Völlig beschränkt

Objektvalidierung mit Constraints

Achim Demelt

Allen datenverarbeitenden Anwendungen liegt eine Prämisse zu Grunde: Die Daten müssen korrekt sein. Die Sicherstellung der Datenkonsistenz ist eine zentrale Aufgabe beim Entwickeln solcher Anwendungen. Das Constraint-Konzept bietet hierbei Unterstützung. Der Artikel beschreibt das Konzept der Gültigkeitsprüfungen im Rahmen des pleXX-Frameworks.

Das Geschäftmodell ist der Kern einer jeden Anwendung, die klassische Datenverarbeitung betreibt. Es definiert die Geschäftsobjekte, deren Attribute und die Beziehungen der Objekte untereinander. Das Modell allein ist jedoch in den meisten Fällen nicht ausreichend. Vielmehr müssen Pflichtfelder gesetzt sein oder Wertebereiche eingehalten werden. Manche Objektkonstellationen sind außerdem nur unter bestimmten Voraussetzungen gültig.

Jede solche Rahmenbedingung kann formal als Geschäftsregel (*Business Rule*) formuliert werden. Eine Geschäftsregel macht Aussagen darüber, unter welchen Bedingungen ein Geschäftsmodell (bzw. Teile desselben) korrekt ist. Der Begriff *Geschäftsregel* beschreibt also ein Konsistenzkriterium für das Objektnetz. Sie trifft eine ja/nein-Aussage bezüglich der Korrektheit der Objekte, basierend auf der Analyse der Objektzustände. Im Unterschied zu Geschäfts*prozessen* greift eine Geschäftsregel nicht manipulierend auf den Objektgraphen zu.

Betrachten wir die Situation an einem einfachen Beispiel. Abbildung 1 zeigt ein Geschäftsmodell, mit dem Personen verwaltet werden können. Jede Person kann beliebig viele Adressen besitzen. Eine Adresse kann als Hauptwohnsitz gekennzeichnet sein. Außerdem wird die Beziehung zum Ehepartner mit den Business-Methoden `heirate()` und `trenne()` verwaltet.

Gültigkeitsprüfungen

Ohne Geschäftsregeln ist das Modell fachlich unvollständig. Sicherlich sollte doch jede Person einen Namen haben, der aus einem oder mehreren Buchstaben besteht. Jedoch darf eine Maximallänge auch nicht überschritten werden, da die Datenbankspalte eine Obergrenze vorgibt. Diese und auch alle anderen zu prüfenden Regeln sind in Tabelle 1 zusammengefasst.

In dialogorientierten Applikationen werden Prüfungen für Mindest- und Maximallängen von Zeichenketten oftmals direkt in der Eingabemaske verankert. Beim Drücken des OK-Buttons werden die Textfelder analysiert und der Benutzer wird auf Falscheingaben hingewiesen. Kein ungültiger Wert verlässt so die Oberfläche und wandert in die Datenbank.

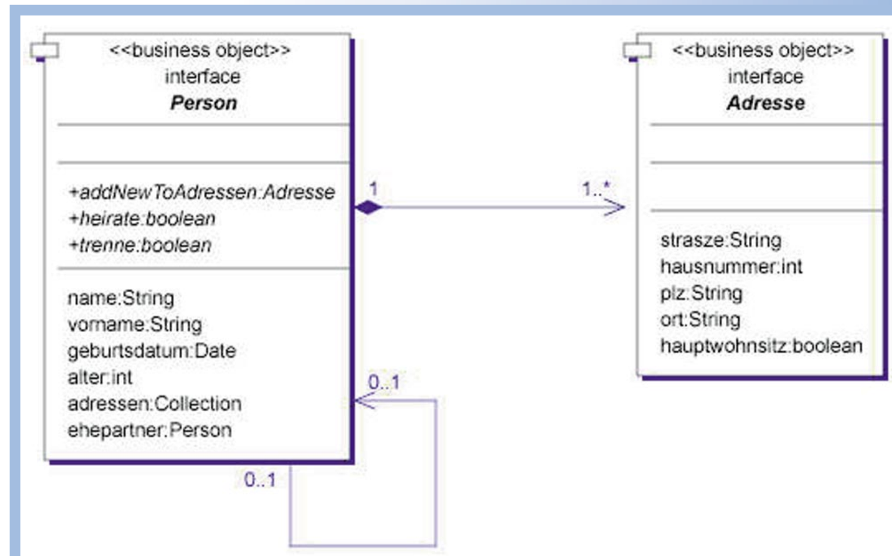

Abbildung 1: Ein einfaches Geschäftsmodell

Wenn die Dialoge die einzige Quelle für die Daten sind, so ist diese Strategie in den meisten Fällen ausreichend. Sobald jedoch Daten desselben Geschäftsobjekts (z. B. Adressen) in mehreren Dialogen gepflegt werden, ist große Vorsicht geboten, um nicht die eine oder andere Validierung zu übersehen. Aussichtslos wird die Lage, wenn die Daten nun nicht mehr nur vom Benutzer über Dialoge eingepflegt werden, sondern auch beispielsweise durch Imports aus anderen Quellen regelmäßig automatisch erzeugt und aktualisiert werden.

Der kanonische Ansatz für solche Prüfungen ist, die Werte in den `set`-Methoden auf Gültigkeit zu untersuchen. Entspricht der Parameter nicht den Kriterien, so wird dies mit einem speziellen Rückgabewert oder einer Exception quittiert. Eine typische Implementierung, wie sie jeder Entwickler sicherlich in ähnlicher Form bereits selbst geschrieben hat, ist in Listing 1 zu sehen.

Der Code ist nicht falsch. Nachteilig ist allerdings die Tatsache, dass dieselbe Logik zur Überprüfung in ähnlicher Form an zahlreichen anderen Stellen ebenfalls zu finden sein wird. Allein in obigem kleinen Beispiel noch bei Vorname, Straße, Postleitzahl und Ort. Als Lösung bietet sich an, diesen Überprüfungscode an eine zentrale Stelle auszulagern, etwa in eine Hilfsklasse als statische Methode.

Ein Framework

Besser wäre, diesen Code überhaupt nicht mehr programmieren zu müssen. Stattdessen sollte lediglich genau einmal an einer Stelle *definiert* werden, dass eine Postleitzahl aus fünf Ziffern zu bestehen hat, und der Vorname mindestens einen und maximal 64 Buchstaben besitzen muss. Diesen Komfort kann nur ein Framework bieten. Dieses Framework muss die Definition der Geschäftsregeln auswerten und die entsprechenden Prüfungen beim Setzen der Attribute durchführen.

Der restliche Teil des Artikels beschreibt das Konzept der Gültigkeitsprüfungen im Rahmen des *pleXX*-Frameworks [Dem03]. Der Fokus des Frameworks liegt auf Anwendungen, die ihre Daten mit herkömmlichen O/R Mappern in relationalen Datenbanken persistieren, es bietet darüber hinaus jedoch noch weitere Dienste. Es besteht aus einem Codegenerator und einer Laufzeitumgebung. Das Geschäftsmodell wird mit Java-Interfaces modelliert, aus welchen der Generator konkrete Implementierungen erzeugt. Dieser generierte Code enthält unter anderem die Verknüpfung zu den im Modell hinterlegten Gültigkeitsprüfungen. Die zu prüfenden Geschäftsregeln werden im Kontext von pleXX *Constraints* („Einschränkungen") genannt.

Klasse	Attribut	Regel(n)
Person	Name	* Pflichtfeld * Länge zwischen einem und 64 Zeichen
Person	Vorname	* Pflichtfeld * Länge zwischen einem und 64 Zeichen
Person	Geburtsdatum	* Pflichtfeld * Nach dem 01.01.1900, vor dem aktuellen Datum
Person	Adressen	* Mindestens eine Adresse * Genau ein Hauptwohnsitz
Person	Ehepartner	* Maximal eine Person * Nicht Ehepartner von sich selbst
Adresse	Straße	* Pflichtfeld * Länge zwischen einem und 64 Zeichen
Adresse	Hausnummer	* Pflichtfeld * Wert größer gleich eins
Adresse	Postleitzahl	* Pflichtfeld * Genau fünf Ziffern
Adresse	Ort	* Pflichtfeld * Länge zwischen einem und 64 Zeichen
Adresse	Hauptwohnsitz	* Genau einer für alle Adressen einer Person

Tabelle 1: Geschäftsregeln des Beispielmodells

Granularität

Auch andere Frameworks bieten Möglichkeiten zur Überprüfung der Geschäftsregeln. Meist werden diese Prüfungen am Geschäftsobjekt als Ganzem, also an der Klasse, festgemacht. Pro Klasse gibt es eine definierte Methode, die vom Framework an bestimmten Stellen aufgerufen wird, um eine Objektinstanz auf Gültigkeit zu überprüfen.

PleXX verfolgt hier einen anderen Ansatz. Statt alle Prüfungen in einer einzigen Methode pro Geschäftsobjekt zu sammeln, wird jede einzelne Prüfung in eine eigene Klasse extrahiert. Die entstehenden Klassen sind sehr klein, da jede für sich nur einen Aspekt der Geschäftslogik darstellt. Sie lassen sich also direkt mit den formulierten Geschäftsregeln in Verbindung bringen. Jede einzelne Klasse für sich ist überschaubar und von den anderen Prüfungen gekapselt. Sie stellt somit eine optimal wartbare Einheit dar.

Diese Constraint-Klassen werden nicht mit dem gesamten Geschäftsobjekt verknüpft, sondern immer nur mit einzelnen Attributen bzw. Relationen. Die Prüfung zur syntaktischen Korrektheit einer E-Mail-Adresse wird nur dem Attribut `email` zugeordnet, während die Regel, dass eine Person vor ihrem Todestag geboren sein muss, an den Attributen `geburtstag` und `todestag` hängt. Das Framework kann nun dafür sorgen, dass Prüfungen wirklich nur dann ausgeführt werden, wenn sich ein für die Prüfung relevantes Attribut geändert hat. Natürlich kann dieselbe Constraint-Klasse auch mit mehreren Attributen in unterschiedlichen Objekten verknüpft werden.

```
public void setName(String name) {
  if (name == null) {
    throw new IllegalArgumentException("...");
  }
  if (name.length() < 1 || name.length() > 64) {
    throw new IllegalArgumentException("...");
  }

  this.name = name;
}
```

Listing 1: Typische Überprüfung für gültige Werte

Das Konzept

Ein Constraint in pleXX ist lediglich definiert durch ein Interface mit einer einzigen Methode:

```
checkAttribute(Object attributeOwner,
  String attributeName, Object attributeValue)
```

Genauer gesagt handelt es sich hierbei um einen *Attribut*-Constraint, da er für die Überprüfung genau eines Attributwerts zuständig ist (und dabei wiederum nur für einen Aspekt).

Konkrete Constraint-Klassen für Geschäftsregeln implementieren dieses Interface und führen die gewünschte Überprüfung durch. Ein solcher Constraint muss im Modell mit den betroffenen Attributen verknüpft werden. Bei der Änderung des Attributwerts sorgt die pleXX-Laufzeitumgebung dafür, dass die `checkAttribute()`-Methode aufgerufen wird. Bei einem Verstoß gegen die Geschäftsregel muss dort eine `ConstraintViolationException` geworfen werden. Es handelt sich hierbei um eine `RuntimeException`, die also nicht in einer `throws`-Klausel deklariert werden muss.

PleXX bietet bereits einige Standard-Implementierungen für häufig auftretende Constraints. So prüft beispielsweise der `MandatoryConstraint`, ob ein Attributwert gesetzt ist. Es gibt ferner den `MinMaxLengthConstraint`, der die Länge einer Zeichenkette prüft, den `MinMaxValueConstraint`, der die Einhaltung von Wertgrenzen forciert, und den `RegularExpressionConstraint`, der nur Werte zulässt, die einem gegebenen regulären Ausdruck genügen.

Analog zum Attribut-Constraint gibt es einen *Beziehungs*-Constraint mit `checkLink()`- und `checkUnlink()`-Methoden. Ähnlich parametrisiert werden die Prüfungen beim Knüpfen bzw. Lösen einer Objektrelation aufgerufen.

Einfache Prüfungen

Wie sieht das Konzept nun in der Praxis aus? Betrachten wir zunächst einfache Attributprüfungen. In vielen Fällen handelt es sich dabei um Standardfälle, die in nahezu jeder Anwendung vorhanden sind. Auch im Beispielmodell sind zahlreiche solcher Prüfungen durchzuführen: Pflichtfelder, Längen von Zeichenketten, erlaubte Wertebereiche. Die Listings 2 und 3 zeigen die Definition solcher Standard-Constraints mit pleXX für die Geschäftsobjekte `Person` und `Adresse`.

```
/** @stereotype business object */
public interface Person {
  /**
   * @mandatory
   * @minMaxLength 1, 64
   */
  String getName();

  void setName(String name);

  /**
   * @mandatory
   * @minMaxLength 1, 64
   */
  String getVorname();

  void setVorname(String vorname);

  /**
   * @mandatory
   * @minMaxValue new java.util.GregorianCalendar(
   *   1900, 1 - 1, 1).getTime(), new java.util.Date()
   */
  Date getGeburtsdatum();

  void setGeburtsdatum(Date geburtsdatum);

  /**
   * @associates Person
   * @supplierCardinality 0..1
   * @clientCardinality 0..1
   * @constraint SelbstHeiratConstraint
   */
  Person getEhepartner();

  // ...
}
```

Listing 2: Definition einfacher Attributprüfungen im Geschäftsobjekt Person

```
/** @stereotype business object */
public interface Adresse {
  /**
   * @mandatory
   * @minMaxLength 1, 64
   */
  String getStrasze();

  void setStrasze(String strasze);

  /**
   * @minMaxValue new Integer(1), null
   */
  int getHausnummer();

  void setHausnummer(int hausnummer);

  /**
   * @mandatory
   * @regExp [0-9]{5}
   */
  String getPlz();

  void setPlz(String plz);

  /**
   * @mandatory
   * @minMaxLength 1, 64
   */
  String getOrt();

  void setOrt(String ort);

  boolean isHauptwohnsitz();

  void setHauptwohnsitz(boolean hauptwohnsitz);
}
```

Listing 3: Definition einfacher Attributprüfungen im Geschäftsobjekt Adresse

Der Unterschied zu einem @regExp „herkömmlichen" Java-Interface liegt lediglich in einigen wenigen Zeilen JavaDoc ähnlicher Kommentare vor den get-Methoden der Attribute. Sie sind im Listing durch Fettschrift hervorgehoben. Recht häufig ist @mandatory zu sehen. Das bedeutet, dass das Attribut keinen null-Wert besitzen darf, und entspricht somit einem NOT NULL-Constraint in der Datenbank. Letzterer wäre jedoch für die Anwendung nicht vernünftig auswertbar, denn beim Einfügen oder Aktualisieren einer Zeile in der DB würde ein NULL-Wert mit einem unverständlichen Datenbankfehlercode quittiert. Findet die Prüfung bereits im Anwendungscode statt, können Fehler dort besser behandelt werden. Und genau diese Prüfung in Form eines MandatoryConstraints wird im Anwendungscode durch den pleXX-Generator auf Grund des @mandatory-Tags eingefügt.

Daneben findet man RegExp- im Beispiel noch die @minMaxLength-, @minMaxValue- und Constraints, die den oben vorgestellten Standard-Constraints entsprechen. Damit sind bereits nahezu alle in Tabelle 1 geforderten Geschäftsregeln umgesetzt. Es fehlt jedoch noch die Einschränkung, dass eine Person sich nicht selbst heiraten darf. Dieser fachspezifische Constraint wird in einer eigenen Klasse SelbstHeiratConstraint (s. Listing 4) implementiert, welche mit dem @constraint-Tag dem Ehepartner-Attribut zugeordnet ist.

Das Vorgehen ist also recht einfach: Die meisten Geschäftsregeln werden durch Standard-Constraints abgedeckt und können mit speziellen Tags in die Geschäftsobjekte eingebaut werden. Für alle anderen Fälle wird jeweils eine eigene Klasse erstellt, die ein einfaches Interface implementiert. Diese Klasse wird mit dem @constraint-Tag dem entsprechenden Attribut bzw. der Relation assoziiert.

Komplexere Prüfungen

Nicht alle Geschäftsregeln sind so einfach wie die Validierung einzelner unabhängiger Attributwerte. In obigem Beispielmodell wird der Hauptwohnsitz einer Person mit einem Flag gekennzeichnet. Die Geschäftsregel besagt jedoch, dass eine Person immer nur genau einen Hauptwohnsitz haben darf.

```
public class SelbstHeiratConstraint
implements AttributeConstraint {
  public ConstraintViolationException checkAttribute(
        Object attributeOwner,
        String attributeName,
        Object attributeValue) {
    if (attributeOwner == attributeValue) {
      throw new ConstraintViolationException("...");
    }

    return null;
  }
}
```

Listing 4: Implementierung des SelbstHeiratConstraints

Hier ist Vorsicht geboten. Es gibt nun zwei Stellen, an denen diese Bedingung verletzt werden kann. Zum einen direkt beim Ändern des Hauptwohnsitzkennzeichens und zum anderen beim Hinzufügen oder Löschen einer Adresse zu einer Person. Ist dieser Umstand bekannt, so ist die Lösung ähnlich einfach wie für unabhängige Attributwerte: Derselbe Constraint muss eben beiden Stellen zugewiesen werden.

Sobald ein Constraint für mehrere Attribute oder Relationen zuständig ist, muss unterschieden werden zwischen dem *Kontext* und den *Auslösern* (*Trigger*) der Prüfung. Der Kontext ist das Objekt bzw. das Attribut, in dessen Rahmen die Prüfung ablaufen soll. Die checkAttribute()-Methode des Constraints erhält also diesen Kontext zur Prüfung. Es gibt für jeden fachlichen Constraint immer genau einen Kontext. Alle anderen für die Prüfung relevanten Stellen sind Trigger, also Auslöser, der Prüfung.

Der Kontext sollte so gewählt werden, dass die Durchführung der Validierung möglichst einfach gestaltet werden kann. Im Beispiel ist das offensichtlich das Geschäftsobjekt Person, denn von dort hat man direkten Zugriff auf all dessen Adressen. Der entsprechende Constraint ist in Listing 5 abgebildet. Obwohl der Kontext eigentlich eine Beziehung ist (die Adressen einer Person), kann dennoch ein AttributeConstraint verwendet werden. In Java liegt ja letztendlich jeder zu-n-Beziehung ein Attribut vom Typ Collection zu Grunde. Genau dieses wird auch als attributeValue an die Constraint-Prüfung übergeben.

```
public class EinHauptwohnsitzConstraint
implements AttributeConstraint {
  public ConstraintViolationException checkAttribute(
        Object attributeOwner,
        String attributeName,
        Object attributeValue) {
    Collection adressen = (Collection)attributeValue;

    boolean hauptwohnsitzGefunden = false;
    Iterator ait = adressen.iterator();
    while (ait.hasNext()) {
      Adresse a = (Adresse)ait.next();
      if (a.isHauptwohnsitz()) {
        if (hauptwohnsitzGefunden) {
          throw new ConstraintViolationException("...");
        }
        hauptwohnsitzGefunden = true;
      }
    }

    if (!hauptwohnsitzGefunden) {
      throw new ConstraintViolationException("...");
    }
    return null;
  }
}
```

Listing 5: Implementierung des EinHauptwohnsitzConstraints

Nun muss dem Framework noch mitgeteilt werden, welches Attribut der Kontext ist, und welche die Trigger sind. Der Kontext bedarf keiner weiteren Kennzeichnung. Es genügt der oben vorgestellte `@constraint`-Tag mit der Angabe des Constraint-Klassennamens. Auch ein Trigger wird mit diesem Tag gekennzeichnet. Jedoch muss dem Klassennamen der Navigationsweg vom Trigger zum Kontext nachgestellt werden. Dafür werden die regulären Attribut- und Relationsnamen der Geschäftsobjekte verwendet, die durch Punkte aneinandergehängt werden. Der gesamte Weg wird in geschweifte Klammern gesetzt. Beispielsweise definiert der folgende Ausdruck einen Navigationsweg:

```
{ehepartner.offeneRechnungen.rechnungsposten.artikel}
```

Was passiert aber, wenn wie in unserem Beispiel keine Navigation vom Trigger (dem Hauptwohnsitzkennzeichen) zum Kontext (der Adressenliste einer Person) definiert ist? Es gibt ja lediglich den Weg von der Person zu den Adressen, aber nicht umgekehrt. Hierfür bietet das pleXX-Framework jedoch die so genannten *Backlinks*, also entgegengesetzte Objektbeziehungen. Um den negativen Weg von einer Person über deren Adressenliste zu navigieren, kann `{personAdressenBacklink}` verwendet werden. Da der Navigationsweg für einen Trigger immer bei einem Kontext enden muss, sieht der vollständige Navigationsausdruck wie folgt aus: `{personAdressenBacklink.adressen}`.

Zeitpunkt der Prüfung

Bis zu diesem Punkt ist bereits sehr viel erreicht: Neben Prüfungen für einzelne Attribute können auch kontextbezogene Prüfungen durchgeführt werden, die viele verschiedene Auslöser im gesamten Objektmodell besitzen. Doch leider hat die bisher vorgestellte Lösung für das Hauptwohnsitzproblem einen entscheidenden Haken. Die Prüfung der Constraints wird immer sofort bei der Durchführung der Änderung, also z. B. beim Setzen des Attributwerts, aufgerufen. Damit ist ein Wechsel des Hauptwohnsitzes in obigem Modell nicht möglich, denn dafür sind zwei Schritte notwendig. Erstens muss der alte Hauptwohnsitz gelöscht bzw. sein Hauptwohnsitzkennzeichen entfernt werden. Und zweitens muss die neue Adresse erzeugt und dort das Kennzeichen gesetzt werden.

Egal welcher Schritt zuerst ausgeführt wird, das Objektmodell hat danach einen ungültigen Zustand. Beim Löschen der alten Adresse hat die Person *keinen* Hauptwohnsitz mehr. Beim Erzeugen der neuen Adresse und dem Setzen des Hauptwohnsitzflags hat sie *zwei*. In beiden Fällen wird der `EinHauptwohnsitzConstraint` die Verletzung einer Geschäftsregel anzeigen und die Aktion verweigern.

Unabhängig von der Reihenfolge der Manipulationen wird die Aktion also nicht von Erfolg gekrönt sein. Die Prüfung des Constraints darf nicht sofort bei der Änderung der Objekte durchgeführt werden, weil kurzfristig immer inkorrekte Zustände entstehen. Dennoch muss garantiert sein, dass der persistente Datenbestand immer nur einen Hauptwohnsitz pro Person aufweist.

Die beiden Schlagworte sind nun bereits gefallen: *sofort* und *persistent*. Solange die veränderten Daten noch nicht in der Datenbank gespeichert (also noch nicht persistent) sind, können vorübergehend inkorrekte Zustände akzeptiert werden. Spätestens jedoch vor dem Schreiben der Daten (also beim `commit()`) müssen die Constraints geprüft werden. Dieses verzögerte Prüfen wird mit dem Tag `@deferredConstraint` angezeigt. Er wird analog zum regulären `@constraint`-Tag verwendet, zeigt eben lediglich an, dass die Prüfung nicht sofort, sondern später durchgeführt werden soll. Listing 6 zeigt die Einbindung der Hauptwohnsitzprüfung in die Geschäftsobjekte.

```
public interface Person {
  ...
  /**
   * @associates Adresse
   * @link aggregationByValue
   * @clientCardinality 1
   * @supplierCardinality *
   * @deferredConstraint EinHauptwohnsitzConstraint
   */
  Collection getAdressen();
  ...
}

public interface Adresse {
  ...
  /**
   * @deferredConstraint EinHauptwohnsitzConstraint
     {personAdressenBacklink.adressen}
   */
  boolean isHauptwohnsitz();
  ...
}
```

Listing 6: Anwendung des EinHauptwohnsitzConstraints

Fazit

Das Constraint-Konzept des pleXX-Frameworks bietet eine flexible Lösung zur Überprüfung von Geschäftsregeln. Anders als bei vielen anderen Frameworks sind die einzelnen Geschäftsregeln hier klar gekapselt und nicht mit anderer Geschäftslogik oder gar dem Oberflächencode vermischt. Die Verknüpfung der Constraints zu den zu prüfenden Daten erfolgt deklarativ direkt im Geschäftsmodell.

Neben einfachen Prüfungen für einzelne, unabhängige Attribute können auch komplexe Zusammenhänge zwischen mehreren Attributen verschiedener Objekte mit kontextbezogenen Constraints definiert werden.

Die konkrete Durchführung der Überprüfungen wird von der Laufzeitumgebung angestoßen. Dabei besteht die Wahl zwischen sofortiger und verzögerter Prüfung. Während erstere häufig bei einfachen Attributprüfungen Verwendung findet, sind komplexere Prüfungen über mehrere Objekte ohne die letztere Möglichkeit oft gar nicht durchführbar.

Das Konzept hat sich im Produktiveinsatz sowohl in Web-Anwendungen als auch in Rich-(Swing)-Client-Applikationen bereits in mehreren Projekten bewährt, darunter ein Großprojekt mit weit mehr als 100 Geschäftsobjekten. Es handelt sich dabei um die Neuentwicklung eines Abfallgebührenveranla-

gungssystems. Die Überprüfungen von Geschäftsregeln enthält hier eine weitere Komplexitätsebene, da nahezu jeder Datensatz in diesem Fachgebiet eine zeitliche Komponente besitzt. So ändern sich beispielsweise die geltenden Gebührenverordnungen jedes Jahr, abrechnungsrelevante Personen ziehen zu beliebigen Zeitpunkten ein und aus, oder ein Bürger beantragt für den kommenden Monat ein größeres (und damit teureres) Müllgefäß.

Das pleXX-Framework unterstützt auch diese temporale Datenspeicherung. Ein Folgeartikel beschreibt die dort zu Grunde liegenden Konzepte, und was dabei für die Modellierung und Programmierung zu beachten ist.

Literatur

[Dem03] A. Demelt, Ein Framework für Business-Objekte, in: JavaSPEKTRUM, 1, 2003

Dipl-Inf. (FH) Achim Demelt ist Softwarearchitekt und Projektleiter bei der eXXcellent solutions GmbH in Ulm. Einer seiner Schwerpunkte ist die Steigerung der Effizienz bei der Implementierung datengetriebener Anwendungen in Java.
E-Mail: a.demelt@exxcellent.de.

Weiterführende Informationsquellen http://www.exxcellent.de