Model Driven Development mit XDoclet und Annotations

Der heilen Welt so nah

VON BERND RÜCKER

Dieser Artikel beschreibt das pragmatische Herangehen in einem Reengineering-Projekt mit Model Driven Development (MDD) durch Java-Modelle mit Tags und einem XDoclet-Code-Generator. Mit dieser Lösung lassen sich schnell Geschäftsanwendungen umsetzen, saubere Architekturen erzielen und die Komplexität von J2EE wird beherrschbar.

In einem Kundenprojekt führten wir ein Reengineering einer bestehenden J2EE-Warenwirtschaft durch, wobei die zugrunde liegenden Geschäftsmodelle möglichst unverändert übernommen werden sollten. Der Kunde hatte schlechte Erfahrungen mit J2EE gemacht, da die Komplexität der Architektur die Anwendung instabil machte. Eine nahe liegende Lösung war nun die Nutzung der Model Driven Architecture (MDA). Nach einem Prototyp mit AndroMDA haben wir uns bei diesem Projekt jedoch für die pragmatischere Variante Model Driven Development (MDD) entschieden.

Dazu haben wir viel Arbeit in die eigentliche Architektur gesteckt und darauf aufbauend Code-Generatoren für den notwendigen technischen Code selbst entwickelt. Notwendige Metainformationen haben wir in ein einfaches Java-Modell gepackt. So konnten wir auch das Geschäftsmodell der alten Anwendung zu großen Teilen eins zu eins wieder verwenden. Beim Projekt selbst ist nebenbei ein Framework entstanden, welches Grundlage vielfältiger Geschäftsanwendungen sein kann. Dieser Artikel beschreibt neben der grundsätzlichen Architektur vor allem das Vorgehen mit dem XDoclet-Code-Generator.

Architektur

Bei unserer neuen Architektur steht das Geschäftsmodell im Vordergrund. Dabei setzen wir ein einfaches Java-Modell (POJOs – Plain Old Java Objects) ein und können den vollen Umfang der Objektorientierung und alle Java-Features nutzen. Als Persistenztechnik fiel die Wahl auf JDO, wobei die Implementierung Kodo von SolarMetric [1] zum Einsatz kam. Die Oberfläche wurde hauptsächlich in Java-Swing entwickelt. Im Produktivbetrieb verrichten JBoss [2] als Application Server und MySQL [3] als Datenbank ihre Dienste.

Um nun die Vorteile einer J2EE-Umgebung wie Clustering oder Transaktionskontrolle nutzen zu können, muss die Anwendung zumindest in die klassischen drei Schichten aufgeteilt werden: Client, Server und Datenhaltung. Die Datenhaltungsschicht wird komplett von JDO übernommen und braucht nicht weiter betrachtet werden. Bekannte J2EE Patterns [4] helfen bei den verbleibenden Schichten, trotz steigender Komplexität den Überblick zu behalten.

Abbildung 1 zeigt unsere Architektur im Überblick und im Folgenden werden die einzelnen Design Patterns, die zum Einsatz kamen, beschrieben:

- *DataTransferObject (DTO)*: DTO ermöglichte es uns, den Client unabhängig vom POJO-Modell zu halten. Zwar könnten wir auch die Modellklassen direkt an den Client weitergeben, da sie normale Java-Klassen sind. Da JDO jedoch den Bytecode der Klassen verändert, um Persistenz zu implementieren, wäre in diesem Fall der Client abhängig von JDO-Laufzeitbibliotheken geworden. Dies konnten wir durch den Einsatz der DTOs vermeiden.
- *DTOAssembler:* Der Assembler ist dafür zuständig, DTOs aus Modellobjekten zu erstellen bzw. bestehende Modellobjekte anhand der DTO-Daten zu verändern.
- *BusinessDelegate:* Das Muster stellt die einzige sichtbare Schnittstelle zwischen Client und Server dar. Damit sind Client- und Server-Komponenten ideal entkoppelt (der Client muss nur die DTOs und das BusinessDelegate kennen).
- *SessionFacade:* Die SessionFacade ist das Gegenstück zum Business Delegate auf Serverseite. Sie nimmt die Aufrufe vom Client entgegen und leitet diese an die entsprechenden Services (siehe unten) weiter. Sie ist als Stateless Session Bean realisiert und wird daher durch den Application Server konfiguriert.

Für Services einer Komponente haben wir eigene Konzepte entwickelt:

- *CRUD Services:* Für einige Geschäftsobjekte, die eine eigenständige fachliche Identität besitzen, werden Methoden zum Erzeugen (Create), Abfragen (Read), Ändern (Update) und Löschen (Delete) bereitgestellt.

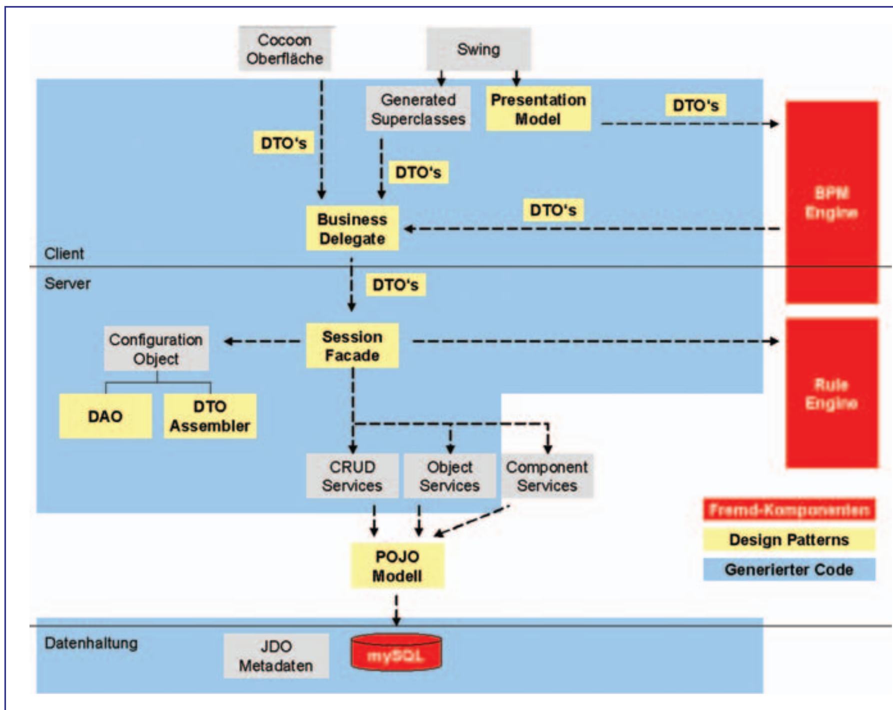

Abb. 1: Die Architektur im Überblick

- *Object Services:* Geschäftsobjekte können Methoden anbieten, die Clients zur Verfügung gestellt werden sollen. Diese Methoden werden in Services gekapselt, die dem Client einen Aufruf der Methode ermöglichen. Der Service sucht dabei zuerst das Zielobjekt, um dann die Methode auf ihm aufzurufen. Der Ablauf ist beispielhaft in Abbildung 2 dargestellt.
- *Component Services:* Meist gibt es Anwendungsfälle, die es erfordern, dass viele (verschiedene) Geschäftsobjekte verarbeitet werden müssen. Da der notwendige Java-Code nicht in einem einzelnen Geschäftsobjekt untergebracht werden kann (da mehrere, evtl. unabhängige Objekte verarbeitet werden) gibt es diese Art von Services.

Services sind als einfache Java-Klassen (POJO) implementiert, denen ein ConfigurationObject übergeben wird (dieses Prinzip nennt sich Dependency Injection und wurde vor allem durch das Spring Framework [6] bekannt). Dieses Objekt bietet Ihnen Zugriff auf das DAO oder andere benötigte Objekte. Alle Methoden in Serviceobjekten werden im BusinessDelegate veröffentlicht.

Ein großer Vorteil, der durch POJOs als Services entsteht, ist, dass die Services nicht nur in einer J2EE-Umgebung funktionieren. Für unsere Tests generieren wir beispielsweise auch eine BusinessDelegate-Implementierung, welche die Services direkt aufruft und ein ConfigurationObject übergibt, das ein DAO für eine lokale hsqldb [5] zurückliefert.

Probleme der Patterns

Die Entwicklung mit J2EE und Design Patterns hat nun leider nicht nur Vorteile. Denn gerade durch Patterns wie DataTransferObject oder BusinessDelegate wird viel Code benötigt, der keinerlei fachlichen Zwecken dient. Dieser Code ist nicht nur zeitaufwendig zu entwickeln, sondern es ist vor allem eine sehr anspruchslose, demotivierende Arbeit, an der jeder Entwickler schnell die Freude verliert. Wir mussten feststellen, dass man auf die Dauer selbst kleine Code-Änderungen an der Anwendung scheut, da dies oft einen Rattenschwanz an Änderungen im technischen Code nach sich zieht.

Ein weiteres Problem, welches wir in der ersten Version der Anwendung identifizierten, war, dass es vor allem in Swing kaum verbreitete Patterns für Standard-

Anzeige

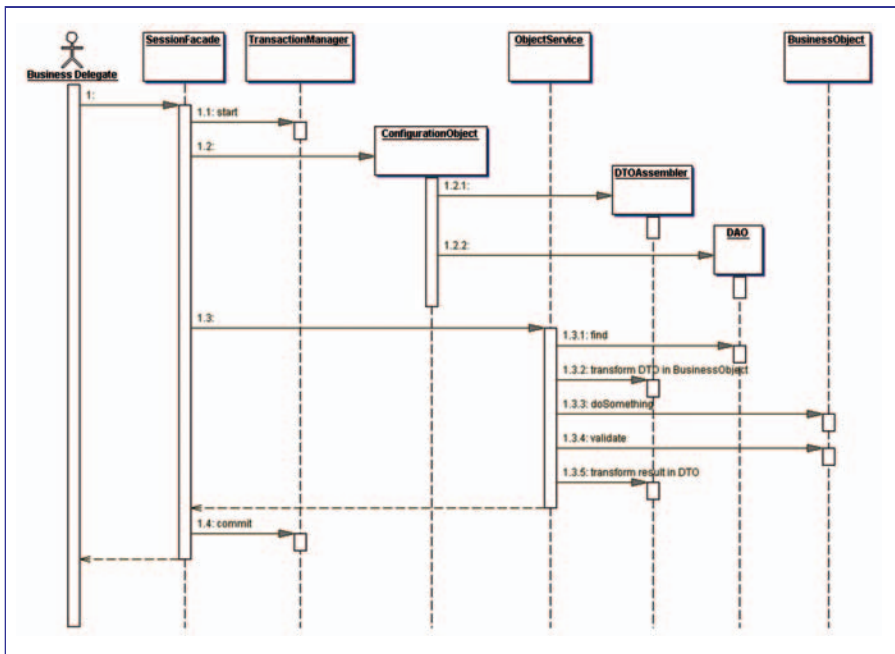

Abb. 2: Sequenzdiagramm eines Object Service

probleme, wie beispielsweise Data Binding, gibt. Dies führte bei uns dazu, dass der GUI-Code teilweise sehr schlecht wartbar war.

Lösungsansatz MDA

MDA (Model Driven Architecture) adressiert nun die angesprochenen Probleme. Es werden die fachlichen Aspekte der Anwendung technologieunabhängig entwickelt, der notwendige technische Infrastruktur-Code wird vollständig generiert. Dabei liegen dann fachliche Informationen nur an einer Stelle: im Modell.

In unserem Projekt hatten wir einen Prototyp mit AndroMDA [7] erstellt, uns aber dann gegen diese Lösung entschieden. Denn einerseits hatten wir spontan Probleme mit dem benötigten XMI-Code unseres Tools Borland Together und andererseits konnten wir den XMI-Export nicht mit Ant automatisieren, sodass sich unser Build-Prozess verkomplizierte.

Lösungsansatz MDD

Model Driven Development (MDD) verwirklicht nach unserem Verständnis die gleichen Ideen wie MDA: Informationen im Modell unterbringen und technischen Code generieren. MDD konnte uns im Projekt dabei wesentlich mehr begeistern, denn es geht deutlich pragmatischer von einem Modell in einer Programmiersprache, in unserem Fall Java, aus. In dieses Modell werden zusätzliche Informationen hinterlegt, in Java geschieht dies am einfachsten durch so genannte Tags, bekannt von Javadoc Tags (Listing 1). Tabelle 1 zeigt einige der von uns verwendeten Tags. In unserem Projekt haben wir außerdem Borland Together entsprechend angepasst, sodass bestimmte Tags zu farblichen Hervorhebungen im UML-Modell führen (Abb. 3) und wir die Tags über ein eigenes Property-Panel pflegen können (Abb. 4).

In Java 5 wurden Annotations in die Sprache mit aufgenommen, die den Tags prinzipiell sehr ähnlich sind. Java 5 bietet jedoch viel mehr Möglichkeiten als die noch von uns verwendeten XDoclet Tags; ein heute startendes Projekt sollte sich also durchaus mit den Java 5 Annotations beschäftigen.

Wie bei MDA ermöglichen uns eine genau definierte Standardarchitektur und der Einsatz von Design Patterns die Generierung großer Teile des Codes. MDD mit Java und Tags hat unserer Meinung nach folgende Vorteile:

- Java-Modell und Tags sind dem Entwickler vertraut.
- Im Modell stehen alle bekannten objektorientierten Konzepte und Java-Techniken zur Verfügung.
- Tag-Bearbeitung wird von vielen Tools unterstützt.
- Tags ändern nichts am Code.
- Geschäftsmodell ist direkt per JUnit ohne Infrastruktur testbar.

Listing 1

```
/**
 * @stereotype BusinessEntity
 * @primaryBusinessClass
 */
public class ArticleCategory {

  /**
   * @assertion
   * @businessException PROPERTY_DEFINITION_
                                    ALREADY_EXISTS
   */
  public void validate(ErrorList errors) {
  }

  /**
   * @required
   * @unchangeable
   * @includeInDto
   * @businessKey
   */
  private String id;

  /**
   * @directed
   * @supplierRole parent
   * @supplierCardinality 0..1
   * @includeInDto
   * @associationMappingType includeKeys
   */
  private ArticleCategory parent;

  /**
   * @directed
   * @supplierRole children
   * @supplierCardinality 0..*
   * @link aggregationByValue
   * @associates com.camunda.ccs.module.article.
                                    model.ArticleCategory
   * @associationMappingType includeKeys
   * @includeInDto
   */
  private ArrayList children = new ArrayList();
}
```

- Techniken sind bekannt und bewährt und schon heute einsatzbereit.

Was genau kann nun generiert werden? In Abbildung 1 wurden alle generierten Codeteile blau hinterlegt, dies sind unter anderem:

- DTOs und DTO Assembler
- SessionBeans
- BusinessDelegate, mit Implementierung für eine lokale Testvariante und einer Implementierung mit Zugriff auf die SessionBeans
- CRUD und Object Services
- DAOs
- JDO-Metadaten
- Basisklassen für Swing: Diese Basisklassen enthalten bereits an DTOs gebundene Oberflächenkomponenten. Außerdem werden aus Prozessbeschreibungen PresentationModels generiert, die die Funktionalität kapseln. Somit reduziert sich die Oberflächenentwicklung auf das Layout der Fenster. Soll weiterer Code generiert werden, zum Beispiel für die Cocoon- oder eine Struts-Oberfläche, so muss lediglich der Generator erweitert werden.

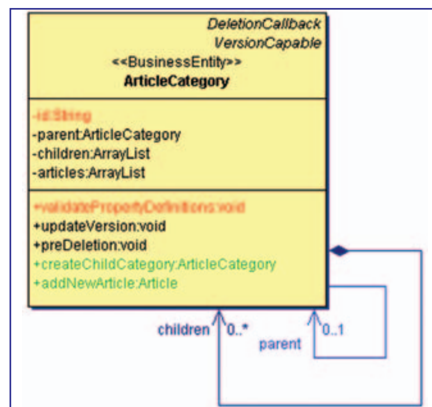

Abb. 3: Modell mit Tags in Borland Together

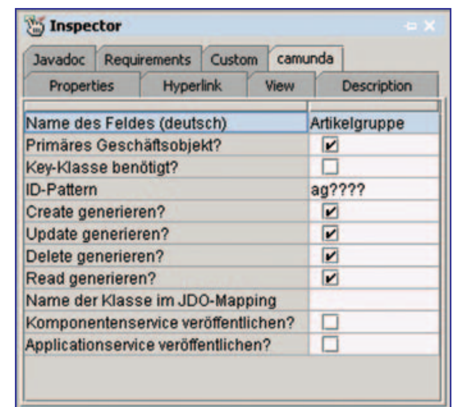

Abb. 4: Inspector Panel für Tags

Tag	Gültiger Kontext	Bedeutung
primaryBusinessClass	Klasse	Primäres Geschäftsobjekt (Für „primäre Geschäftsobjekte gibt es eine Suche, ein Hauptfenster …).
required	Attribut	Das Feld darf nicht *null* sein.
regExp	Attribut	Der Wert des Feldes muss dem angegebenen regulären Ausdruck entsprechen, sonst wird eine Exception geworfen.
businessKey	Attribut	Das Feld gehört zum fachlichen Schlüssel.
includeInDto	Attribut	Das Feld soll im DTO enthalten sein.
associationMappingType	Assoziation	Gibt an, wie eine Assoziation im DTO gemappt werden soll (alle abhängigen Objekte mitliefern, Keys der abhängingen Objekte …).
Assertion	Methode	Die Methode muss zur Validierung des Objektzustandes aufgerufen werden.
businessException	Methode	Die Methode kann die genannte Exception werfen.
provideAsService	Methode	Die Methode wird als Objekt-Service bereitgestellt.

Tabelle 1: Tags für den Generator

Code-Generatoren

Um einen Code-Generator zu entwickeln, sollte man auf ein bestehendes Framework zurückgreifen, um nicht das Rad neu erfinden zu müssen. Ein bekanntes Framework ist XDoclet [9], welches viele zumindest von der Generierung von EJB-Deployment-Deskriptoren her kennen.

XDoclet implementiert hierbei eine Template Engine, die es ermöglicht, Code über eigene Skriptsprachen zu implementieren. Wir haben uns im Projekt für XDoclet entschieden, da es ein bekanntes und verbreitetes Projekt ist, über das es auch qualifizierte Literatur gibt [10]. Zum Zeitpunkt des Projektstarts waren zudem XDoclet2 [11] noch nicht erhältlich und die Zukunft von EMF, dem Eclipse Modeling Framework [12], noch ungewiss.

Bei der Nutzung von XDoclet konnten wir uns auf die Erstellung der Templates konzentrieren. Dabei ist das Vorgehen immer gleich: Zuerst wird der zu ge-

Anzeige

nerierende Code in einem Prototypen manuell erstellt und getestet. Erst wenn er fehlerfrei funktioniert, wird daraus ein Template für den Code-Generator erstellt. Auf diesem Weg behalten die Entwickler Spaß an der Arbeit, da das erstmalige Erstellen des Codes eine Herausforderung ist und der generierte Code durch entsprechend sauberes Testen stabil wird. Um unsere Anforderungen mit XDoclet umzusetzen, mussten wir auch einige Erweiterungen an XDoclet vornehmen und vor allem eigene Tasks schreiben.

Erfahrungen mit XDoclet

Die Entwicklung der XDoclet-Erweiterungen sowie der Templates war aus mehreren Gründen sehr aufwendig. Insgesamt mussten wir sehr viele Tags implementieren, um unsere Anforderungen umzusetzen, wodurch der gesamte Generator etwas unübersichtlich wurde. Dies führte mit den XDoclet Templates zu einem schwer wartbaren Code-Generator. Am meisten hat jedoch gestört, dass kein Debugging möglich war und wir keine Möglichkeit gefunden haben, mit vertretbarem Aufwand Unit-Tests zu schreiben. Somit mussten wir lange „Try and Error"-Zyklen durchlaufen.

Außerdem haben uns die meist nichts sagenden XDoclet Exceptions auf Dauer „in den Wahnsinn" getrieben. Die dadurch notwendigen Ausflüge in den XDoclet-Sourcecode waren keine wirkliche Freude, einen Schönheitspreis haben die XDoclet-Entwickler dafür nicht verdient. Auch haben wir den starken Verdacht, dass einige Methoden unerwartete Seiteneffekte besitzen, denn vereinzelt konnten wir sehr merkwürdiges Verhalten beobachten (z.B. hat XDoclet eine Klasse X korrekt generiert und die inhaltlich gleiche Klasse mit dem Namen Y nicht).

Trotz aller Probleme hat sich die Arbeit aber auf jeden Fall gelohnt. Der Einsatz des Generators beschleunigt die Entwicklung extrem und auch die Fehlerquote, zumindest im technischen Code, ist rapide gesunken. Der Einbau der Rule Engine war beispielsweise überhaupt kein Problem und erforderte lediglich kleinere Anpassungen an den Generator-Templates, sodass die Rule Engine automatisch in den Session-Facades eingebunden wird.

Mit XDoclet2 wurden laut den Entwicklern viele der Probleme von XDoclet behoben. Allerdings ist XDoclet2 eine komplette Neuentwicklung, die auch einen anderen Template-Mechanismus benutzt. Dies macht einen Wechsel für uns momentan zu aufwendig. Darüber hinaus ist zu XDoclet2 bisher keine gute Dokumentation verfügbar.

Dennoch bin ich davon überzeugt, dass X-Doclet2 für neue Projekte die bessere Wahl darstellt und einige Probleme von XDoclet umgeht. Man darf jedoch gespannt sein, ob es die gleiche Popularität wie sein Vorgänger erfährt, momentan

XDoclet

Eine allgemeine Einführung zu XDoclet gibt am einfachsten [10]. Kurz zusammengefasst benutzt XDoclet eine an JSPs angelehnte Template-Sprache. Dabei ist XDoclet vor allem daher bekannt, dass es viele Templates für Standardprobleme bereits mitbringt (z.B. Templates für EJB-Deployment-Deskriptoren, DAOs und JDO-Metadaten), eine Übersicht findet sich auf der Homepage von XDoclet [9].

Die Template-Sprache ist einfach zu erlernen, jedoch auch sehr begrenzt in ihren Möglichkeiten. Listing 2 zeigt zur Verdeutlichung einen Ausschnitt aus dem Template für das BusinessDelegate.

Man kann sehen, dass Templates sehr einfach sind, wenn man entsprechende Tags zur Verfügung hat. Tags sind die XML-Elemente mit dem Präfix *XDt*….

XDoclet liefert von Haus aus viele Tags mit, am wichtigsten sind hierbei *XDtClass*, *XDtField* sowie *XDtMethod*. Eine Übersicht über alle Tags sowie deren Leistungsspektrum befindet sich im Javadoc-Stil unter [16]. Leider sind dieses Tags oft nur für bestimmte Einsatzszenarien und ohne Weitblick entwickelt worden, sodass Ihnen wesentliche Methoden fehlen. Beispielsweise kann man mit dem *XDtClass* Tag nicht abfragen, ob eine Klasse ein bestimmtes Interface implementiert. Will man nun, wie wir, umfangreiche Code-Generierung mit XDoclet betreiben, kommt man nicht umhin, viele eigene solcher Tags zu schreiben (z.B. *XDtService* im Beispiel oben). Dies macht den Code-Generator dann auf Dauer unübersichtlich, da die Generierungslogik auf unterschiedliche Teile im System verteilt ist.

Listing 2

```
package <XDtConfig:configParameterValue paramName="modulePackage" />.service;

import <XDtConfig:configParameterValue paramName="modulePackage" />.dto.*;
/**
 * @generated by delegate.xdt
 */
public interface <XDtConfig:configParameterValue paramName=
                                        "moduleNameUpperCase" />Delegate {

<XDtMda:forAllClasses>

<XDtClass:ifClassTagValueEquals tagName="stereotype" value="BusinessEntity">
 <XDtClass:ifHasClassTag tagName="needSerivceCreate" superclasses="false">
  public <XDtClass:className />DTO create<XDtClass:className />(
            <XDtClass:className />DTO dto) throws BusinessException;
 </XDtClass:ifHasClassTag>

 <XDtMethod:forAllMethods><XDtMethod:ifHasMethodTag tagName="provideAsService">
  public <XDtService:generateServiceMethodType /> <XDtMethod:methodName />(
        <XDtUtil:cutOfLastComma><XDtMda:keyClassName />Key targetObjectKey,
        <XDtService:generateServiceMethodParameterDeclaration />
        <XDtUtil:cutOfLastComma> ) throws BusinessException;
 </XDtMethod:ifHasMethodTag></XDtMethod:forAllMethods>

</XDtClass:ifClassTagValueEquals>

<XDtClass:ifHasClassTag tagName="componentService">

 <XDtMethod:forAllMethods><XDtMethod:ifIsPublic>
  public <XDtMethod:methodType /> <XDtMethod:methodName /> (
        <XDtParameter:parameterList />) <XDtMethod:exceptionList />;
 </XDtMethod:ifIsPublic></XDtMethod:forAllMethods>
</XDtClass:ifHasClassTag>
</XDtMda:forAllClasses>
}
```

Anzeige

sieht es nicht danach aus. Neuen Projekten würde ich raten, sich neben XDoclet2 auf jeden Fall einmal EMF [12] genauer anzuschauen. Dieses ist Grundlage vieler Eclipse Plug-ins und hat auch eine ständig wachsende Community. Ebenfalls sehr interessant ist in diesem Zusammenhang das vollwertige MDA-Framework openArchitectureWare [13], welches jedoch von einem XMI-Modell startet (und welches im *Java Magazin* mit der letzten Ausgabe beginnend in drei Ausgaben sehr ausführlich vorgestellt wird; Anm. der Red.).

Grenzen dieser Lösung

Leider gibt es auch ein paar Nachteile der gewählten Lösung. Neben der Tatsache, dass wir auf ein Java-Modell angewiesen sind (was wir aber aus genannten Gründen nicht als Nachteil, sondern als Vorteil sehen), ist dies vor allem der Umstand, dass die Möglichkeiten durch den Generator oft eingeschränkt sind. Will man etwas implementieren, was der Generator noch nicht beherrscht, muss man sich relativ mühsam eine standardisierte Lösung überlegen und in den Generator implementieren. Andererseits hat dies den Vorteil, dass die Architektur stabil bleibt und vor allem Gleiches im Code überall identisch implementiert ist.

Eine große Gefahr dieses Vorgehens ist übrigens, jede Ausnahme in den Generator packen zu wollen. Dann bekommt man riesige, nicht mehr wartbare Templates und hat eigentlich wenig gewonnen. Daher sollte der Architekt sein Hauptaugenmerk immer auf die Frage legen, was über den Code-Generator gelöst wird und wo evtl. Erweiterungspunkte für Ausnahmen vorzusehen sind.

Leider ist es bei unserer Lösung auch nicht möglich, das zugrunde liegende Java-Modell an sich automatisch zu verändern. Dies wäre für *getter-* und *setter-* sowie für *equals-*, *hashCode-* und *toString-*Methoden wünschenswert. In unserem Projekt haben wir uns dafür einen Workaround geschaffen und kopieren diese Methoden aus den generierten DTOs per Ant-Task in einen geschützten Code-Bereich der Modellklassen. Dies ist zwar keine wirklich saubere Lösung (und wird daher hier auch nicht weiter betrachtet), funktioniert aber für unsere Zwecke ohne Probleme.

Code-Generierung vs. Reflection

Vor der Entscheidung, einen Code-Generator zu verwenden, steht immer noch dir Frage, ob das Problem nicht auch mit dem Java-Reflection-Mechanismus dynamisch gelöst werden kann. Für die Problematik der DTOs hatten wir hierfür auch einen Prototypen entwickelt, einen generischen *DtoAssembler*, der zur Laufzeit über Java Reflection Attribute kopiert hat.

Wir haben uns dann aber doch sehr schnell dagegen entschieden, denn eine Lösung mit Reflection ist schlichtweg nicht wartbar. Debugging wird zur Qual, ist aber die einzige Möglichkeit der Fehlersuche. Hier hat die Code-Generierung einen großen Vorteil: Der entstandene Code kann statisch, beispielsweise durch ein Code Review, geprüft und vom Entwickler verstanden werden. Unserer Erfahrung nach ging es noch dazu bedeutend schneller, den Code-Generator zu entwickeln. Unter Laufzeitaspekten hat diese Lösung übrigens auch die Nase vorn.

Zukunftssicherheit der Lösung

Aktuell kann man die sehr emotional geführte Diskussion über die Zukunft der Persistenztechniken verfolgen. Wie geht es mit JDO weiter, was passiert mit EJB 3.0? [14] Das Schöne an unserer Lösung ist, dass wir sehr gelassen auf solche Fragen reagieren können. Solange wir weiterhin ein Java-Modell als Grundlage einsetzen können (was ja auch mit EJB 3.0 wieder möglich ist), ist der Rest kein Problem. Denn müssen wir auf eine neue Technik umstellen, so passen wir lediglich die Templates für den Generator entsprechend an, und zwar nur einmal an zentraler Stelle. Danach läuft unsere gesamte Applikation mit neuem Gewand.

Ausblick

Den hier beschriebenen Code-Generator haben wir bereits zur Projektlaufzeit aus der Warenwirtschaft in ein eigenes Framework ausgelagert. In diesem Framework namens jBop [15] (Java Business Oriented Platform) haben wir des Weiteren auch eine BPM sowie eine Rule Engine integriert. Die Auslagerung ermöglicht eine einfache Wiederverwendung der Konzepte in anderen Projekten.

Denn durch die Standardarchitektur, die Integration einer Open Source BPM sowie Rule Engine sowie den Einsatz des eigenen Code-Generators können Geschäftsanwendungen einfach und schnell entwickelt werden. Dabei mussten wir keine teuren Application Server, Frameworks oder Tools einsetzen und brauchen den Vergleich mit ihnen doch nicht zu scheuen. Mit diesen Voraussetzungen kann man sagen, dass J2EE-Projekte nicht komplex sein müssen und man wieder beruhigt die Vorteile von J2EE genießen kann.

Da unsere Anwendung eine relativ typische Geschäftsanwendung ist, ist die dargestellte Architektur auf viele andere Anwendungen übertragbar. Momentan läuft bereits das nächste Projekt unter jBop und wir überlegen, ob wir das Framework auch der Öffentlichkeit zur Verfügung stellen sollen.

Bernd Rücker (bernd.ruecker@camunda.com) ist Softwarearchitekt und Geschäftsführer bei camunda, einem Unternehmen, das seinen Schwerpunkt auf die Entwicklung und den Betrieb von Geschäftssoftware in J2EE legt und dabei vornehmlich BPM und Business Rule Engines in Verbindung mit serviceorientierten Architekturen einsetzt

Links & Literatur

[1] www.solarmetric.com
[2] www.jboss.org
[3] www.mysql.com
[4] Deepak Alur, John Crupi, Dan Malks: Core J2EE Patterns, Prentice-Hall 2003
[5] hsqldb.sourceforge.net
[6] www.springframework.org
[7] www.andromda.org
[8] Zum Beispiel Thomas Gebert, Rico Wieser: Model Driven Architecture in der Praxis, in *Java Spektrum*, Heft CeBIT 2005
[9] xdoclet.sourceforge.net
[10] Craig Walls, Norman Richards: XDoclet in Action, Manning 2004
[11] xdoclet.codehaus.org
[12] www.eclipse.org/emf/
[13] architecturware.sourceforge.net
[14] Siehe Schwerpunkt JDO vs. Hibernate in *Java Magazin* 6.2005
[15] www.camunda.com/jbop/
[16] xdoclet.sourceforge.net/xdoclet/templates/