definiert sind, den speziellen Erbauer als Rückgabetypen, was die Definition erweiterbarer DSL-Baukästen mit generischen Erbauern und beliebigen Spezialisierungen erlaubt.

Fazit

Für die Definition von DSLs auf der Java-Plattform werden heute aufgrund ihrer Flexibilität oft Sprachen wie Groovy oder Scala eingesetzt. Anhand eines praktischen Beispiels wurde gezeigt, dass sich selbst mit der etwas in die Jahre gekommenen Sprache Java leistungsfähige DSLs entwickeln lassen, die unter Zuhilfenahme von Generics sogar zu DSL-Baukästen ausgebaut werden können.

Konstantin Diener ist Leading Consultant bei der COINOR AG und im Bereich Wertpapierprozesse tätig. Er beschäftigt sich seit über zehn Jahren mit der Java-Plattform, und sein aktuelles Interesse gilt vor allem Business-Rules-Management-Systemen und Domain Specific Languages sowie agilen Methodiken wie Scrum oder Kanban.

Valentino Pola ist Senior Expert Consultant bei der COINOR AG und im Bereich der Architektur und Entwicklung von Anwendungssystemen zur Abbildung von Bankprozessen tätig. Technologisch beschäftigt er sich mit verschiedenen Technologien der JEE-Plattform, der Einführung von BRM-Systemen in bestehende Unternehmensarchitekturen sowie der Konzeption und Umsetzung von Schnittstellenlandschaften.

Links & Literatur

[1] http://www.martinfowler.com/bliki/FluentInterface.html

[2] Gamma, Erich et al.: „Design Patterns – Elements of Reusable Object-Oriented Software“, Pearson Education, 1995

[3] Fowler, Martin: „Domain Specific Languages“, Addison-Wesley Pearson Education, 2010

[4] http://martinfowler.com/dslCatalog/

[5] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/builder/RouteBuilder.html