syngenio in der Fachpresse

Medium: Javamagazin

Thema: MDA-Praxis: Migration einer Datenbank-Zugriffsschicht

Autor: Thomas Schmidt

Ausgabe : 9.2006

Reifenwechsel

Wie können wir qualitativ hochwertigen Code und einen wesentlichen Teil der Dokumentation in kurzer Zeit erstellen und dabei alle Gesichtspunkte des Designs im Blick behalten? Die Model Driven Architecture (MDA) scheint dafür ein guter Lösungsansatz zu sein. Doch wie sieht das in der Realität aus? Welche Probleme treten in einem realen Projekt auf? Der folgende Erfahrungsbericht zeigt Ihnen, welche Vor- und Nachteile der Einsatz von MDA bei der Migration einer Datenbank-Zugriffsschicht bringt.

Kontakt & weitere Informationen:
syngenio AG
Ivonne Machnacz
Andreas-Hermes-Straße 3
D-53175 Bonn
Fon +49 (0)2 28-6 20 95-121
Fax +49 (0)2 28-6 20 95-150
ivonne.machnacz@syngenio.de
www.syngenio.de

MDA-Praxis: Migration einer Datenbank-Zugriffsschicht

Reifenwechsel

VON THOMAS SCHMIDT

Wie können wir qualitativ hochwertigen Code und einen wesentlichen Teil der Dokumentation in kurzer Zeit erstellen und dabei alle Gesichtspunkte des Designs im Blick behalten? Die Model Driven Architecture (MDA) scheint dafür ein guter Lösungsansatz zu sein. Doch wie sieht das in der Realität aus? Welche Probleme treten in einem realen Projekt auf? Der folgende Erfahrungsbericht zeigt Ihnen, welche Vor- und Nachteile der Einsatz von MDA bei der Migration einer Datenbank-Zugriffsschicht bringt.

Der folgende Erfahrungsbericht stützt sich auf ein Projekt, bei dem eine langsame Java EE-Anwendung beschleunigt werden sollte. Durch ein gestiegenes Lastverhalten und das Hinzufügen immer neuer Features wurde die Anwendung zu Spitzenzeiten unerträglich langsam. Eine Analyse der Anwendung ergab, dass die Performanceprobleme durch den Data Access Layer (DAL) verursacht wurden. Um die schon etwas in die Jahre gekommene Technologie der Datenbank-Zugriffsschicht nicht weiter pflegen zu müssen, wurde entschieden, die Schicht komplett neu zu entwickeln. Diese Neuentwicklung bot gleichzeitig die Möglichkeit, die verwendete Technologie durch eine aktuelle zu ersetzen. Die Migration unterlag der Einschränkung, keine Änderungen an der Datenbank und am Rest der Anwendung vorzunehmen. Sobald die Datenzugriffschicht der Anwendung migriert war, sollte in einem zweiten Teil des Projektes die Anwendung um einen Web Service erweitert werden. Dieser Web Service sollte es einer anderen Anwendung ermöglichen, Statistikdaten aus der Anwendung abzufragen.

Als Persistenz-Framework sollte Hibernate 3 in Verbindung mit dem Spring Framework eingesetzt werden. Da es sich um ein relativ großes System handelte und der Zeitrahmen für die Umstellung sehr kurz war, sollte mithilfe von MDA die Entwicklungszeit beschleunigt werden. Nach einem Vergleich verschiedener MDA-Generatoren bot AndroMDA sehr gute Möglichkeiten, um mithilfe der vorgefertigten Cartridges ein schnelles Ergebnis zu erzielen. Bei der Implementierung des Web Service sollte Apache Axis zum Einsatz kommen. Für die Modellierung wurde auf das bereits im Unternehmen vorhandene UML-Tool Enterprise Architect 5 (EA) gesetzt.

Analyse

Die Basis jeder Anwendung ist die Architektur. Bevor jedoch mit der Festlegung der zu verwendenden Architektur begonnen werden kann, sollten der bestehende Code und die Anforderungen an die Persistenz-Schicht genau analysiert werden. Die wichtigste Frage, die sich zunächst stellte, ist die der Schnittstelle zwischen der Datenbank-Zugriffsschicht und der restlichen Anwendung. Von Vorteil ist es, wenn die Schnittelle bereits klar definiert ist. Sie kann dann – ohne Anpassungen in der Anwendung durchzuführen – 1 : 1 übernommen werden. Eine Optimierung der Schnittstelle sollte zunächst nicht erfolgen. Die beiden einzigen Änderungen an der Businesslogik-Schicht der Anwendung werden sich auf die Aufrufe der Data Access Object-(DAO-)Instanzen und des Transaktions-Handlings beschränken. Nachdem der Aufbau der Schnittstellen geklärt ist, kann nun die Fachlogik im bisherigen DAL analysiert werden. Dazu sollten unter anderem alle enthaltenen Datenbank-Abfragen herausgesucht werden. Diese werden später in das Modell bzw. in den erzeugten Code eingebaut.

Neben dem Code gehört auch die Datenbank selbst zu den zu analysierenden Objekten. Dabei sollte man sich in erster Linie darauf konzentrieren, welche Strukturen in der Datenbank vorhanden sind. Auf folgende Strukturen sollte man dabei besonderes Augenmerk haben:

- Tabellen ohne Primärschlüssel
- Tabellen mit DB-generierten IDs
- Tabellen mit zusammengesetzten Schlüsseln
- fehlende referenzielle Integritäten
- Tabellen mit Diskriminator-Spalte

Phasen

Die gesamte Entwicklung wird in folgende fünf Phasen gegliedert:

1. Architekturanalyse
   - Analyse der bestehenden Architektur
   - Architektur der neuen Datenbank-Zugriffschicht festlegen und Durchstich implementieren
2. PSM erstellen
   - PSM – Erweiterung des Generators um neue Funktionalitäten und Anlegen von Templates
   - Konfiguration
   - Testen
3. PIM erstellen
   - Schem2XMI – Datenbank-Schema in PIM übernehmen
   - UML-Tool
   - PIM anpassen
4. Einbindung der neuen Schicht in die alte Anwendung („Hochzeit“)
5. Erweiterung durch Erweiterungen an PIM und PSM

- 1 : 1-Referenzen
- Spalten mit Konstanten
- Views

Die meisten dieser Strukturen sind für Hibernate kein Problem. Tauchen jedoch Tabellen ohne Primärschlüssel auf, dann müssen diese einen Primärschlüssel bekommen. Am einfachsten lässt sich diese Erweiterung durchführen, indem der Tabelle eine neue ID-Spalte hinzugefügt wird, die einen automatisch von der Datenbank generierten Wert enthält. Dieser dient in Zukunft der eindeutigen Identifizierung eines Datensatzes und wird in der Anwendung nicht auftauchen. Nachdem der Code und die Datenbank analysiert wurden, kann man sich Gedanken über die zu verwendende Architektur machen.

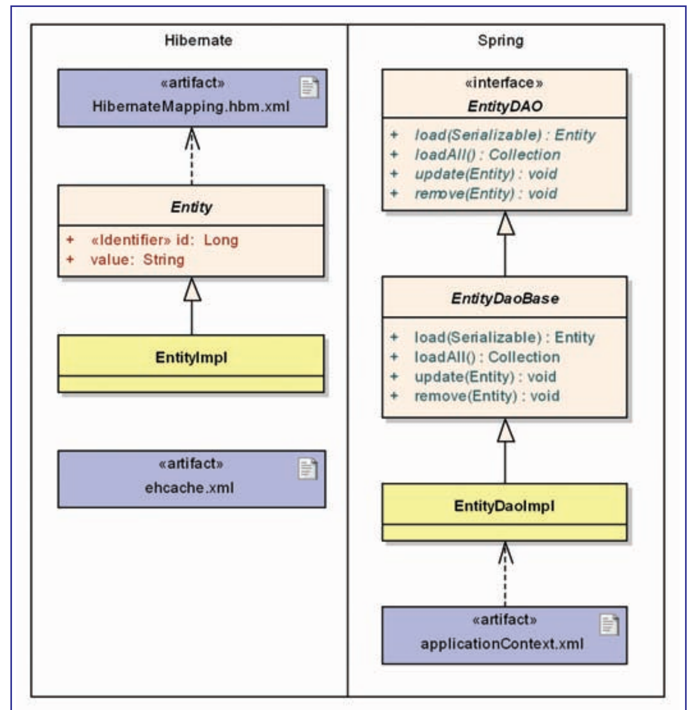


Abb. 1: Klassendiagramm der DAO-Architektur, die AndroMDA standardmäßig generiert

Architektur

Als Basis für die Architekturüberlegungen empfiehlt es sich, sich einmal die Architektur anzuschauen, die AndroMDA ohne Modifikationen generieren würde (Abb. 1). Lehnt man seine eigene Architektur nun an diese an, sind die Template-Anpassungen in der nächsten Phase relativ gering. In der Praxis hat es sich bewährt, die theoretischen Architekturüberlegungen anhand eines technischen Durchstichs zu erproben. Dieser ermöglicht es, die technischen Aspekte anhand eines Mini-Einsatzszenarios praktisch auszuprobieren und damit die Risiken zu minimieren.

Anzeige

Für die Implementierung des Durchstichs kommen keine Generierungswerkzeuge zum Einsatz. In unserem Fall beschränkt sich der Durchstich auf die Implementierung des DAL mit einem Zugriff auf einige wenige Datenbank-Tabellen. Ein JUnit-Test steuert diese Mini-Anwendung und erlaubt das Testen von Operationen. Der Implementierung eines Durchstichs kommt bei der Verwendung von MDA eine besondere Bedeutung zu. Die hier erstellten Klassen werden als Grundlage für die zu generierenden Klassen verwendet. Bei der Entwicklung des Architektur-Prototyps sollte daher besonders gründlich vorgegangen werden, da später der gesamte generierte Code wie der Durchstich-Code aussehen wird.

Neben der reinen Architektur sollten in diesem Zusammenhang auch die Aspekte Caching, Transaktionen und Nebenläufigkeit nicht außer Acht gelassen werden. Von Vorteil erweist sich auch hier wieder, dass AndroMDA diese Dinge bereits generieren kann. Mit ein paar Handgriffen kann der benötigte Transaktionsmanager per Spring-Aspekt an die DAO-Methoden angebunden werden. Für die Transaktionssteuerung aus der Anwendung heraus kann das Transaktions-API von Hibernate auch durch einen direkten Aufruf im Code verwendet werden. Außerdem haben wir uns generell für eine pessimistische Nebenläufigkeitskontrolle entschieden. Wahlweise könnten durch das PIM (Platform Independent Model) einzelne Objekte auch optimistisch ge-„locked“ werden. Für das Caching wird der bereits von AndroMDA eingebundene EHCache verwendet.

Mit der Erstellung der Architektur und des Durchstichs ist die erste Phase der Migration bereits abgeschlossen. Im nächsten Schritt können wir uns nun der Anpassung des Generators an die gewählte Architektur widmen.

PSM (Platform Specific Model)

Je nach verwendetem MDA-Generator kann es zunächst notwendig sein, diesen um fehlende Funktionalitäten zu erweitern. Für uns bedeutete das, die benötigten Funktionalitäten mit den von AndroMDA bereitgestellten Funktionalitäten zu vergleichen. Dabei stellte sich heraus, dass die Steuerung der Diskriminatoren verbessert werden musste. Bei AndroMDA ist die Diskriminator-Generierung sehr statisch implementiert. Für die Anwendung wurde aber eine flexiblere Steuerungsmöglichkeit benötigt. Dazu war es notwendig, zwei Klassen und ein Template im Hibernate Cartridge entsprechend anzupassen und daraus eine eigene Version der Cartridge zu erzeugen. Außerdem

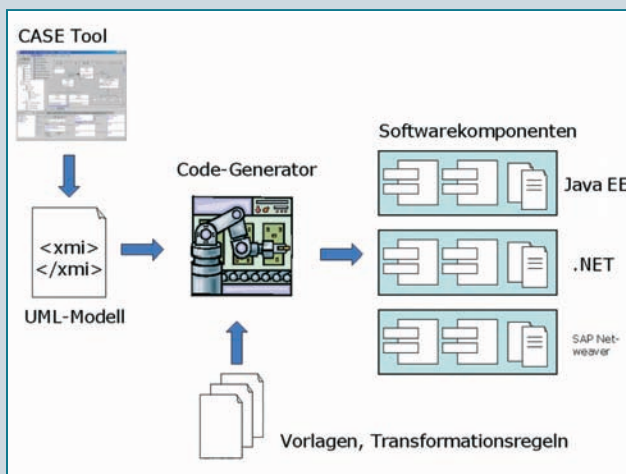
AndroMDA 3

AndroMDA ist ein leistungsfähiger MDA-Generator. Dieser basiert auf so genannten Cartridges, die vorgefertigte Generierungskomponenten darstellen. Daraus ergibt sich der Vorteil, dass AndroMDA sehr schnell einsatzfähig ist. Die enthaltenen Cartridges können durch den Entwickler problemlos ergänzt werden, sodass man relativ schnell Anwendungen in einer eigenen Architektur generieren kann. Mit AndroMDA 3.1 ausgelieferte Cartridges:

- BPM4Struts (Business Process Modeling for Struts)
- jBPM (JBoss Workflow Engine)
- JSF
- EJB
- Hibernate
- Java
- Meta
- Spring
- Web Service (Apache Axis)
- XmlSchema

CRUD-Anwendung

Erstellt man sich mit AndroMDA ein Modell für einen Datenbank-Zugriff, dann bietet es sich an, zusätzlich eine komplette CRUD-Anwendung generieren zu lassen (CRUD = Create, Read, Update, Delete). Diese Anwendung erlaubt es, alle Entities über eine Webanwendung zu bedienen. Um diese Anwendung zu erstellen, sind lediglich die Entities mit einem zusätzlichen Stereotyp <<*Managable*>> zu versehen und die BPM4Struts Cartridge in die Konfiguration miteinzubinden. Man sollte auch darauf achten, dass beim Erstellen der CRUD-Anwendung die Original-Templates verwendet werden.



Funktionsweise von AndroMDA

Listing 1

Merge-Datei, die Hibernate eine weitere Datei generieren lässt

```
<mappings name="CompositeID_Merge_Mapping">
  <mapping>
    <from>
      <![CDATA[<!-- cartridge-template merge-point -->]]>
    </from>
    <to>
      <![CDATA[
  <template
    path="templates/hibernate/HibernatePK.vsl"
    outputPattern="$generatedFile"
    outlet="entities"
    overwrite="true"
    outputOnEmptyElements="false">
    <modelElements variable="entity">
      <modelElement>
        <type name="org.andromda.cartridges.
              hibernate.metafacades.HibernateEntity"/>
      </modelElement>
    </modelElements>
  </template>
      ]]>
    </to>
  </mapping>
</mappings>
```

Diskriminator

Als Diskriminatoren werden Felder bezeichnet, die den Typ eines Datensatzes kennzeichnen. Beispiel: Es gibt eine Tabelle *PAYMENT*, die unter anderem die Art des Zahlungsmittels enthält. Die Spalte mit dem Zahlungsmittel-Typ wird dann als Diskriminator-Spalte bezeichnet. Hibernate bietet die Möglichkeit, die Instanziierung einer Klasse davon abhängig zu machen, was in der Diskriminator-Spalte steht. Steht dort also z.B. *BANK*, wird eine Klasse *Bank* instanziiert, ist die Rolle *CREDIT* wird die Klasse *Creditcard* instanziiert.

fehlte die Funktionalität, mehrere Spalten zu einem zusammengesetzten Schlüssel zusammenfassen zu können. Um dieses Manko zu beheben, wurde diesmal ein anderer Weg gewählt. Ohne die Erweiterung der Cartridge sollte die Funktionalität nur durch das Ändern von Templates erfolgen. Dank der so genannten Merge Points, die es erlauben, AndroMDA-Konfigurationsdateien um eigene Konfigurationsteile zu erweitern, war es vergleichsweise einfach, AndroMDA eine Primary-Key-Datei generieren zu lassen (Listing 1). Durch Anpassung der restlichen Spring- und Hibernate Templates konnte AndroMDA schließlich ohne die Änderung von Cartridges zusammengesetzte Schlüssel verarbeiten.

Sobald sichergestellt war, dass die benötigte Funktionalität vom Generator zur Verfügung gestellt werden konnte, wurde mit dem Anpassen der Templates für die gewählte Architektur begonnnen. AndroMDA verwendet Templates auf Basis der Template Engine Velocity, um Dateien aus den Daten des PIM zu generieren. Die Anpassungen der Templates sind daher in der Regel ohne großen Aufwand möglich.

Konfiguration

Im nächsten Schritt gilt es nun, den Generator so zu konfigurieren, dass die gewünschten Ergebnisse generiert werden können. Neben der eigentlichen Konfiguration über die *andromda.xml* müssen die Datentypen-Mappings angepasst werden.

Da das PIM plattformunabhängig sein sollte, müssen auch die darin verwendeten Datentypen unabhängig von den verwendeten Technologien sein. Daher definiert AndroMDA einen Satz Datentypen, der jederzeit durch eigene Datentypen erweitert werden kann. Je nach verwendeten Technologien werden dann die allgemein gültigen Datentypen in die jeweiligen Plattform-Datentypen gemappt. Für die Generierung einer Datenbank-Zugriffsschicht ist das Mapping in Java-, Hibernate- und Datenbank-Datentyp notwendig. AndroMDA liefert bereits vorgefertigte Mapping-Dateien mit aus. Diese müssen unter Umständen geändert oder ergänzt werden. Bei der zu migrierenden Anwendung gab es z.B. ein Datenbank-Feld, das vom Typ *String* war und Daten im Format *key=value;key=value;...* enthielt. Daraufhin wurde ein User-Datentyp für Hibernate definiert, der automatisch das Feld in eine *HashMap* einliest und eine *HashMap* auch wieder in diesem String-Format speichern kann. Um diesen neuen Datentyp AndroMDA bekannt zu machen, waren Änderungen in allen drei Mapping-Dateien notwendig (Listing 2).

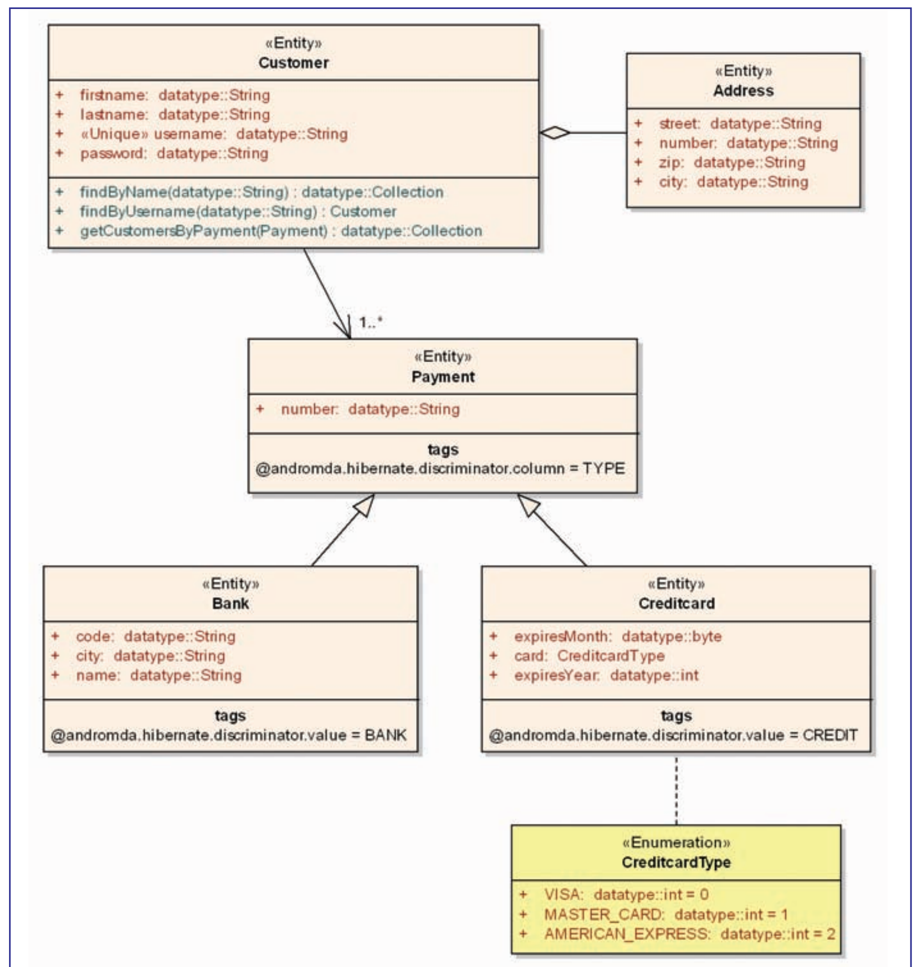


Abb. 2: PIM des Durchstichs

Testen

Sobald Änderungen am Generator vorgenommen werden, sollten diese auch getestet werden. Dazu erstellt man zunächst ein Mini-PIM, das der Funktionalität der Durchstich-Implementierung entspricht (Abb. 2).

Lässt man den Generator nun mit dem PIM laufen, sollte er exakt denselben Code erzeugen wie der manuell erstellte Code. Erst wenn diese beiden Quelltexte übereinstimmen, kann man sicher sein,

Listing 2

Beipiel der angepassten Mapping-Dateien für eine HashMap

```
<mappings name="JavaExtension">
  <extends>Java</extends>
  <mapping>
    <from>datatype::HashMap</from>
    <to>java.util.HashMap</to>
  </mapping>
</mappings>

<mappings name="DB2Extension">
  <extends>DB2</extends>
  <mapping>
    <from>datatype::HashMap</from>
    <to>VARCHAR</to>
  </mapping>
</mappings>

<mappings name="HibernateExtension">
  <extends>Hibernate</extends>
  <mapping>
 <from>datatype::HashMap</from>
 <to>org.andromda.usertypes.HashMapString</to>
  </mapping>
</mappings>
```

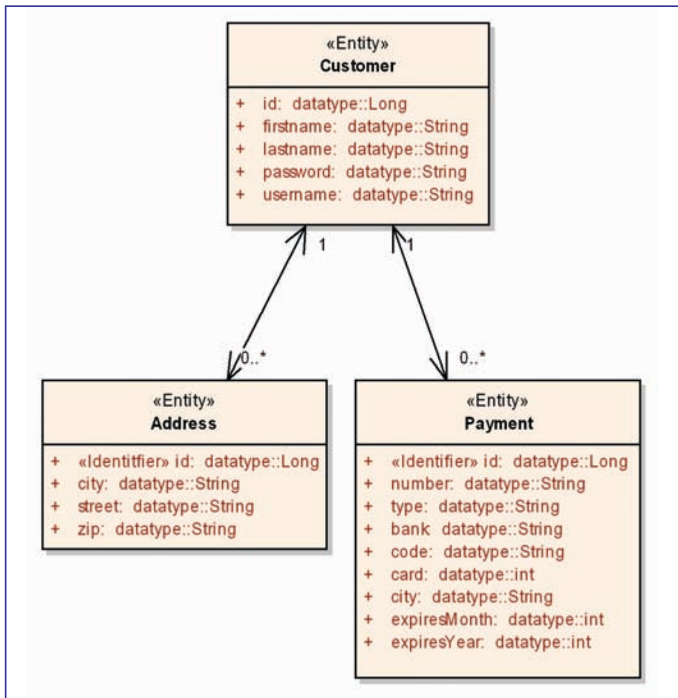


Abb. 3: Auszug aus einem von Schema2XMI generierten Modell

das die Qualität des generierten Codes dem gewünschten Qualitätsstandard entspricht.

Schema2XMI

Sämtliche Vorbereitungen sind nun abgeschlossen. Jetzt kann das Modell der neuen Datenbank-Schicht erstellt werden. Das Übernehmen von über 100 Tabellen mit allen Feldern und Assoziationen in ein UML-Klassendiagramm ist eine sehr mühselige Arbeit, nicht zuletzt wegen der hohen Fehleranfälligkeit beim Eingeben von Spaltennamen. AndroMDA liefert daher ein Tool namens Schema2XMI aus. Damit ist es möglich, die Meta-Daten einer Datenbank über eine JDBC-Verbindung direkt auszulesen. Das Tool speichert die ausgelesenen Metadaten in einem XMI File (Version 1.2) und versieht auf Wunsch die Objekte mit einem Stereotyp und einem Tagged Value für den Namen aus der Datenbank. Außerdem werden auch die referenziellen Integritäten als Assoziationen abgelegt.

So weit, so gut, gäbe es da nicht die Eigenheiten der Datenbanken, die über den JDBC-Treiber nicht erkannt werden. DB2-Datenbank-Schemata können z.B. Aliase auf Tabellen in anderen Schemata enthalten. Aus Datenbank-Sicht verhalten sich diese dann so, als wenn die Daten in dem aktuellen Schema liegen. Leider erkennt der verwendete JDBC-Treiber lediglich den Namen der Alias-Tabellen. Die Felder werden nicht erkannt. Als Lösung bleibt dann nur die Möglichkeit, die Felder von Hand anzulegen oder auch die Schemata aus der Datenbank auszulesen, die die Original-Tabellen enthalten. In Abbildung 3 ist ein Ausschnitt aus einem solchen Schema-Export zu sehen.

UML-Tool

In aller Regel ist das erste generierte XMI nicht aussagekräftig genug, um daraus eine komplette Anwendung zu erstellen. Darum muss das XMI in ein UML-Tool geladen und dort weiter bearbeitet werden. Bei der Wahl des UML-Tools sollte man darauf achten, ein Tool zu wählen, das den XMI 1.2-Standard vollständig versteht. Bei Tools, die den Standard nicht vollständig unterstützen, kann es zu den unterschiedlichsten Problemen beim Laden und Speichern von XMI-Dateien kommen. Anstelle des von uns eingesetzten Enterprise Architect können auch andere Tools eingesetzt werden. Auf der Website von AndroMDA findet sich dazu eine Liste von UML-Tools, die mit AndroMDA getestet wurden.

AndroMDA hat seine plattformunabhängigen Datentypen in einem Package *datatypes* abgelegt. Beim Versuch, die XMI-Datei in EA zu importieren, stellten wir fest, dass die Package-Namen, in denen die Datentypen abgelegt waren, verschwunden waren. Als Lösung blieb nur, alle Datentypen im Modell manuell zu ändern. Dazu musste vor jedem Datentyp *datatype::* geschrieben werden (Abb. 2). Leider kam erst nach der Anpassung jemand auf die Idee, die Mapping-Datei von AndroMDA anzupassen (Listing 2). Das wäre mit geringem Aufwand möglich gewesen. Da wir die Umstellung aber schon gemacht hatten, beließen wir die Mapping-Dateien so. Als Nächstes mussten wir feststellen, dass alle Tagged Values, die von Schema2XMI erzeugt wurden, nicht importiert wurden. Durch Modifikationen an der generierten XMI-Datei konnte dieses Problem allerdings behoben werden.

Ein weiteres Feature, das beim EA fehlte, war die Möglichkeit, die AndroMDA-Profile zu importieren. EA unterstützt zwar Profile, jedoch in einem eigenen XML-Format. Das hatte also zur Folge, dass die benötigten Profil-Daten in dieses XML-Format konvertiert werden mussten.

PIM

Der größte Teil der Arbeit ist nun getan. Das PSM erzeugt aus einem PIM eine Anwendung in der benötigten Architektur und das Modell liegt in einer ersten Version im UML-Tool vor. Zunächst sollte nun das Modell in die richtige Struktur gebracht werden. Neben der Konfiguration der Assoziationen (Multiplizitäten, Namen, Richtung) sollten auch die Hierarchien der Klassen geprüft werden. Ist also z.B. in einer Tabelle *Kunde* eine Adresse enthalten, sollten im Modell daraus zwei Objekte gemacht werden. Oder sind in der Tabelle Zahlungsmittel, Kontoverbindungen und Kreditkarteninformationen gespeichert, kann die Tabelle über den Diskriminator in drei Objekte zerlegt werden (Zahlungsmittel, Konto, Kreditkarte). Sind in den Tabellen Konstanten enthalten, sollten diese als Enumerationen abgelegt werden. Man sollte also alle Möglichkeiten nutzen, um das Modell typsicher zu bekommen (Abb. 2).

Sind alle Strukturanpassungen im Modell vorgenommen worden, kann das

PIM getestet werden. Erst wenn eine Generierung ohne Fehlermeldungen möglich ist, sollte mit der Anpassung des PIM fortgefahren werden. Funktioniert das PIM, kann damit begonnen werden, die benötigten DAO-Schnittstellen-Methoden in den Entities zu definieren. Dazu sind für einfache Anfragen Query-Methoden anzulegen, deren SQL von AndroMDA generiert werden kann. Bei komplexeren Methoden hat man drei Möglichkeiten, eine Query zu definieren:

- über ein Tagged Value,
- per OCL-Constraint (die Query wird in der Object Constraint Language im UML definiert) oder
- indem die Query direkt im Code abgelegt wird.

Für den letzten Fall generiert AndroMDA lediglich eine leere Methode, deren Inhalt manuell angegeben werden kann. Am Ende sollte das Interface der ehemaligen Schnittstelle entsprechen. Ist das PIM fertig erstellt und getestet, kann mit der plattformspezifischen Optimierung des PIM begonnen werden. Einige Kritiker werden jetzt natürlich fragen: Warum muss ein plattformunabhängiges Modell für eine Plattform speziell angepasst werden? Die Antwort ist einfach: Das PIM-Modell ist kein reines PIM. In der Praxis kann der AndroMDA-Generator noch nicht so intelligent arbeiten, wie es für die Verarbeitung eines PIMs nötig wäre. Würde man sich für jede Anwendung ein eigenes PSM erstellen, könnte man das natürlich so aufbauen, dass es speziellen Code ohne Definition plattformspezifischer Tagged Values auskommt. Bei einem Generator wie AndroMDA, der auf eine schnell aufzubauende Anwendung setzt, ist eine plattformspezifische Anpassung also fast immer nötig. Das entspricht zwar nicht den Eigenschaften eines reinen PIM, es ist aber ein guter Kompromiss zwischen Plattformunabhängigkeit und dem Bereitstellen allgemein gültiger Cartridges.

Im Rahmen der Migration bedeutet das, dass einige Tagged Values angepasst werden mussten, mit denen man z.B. bei Hibernate das Lazy Loading oder das Cache-Verhalten beeinflussen kann. Die plattformspezifischen Tags sind bei AndroMDA leicht am Plattformnamen, im Namen eines Tagged Value, zu erkennen (z.B. *@andromda.hibernate.discriminator.column*).

Nach der Fertigstellung kann das Modell am Ende wieder als XMI abgespeichert werden. Beim Start von AndroMDA wird das erstellte Modell automatisch auf Korrektheit überprüft. Sind Fehler vorhanden, müssen diese im Modell ausgebessert werden. Eine Generierung erfolgt erst, wenn alle Modellfehler behoben worden sind.

„Hochzeit“

Jetzt kommt es darauf an, wie gut das PSM erstellt und getestet wurde. Wurden alle technischen Features erkannt und ausprobiert oder sind Anpassungen am PSM oder gar am PIM erforderlich? Nach der Generierung der DAO-Schicht und den notwendigen Anpassungen an der Anwendung sollte die Anwendung komplett funktionieren. In der Praxis sind nach einigen kleineren Iterationen die Probleme ausgeräumt und die Anwendung funktioniert mit der neuen DAO-Schicht. Nachdem entsprechende Lasttests die neue Schicht getestet haben, kann die Migration der Anwendung abgeschlossen werden. Der Aufwand, der bis hier in die Migration geflossen ist, ist etwas geringer als die geschätzte Entwicklung von Hand. MDA zahlt sich also trotz kleiner Probleme, die hauptsächlich auf das Zusammenspiel mit dem UML-Tool zurückzuführen sind, bei der Migration einer Datenzugriffschicht aus.

Erweiterungen

Bei Erweiterungen am PIM kann MDA nun seine ganzen Stärken ausspielen. Der jetzt einzubauende Web Service soll dazu genutzt werden können, Statistikdaten aus dem System abzufragen. Dazu ist die Erweiterung des Datenbank-Schema um eine Tabelle notwendig, in der entsprechende Zugangsdaten für die Web-Service-Anwendung hinterlegt sind.

Um eine Tabelle zu ergänzen, kann das existierende PIM in das UML-Tool geladen werden. Nachdem die Tabelle in der Datenbank und im PIM angelegt wurde, kann die DAO-Schicht neu generiert werden. In der DAO-Schicht stehen daraufhin die erforderlichen Methoden

Anzeige

zur Verfügung, um auf diese Tabelle zugreifen zu können. Zusätzlich können bei Bedarf noch weitere Query-Methoden ins PIM eingebaut werden. Um nun noch den Web Service selbst zu erstellen, wurde im Modell eine entsprechende Serviceklasse angelegt und mit den benötigten Web-Service-Methoden versehen (Abb. 4).

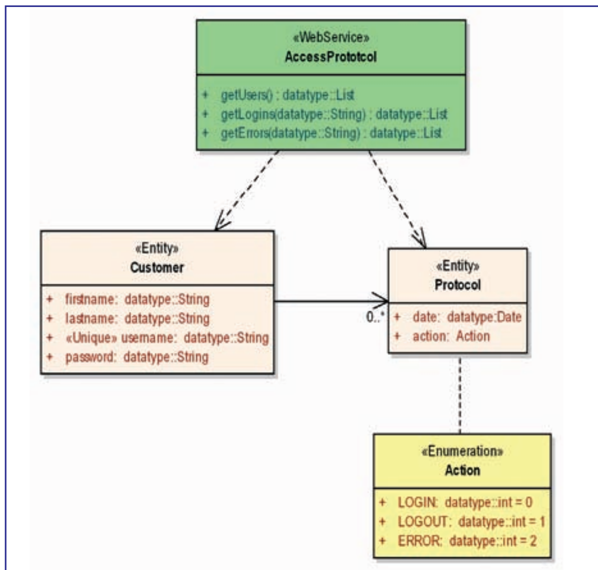


Abb. 4: Modellerweiterung um einen Web Service

Nachdem auch auf PSM-Seite die Web Service Cartridge angebunden und angepasst wurde, kann mithilfe von AndroMDA auch die komplette Web-Service-Funktionalität generiert werden. In der Implementierungsklasse des Web Service sind lediglich die Methoden zu implementieren.

Daraus ist ersichtlich, dass die Erstellung einer weiteren Schnittstelle zur Datenbank mit einem existierenden PIM grundsätzlich problemlos möglich ist. Der gesamte Overhead, der durch die Entwicklung der Web-Service-Schnittstelle entsteht, wird von AndroMDA abgenommen.

Neben der Möglichkeit, leichter Anpassungen am System vornehmen zu können, hat man auch für die Zukunft vorgesorgt. Sollte das Ganze später einmal auf EJB 3 umgestellt werden, ist nur noch das EJB 3 Cartridge (das es noch nicht gibt) in die Konfiguration aufzunehmen. Zusätzlich müssen – wie immer – die Templates an die eigene Architektur angepasst werden. Im PIM sind jetzt nur noch die plattformspezifischen Tags für Hibernate durch die Tags von EJB 3 auszutauschen, und schon kann ohne Änderung am Modell die gesamte Anwendung auf EJB 3 laufen. Der Aufwand für weitere Migrationen ist also noch einmal erheblich gesunken, da das PIM schon existiert.

Fazit

Der Artikel beschrieb in kurzer Form den Ablauf eines MDA-Projektes. Weiterhin wurde gezeigt, dass Erweiterungen am Modell sehr einfach möglich sind. Die hier gemachten Angaben sind lediglich als grober Leitfaden zu verstehen und spiegeln bei weitem nicht den Umfang dieses Themas dar. Der Artikel sollte auch zeigen, welches Potenzial in MDA steckt und welche Probleme praktisch beim Entwickeln auftreten können.

Kommen wir nun zur Beantwortung der eingangs gestellten Fragen. Wie beschrieben, kann durch die Anpassung der AndroMDA Templates Code generiert werden, der den gleichen Anforderungen gerecht wird wie manuell erstellter Code. Durch die Generierung hat man weiterhin den Vorteil, dass der gesamte Code gleich aussieht und damit die Qualität an allen Stellen sichergestellt werden kann. Bei Qualitätsproblemen kann der Code an einer zentralen Stelle (im Template) angepasst werden. Die Template-Anpassungen können auch dazu genutzt werden, seine eigenen Designvorstellungen umzusetzen. Durch die „automatische“ Dokumentation der Schnittstellen in einem UML-Klassendiagramm kann die Anwendung später leicht angepasst werden, ohne dass man dabei den Gesamtüberblick verliert. Oft ist es ja so, dass durch eine neue Anforderung eine Kleinigkeit geändert werden muss. Durch die vollständige Dokumentation des Modells in UML werden die Abhängigkeiten in der Datenschicht sehr schnell sichtbar. Die Dokumentation der Klassen und Methoden im UML-Modell erlaubt es auch, sämtliche Javadoc-Kommentierungen im generierten Code zu beeinflussen. Zusätzlich bietet diese Methode den Vorteil, dass neue technische DB-Schnittstellen sehr schnell erstellt werden können. Es ist aber anzumerken, dass auch Generatoren, die schon vorgefertigt Komponenten mitliefern, nicht unbedingt vollständig sind. Es gibt oft Stellen, die man an seine eigenen Bedürfnisse anpassen muss. AndroMDA hat das erkannt und versucht, durch einfache Mechanismen das Anpassen leicht zu machen. Bevor man mit der Umsetzung beginnt, sollte man sich aber die benötigte Funktionalität und Architektur genau anschauen. Weichen die eigenen Vorstellungen zu sehr von den bereitgestellten Ideen ab, sollte überlegt werden, ob nicht die Wahl eines anderen Tools oder die Erstellung einer eigenen Cartridge sinnvoller ist.

Bleibt noch die Frage des Aufwandes für eine MDA-Anwendung. Aus unserer Erfahrung heraus können wir sagen, dass der Einsatz von MDA nur dort Sinn macht, wo gleichartiger Code entstehen soll. Die Persistenzschicht einer Anwendung ist dafür ein guter Kandidat. Es sollte aber von Fall zu Fall entschieden werden, ob sich der Aufwand lohnt. Je öfter ein erstellter Generator eingesetzt werden kann, umso besser ist natürlich sein ROI. Der Aufwand ist für die Erstellung bzw. Anpassung eines Generators immer am größten. Wir verstehen MDA daher eher als Werkzeug, das dem Entwickler hilft, gleichartigen Code zu erstellen.

Thomas Schmidt ist Consultant bei syngenio. Er befasst sich seit mehreren Jahren mit der Konzeption und Realisierung sowie dem Test von Enterprise-Java-Applikationen. Darüber hinaus beschäftigt er sich aktuell intensiv mit dem Thema MDA.

Links & Literatur

[1] AndroMDA: www.andromda.org

[2] Enterprise Architect: www.sparxsystems.com