

**Ausarbeitung**

**MDA-Frameworks: AndroMDA**

im Rahmen des Seminars „Ausgewählte Kapitel des Software Engineering“

Daniel Schulz

Themensteller: Prof. Dr. Herbert Kuchen  
Betreuer: Christoph Lembeck  
Institut für Wirtschaftsinformatik  
Praktische Informatik in der Wirtschaft

---

## Inhaltsverzeichnis

|     |  |    |
|-----|--|----|
| 1   | Einleitung.....  | 1  |
| 2   | Model Driven Architecture.....                               | 1  |
| 2.1 | Übersicht über die Konzepte der MDA .....                    | 1  |
| 2.2 | Umsetzungsalternativen.....                                  | 2  |
| 3   | Codegenerierung mit AndroMDA.....                            | 3  |
| 3.1 | Übersicht über den Codegenerierungsprozess mit AndroMDA..... | 3  |
| 3.2 | Modellierung des plattformunabhängigen Modells.....          | 5  |
| 3.3 | Aufbau und Funktionsweise der AndroMDA-Cartridges.....       | 7  |
| 4   | Entwicklung von J2EE-Anwendungen mit AndroMDA.....           | 12 |
| 4.1 | AndroMDA J2EE-Cartridges .....                               | 12 |
| 4.2 | Entwicklung einer J2EE-Beispielanwendung.....                | 14 |
| 5   | Zusammenfassung, Bewertung und Ausblick .....                | 20 |
|     | Literaturverzeichnis .....                                   | 22 |

## 1 Einleitung

Mit der Model Driven Architecture (MDA) hat die Object Management Group (OMG) einen Rahmenstandard zur modellgetriebenen Softwareentwicklung geschaffen. Hierbei verlagert sich der Fokus des Entwicklers weg von der Implementierung hin zu Modellen. Trotz des theoretischen Nutzens dieses Vorgehens stellt sich die Frage, wie eine Umsetzung der MDA heute aussehen kann. Die vorliegende Ausarbeitung widmet sich dieser Fragestellung und stellt mit AndroMDA ein weit verbreitetes Open-Source-MDA-Tool vor.

Im zweiten Abschnitt wird zunächst ein Überblick über die Konzepte der MDA und deren Umsetzung in Tools gegeben. Dies ermöglicht eine Einordnung der darauf folgenden Darstellungen. Im dritten Abschnitt erfolgt die Darstellung des Codegenerierungsprozesses mit AndroMDA und dessen Anpassung an unterschiedliche Zielplattformen. Das vierte Kapitel ist dem Einsatz des Tools im Rahmen von Java 2 Enterprise Edition (J2EE) Projekten gewidmet. Neben der Entwicklung des Codegenerators liegt hier ein weiterer Fokus der AndroMDA-Entwickler. Die Erläuterungen erfolgen in diesem Abschnitt an Hand einer Beispielanwendung. Die Ausarbeitung schließt mit einer Zusammenfassung, in der auch ein Ausblick auf die weitere Entwicklung von AndroMDA gegeben wird.

## 2 Model Driven Architecture

### 2.1 Übersicht über die Konzepte der MDA

Grundlage der von der OMG in [OMG03] dargestellten Model Driven Architecture ist die schrittweise Transformation von abstrakten Modellen eines Softwaresystems hin zur Implementierung. Zentral ist hierbei die Unterscheidung zwischen plattformunabhängigen (engl. Platform Independent Model, PIM) und plattformspezifischen Modellen (engl. Platform Specific Model, PSM). Ein PIM beschreibt die fachlichen Aspekte eines Softwaresystems ohne Bezug zu bestimmten Technologien. Ein PSM ist angereichert mit zusätzlichen Informationen, die die Umsetzung in eine konkrete Technologie ermöglichen. Im Regelfall wird als Modellierungssprache die Unified Modeling Language (UML) verwendet. Abbildung 1 enthält eine Darstellung des Übergangs von einem PIM zu mehreren PSM und von dort aus zum Quellcode. Ein zentraler Aspekt der

Transformationen ist, dass diese möglichst automatisiert durch Transformationstools durchgeführt werden sollen.

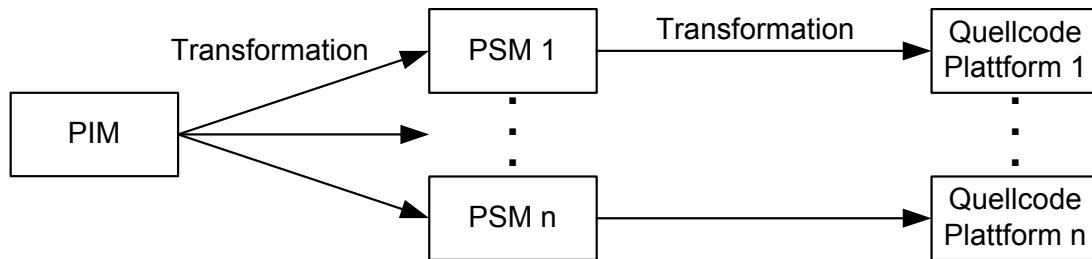


Abbildung 1: Übergang zwischen den Modelltypen und dem Quellcode

Die Nutzenpotentiale der MDA lassen sich nach [KWB03, Kapitel 1.3] in vier Bereiche kategorisieren: Produktivitätssteigerung, Erhöhung der Portabilität von Anwendungen, Verbesserung der Interoperabilität von Komponenten und Vorteile bei Dokumentation und Wartung. [JaHe03, S. 15] nennt zusätzlich als Vorteil der Codegenerierung die gleich bleibend hohe Qualität von generiertem Quellcode.

## 2.2 Umsetzungsalternativen

Für die Umsetzung der MDA-Konzepte in Tools bieten sich verschiedene Alternativen an, die im Folgenden kategorisiert werden sollen. Erstes Kriterium ist laut [KWB03, Kapitel 3.3], welche Stufen des Transformationsprozesses von einem Tool abgedeckt werden. Der populärste Ansatz ist laut [SB03] der, aus einem PIM unter Verzicht auf das PSM direkt Quellcode zu erzeugen. Diese Variante ist auch in der MDA-Spezifikation in [OMG03, Abschnitt 3.7] vorgesehen. Vorteil dieses Vorgehens ist der Verzicht auf Modell-zu-Modell-Transformationen, deren Umsetzung insbesondere durch das Fehlen eines Standards zur Beschreibung von Transformationsregeln erschwert wird. Stattdessen können die Transformationen bei diesem Ansatz mit Hilfe von Templatesprachen beschrieben werden, die die Erstellung von Schablonen für den zu generierenden Quellcode ermöglichen. Das in dieser Ausarbeitung vorgestellte AndromDA setzt diesen Ansatz um.

Zweites Kriterium zur Kategorisierung von MDA Tools ist, welche Funktionalitäten vom Tool selbst unterstützt werden und welche über externe Tools realisiert werden müssen. Einen Überblick über die notwendigen Teilfunktionalitäten bietet [KWB03, Kapitel 3.3]. Ein dritter Aspekt der Umsetzung der MDA sind die Eigenschaften der Transformationen. Nach [KWB03, Kapitel 7.1] sollen Transformationen bidirektional, anpassbar, rückverfolgbar und konsistent bei Neugenerierung sein. Insbesondere die Bi-

direktionalität ist bei template-basierten Generatoren schwer zu realisieren, da sich die Rückgewinnung eines Modells aus Quellcode laut [SB03] als problematisch erwiesen hat.

### **3 Codegenerierung mit AndroMDA**

#### **3.1 Übersicht über den Codegenerierungsprozess mit AndroMDA**

Bei AndroMDA handelt es sich um ein in Java entwickeltes MDA-Tool, welches als Open-Source-Projekt unter der BSD-Lizenz veröffentlicht und über Sourceforge.net verwaltet wird. Die Dokumentation erfolgt ausschließlich über die offizielle Homepage unter [And05], für den Support steht ein Online-Forum zur Verfügung. Zusätzlich werden kostenpflichtige Beratungs- und Schulungsangebote der Entwickler bereitgestellt.

Das Projekt beschäftigt sich bisher nur mit der Entwicklung eines Generatorkerns, eine integrierte Entwicklungsumgebung oder ein integriertes Modellierungstool stehen nicht zur Verfügung.

In Abbildung 2 ist der Codegenerierungsprozess mit AndroMDA schematisch dargestellt. Die Darstellung lehnt sich an [BS03] an, wurde aber um interne Abläufe ergänzt und an die aktuelle Version von AndroMDA angepasst. Kernkomponente von AndroMDA ist ein template-basierter Generator, der aus UML-Modellen Quellcode für beliebige Plattformen und Programmiersprachen generiert. Hierzu liest das Tool UML-Modelle ein, die in externen Modellierungstools entworfen wurden. In diesen Modellen sind die fachlichen Aspekte eines Softwaresystems spezifiziert. Eine detaillierte Erläuterung der Modellierung folgt in Abschnitt 3.2.

Die Anpassung der Codegenerierung an eine spezifische Plattform erfolgt mittels Cartridges. Hierbei handelt es sich um JAR-Archive, die Templates als Schablonen für den zu generierenden Quellcode sowie Java-Klassen (Metafacades) enthalten. Durch Auswechslung der Cartridges kann aus demselben Modell Quellcode für unterschiedliche Plattformen erstellt werden. Die Auswahl der Cartridges wie auch andere Konfigurationsparameter werden dem Tool über die Datei `andromda.xml` mitgeteilt. Aufbau und Funktionsweise der Cartridges sind in Abschnitt 3.3 beschrieben. Abschnitt 4 beschäftigt sich mit den im Rahmen des AndroMDA-Projekts entwickelten J2EE-Cartridges.

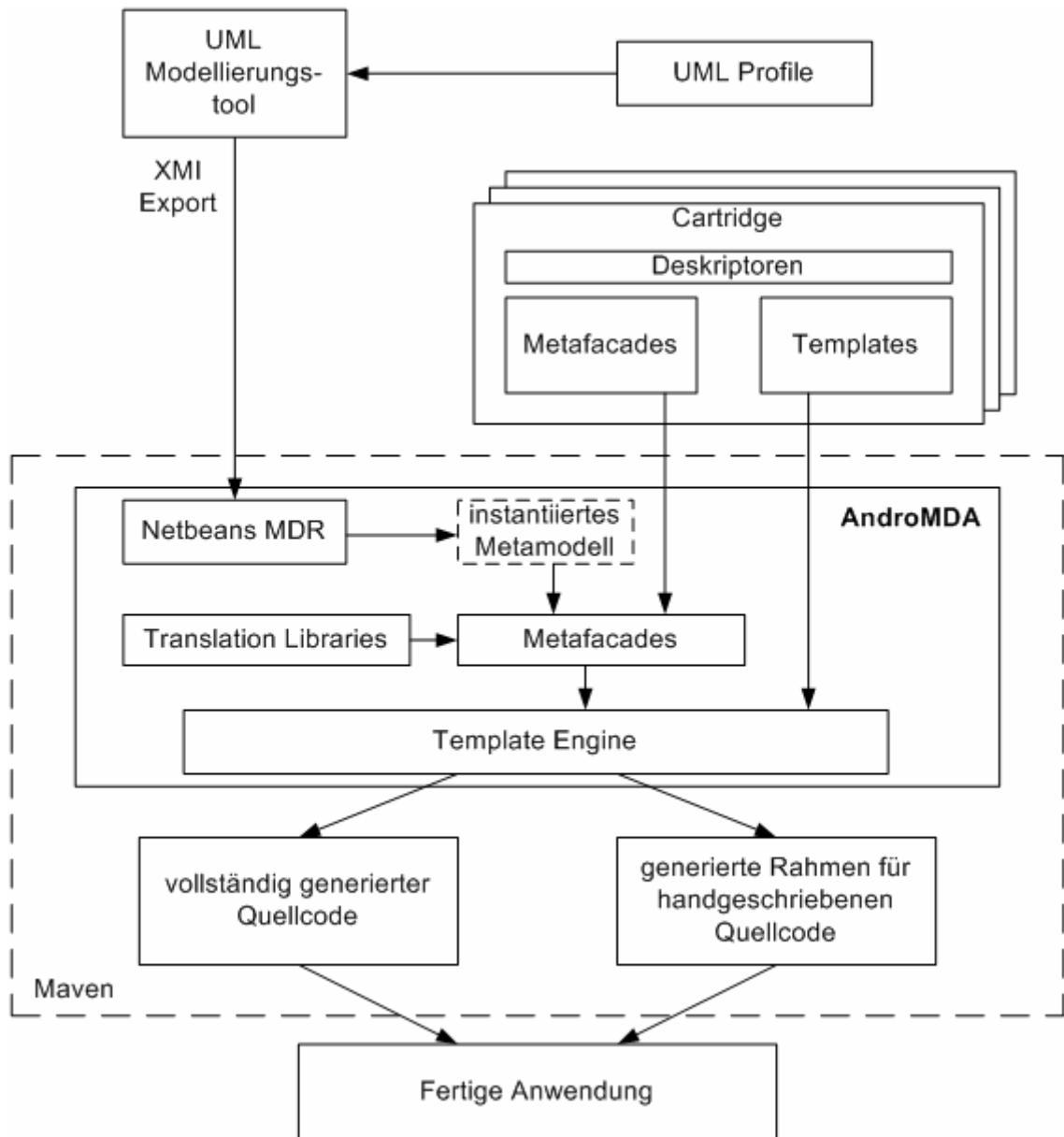


Abbildung 2: Codegenerierung mit AndroMDA

Neben den Cartridges gibt es mit den Translation Libraries noch eine weitere Art von Modulen, die in die Codegenerierung eingehen. Sie werden von Cartridges zur Verarbeitung von Ausdrücken auf Modellebene genutzt, die in der Object Constraint Language (OCL) geschrieben sind. Die OCL ist Teil der UML-Spezifikation und in [OMG05a] dokumentiert. In AndroMDA dienen diese Ausdrücke zur Modellvalidierung (siehe Abschnitt 3.3) und zur Formulierung plattformunabhängiger Datenbankabfragen (siehe Abschnitt 3.2).

Der von AndroMDA generierte Quellcode wird in zwei Gruppen unterteilt, komplett generierten Quellcode und Rahmen zum Ausfüllen mit handgeschriebenem Quellcode. Letztere werden bei einer erneuten Generierung nicht überschrieben. Diese Aufteilung

ermöglicht es, auch nur Teile von Anwendungen zu generieren. Ob eine Komplettgenerierung möglich ist, hängt von den verwendeten Cartridges ab. Unterstützt die Zielsprache Vererbungsmechanismen, bietet sich die Verlagerung von generierten Fragmenten in eine abstrakte Oberklasse an, die bei Neugenerierung überschrieben wird. In einer Unterklasse werden die verbleibenden Fragmente von Hand implementiert. Diese Unterklasse wird nicht überschrieben.

Die Codegenerierung mit AndroMDA wird im Regelfall über Maven gesteuert, kann aber auch unabhängig hiervon erfolgen. Maven ist ein Build-Tool, das als Open-Source-Projekt unter der Schirmherrschaft der Apache Software Foundation entwickelt wird. Die Projekthomepage ist unter [MAV05] zu erreichen. Die Anbindung von AndroMDA an Maven erfolgt mittels eines hierfür entwickelten Plug-Ins. Maven verwaltet Informationen über Software-Projekte in Form von XML-Dokumenten und bietet dem Nutzer die Möglichkeit, den Build-Prozess über Kommandos, sog. Goals, zu steuern. Zusammen mit dem AndroMDA-Plug-In bietet Maven u. a. Goals für das Kompilieren inklusive der Durchführung von Unit Tests, für den Generierungsprozess mit AndroMDA, für das Deployment von Anwendungen in Applikationsserver und für das Erzeugen von Datenbankschemata. Im Zusammenhang mit den in Abschnitt 4 beschriebenen J2EE-Cartridges bietet das Maven Plug-In einen Projektgenerator, der per Frage-Antwort-Dialog die zu verwendenden Cartridges auswählt und ein vorkonfiguriertes Projektverzeichnis erstellt.

### **3.2 Modellierung des plattformunabhängigen Modells**

Für die Modellierung können sowohl die Verhaltens- wie auch die Strukturmodelle der UML verwendet werden. Das dafür verwendete Tool muss den Export der Modelle in das XMI-Format (XML Metadata Interchange) unterstützen. Die Spezifikation dieses Standards findet sich in [OMG05b].

Die Repository-Komponente, die für das Einlesen des XMI-Dokuments und die interne Repräsentation des Modells verwendet wird, ist als austauschbare AndroMDA-Komponente vorgesehen. Momentan wird aber ausschließlich das Netbeans Metadata Repository (MDR) unterstützt. Die Projekthomepage des Netbeans MDR findet sich unter [MDR05]. Durch die Verwendung des MDR sind die Modelle auf die UML Version 1.4 beschränkt, UML 2.0 wird momentan nicht unterstützt.

Die Modellierung für AndroMDA basiert auf der Verwendung von UML Stereotypen und Eigenschaftswerten (*tagged values*).

Stereotype dienen der Klassifizierung von Modellelementen. Trägt bspw. eine Klasse den Stereotyp «Entity» zur Kennzeichnung einer persistenten Klasse, so wird der Quellcode zu dieser Klasse durch AndroMDA mit Hilfe einer Persistenz-Cartridge erzeugt. Die Templates dieser Cartridge sind diesem Stereotypen zugeordnet. Es ist dabei möglich, einem Modellelement mehrere Stereotype zuzuweisen. Außerdem können mehrere Cartridges und mehrere Templates einer Cartridge für dasselbe Modellelement Quellcode erzeugen.

Eigenschaftswerte werden zur weiteren Feinsteuerung des Generierungsvorgangs verwendet. Mittels eines Eigenschaftswerts kann z. B. einer Operation, die zu einer mit «Entity» gekennzeichneten Klasse gehört, eine Datenbankabfrage zugeordnet werden. Ein anderer Eigenschaftswert kann dieser Klasse eine bestimmte Caching-Strategie zuweisen.

Eine Menge zulässiger Stereotype und Eigenschaftswerte wird als UML-Profil bezeichnet. Damit der Modellierer auf die entsprechenden Elemente innerhalb des Modellierungstools zugreifen kann, stehen Profile der mitgelieferten Cartridges als XMI-Dateien zum Import zur Verfügung.

Die für AndroMDA generierten Modelle sollen möglichst plattformunabhängig sein, um im Sinne der MDA Plattformwechsel zu ermöglichen, indem lediglich die Cartridges ausgetauscht werden. Demgegenüber stehen jedoch die häufig plattformspezifischen Eigenschaftswerte, wie sie auch in den im Rahmen des AndroMDA-Projekts entwickelten J2EE-Cartridges vorkommen (siehe Abschnitt 4). Die Plattformunabhängigkeit wird aber an anderer Stelle unterstrichen: So soll auf die Verwendung von programmiersprachenabhängigen Datentypen, z. B. `java.lang.String`, verzichtet werden. Stattdessen kommen generische UML-Datentypen, z. B. `String`, zum Einsatz. Die Zuordnung der Datentypen erfolgt in Mapping-Dateien.

Um auf Modellebene auch unabhängig von der verwendeten Datenbanktechnologie zu sein, wird die Formulierung von Datenbankabfragen in der OCL unterstützt. Diese werden mittels Translation Libraries in plattformspezifische Abfragen übersetzt. Als Beispiel diene die «Entity»-Klasse `Fahrzeug` mit den Attributen `Leistung`, `Baujahr` und `Hersteller`. Eine Operation `getKandidaten` soll alle Fahrzeuge eines Herstel-

lers zurückliefern, die eine bestimmte Mindestleistung bieten und nicht älter als ein bestimmtes Baujahr sind. Hierzu dient die Formulierung des folgenden OCL Ausdrucks:

```
context
Fahrzeug::getKandidaten(herstellername: String, mindestbaujahr:
Integer, mindestleistung: Integer): Collection(Fahrzeug)
body
getKandidatenBody: allInstances() ->
select (Fahrzeug | Fahrzeug.hersteller = herstellername and
Fahrzeug.baujahr >= mindestbaujahr and
Fahrzeug.leistung >= mindestleistung)
```

Der Teil des Ausdrucks nach `context` gibt an, auf welche Methode Bezug genommen wird. `body` kennzeichnet den Ausdruck als Abfrage, nach `body` folgt die Definition der eigentlichen Abfrage.

Derselbe Ausdruck kann nun in Abfragen für unterschiedliche Plattformen übersetzt werden. So kann obiger Ausdruck sowohl in die Hibernate-QL-Abfrage

```
from seminar.Fahrzeug as Fahrzeug
where
Fahrzeug.hersteller = :herstellername and
Fahrzeug.baujahr >= :mindestbaujahr and
Fahrzeug.leistung >= :mindestleistung
```

wie auch in die EJB-QL Abfrage

```
SELECT DISTINCT OBJECT(Fahrzeug) FROM Fahrzeug Fahrzeug
WHERE
Fahrzeug.hersteller = ?1 AND Fahrzeug.baujahr >= ?2 AND
Fahrzeug.leistung >= ?3
```

übersetzt werden. Die Beschreibung der Cartridges, bei denen diese Translation Libraries Verwendung finden, folgt in Abschnitt 4.

Im Rahmen der Entwicklung von AndroMDA ist auch das Tool Schema2XMI entwickelt worden. Es dient dazu, bereits vorhandene Datenbankschemata in UML-Modelle zu transformieren. Dazu verbindet sich das Tool mit einem Datenbankserver und liefert als Ausgabe ein XMI-Dokument. Die Verwendung dieses Tools vereinfacht es, vorhandene Softwaresysteme mit AndroMDA weiterzuentwickeln.

### 3.3 Aufbau und Funktionsweise der AndroMDA-Cartridges

Die AndroMDA-Cartridges sind die zentrale Komponente zur Steuerung der Codegenerierung. Sie bestehen wiederum aus zwei Arten von Komponenten, den Metafacades und den Templates, sowie mehreren Deskriptoren in Form von XML-Dokumenten, die

der Registrierung der Komponenten im AndroMDA-Kern und der Konfiguration dienen.

Um das Zusammenspiel der Komponenten zu erläutern, ist eine Betrachtung der internen Abläufe bei der Codegenerierung nötig. Diese sind in Abbildung 2 dargestellt. Aus dem eingelesenen Modell wird im Speicher durch das MDR eine Repräsentation in Form von Java-Objekten generiert, das instantiierte UML-Metamodell. Das UML-Metamodell umfasst die für die Beschreibung von UML-Modellen benötigten Java-Klassen. So wird bspw. für eine Modellklasse `Fahrzeug` u. a. ein Java-Objekt der Klasse `Class` erzeugt, das `Name`-Attribut dieses Objekts hat den Wert "Fahrzeug". Um Quellcode für die Zielplattform generieren zu können, muss innerhalb der Templates auf diese Modellinformation zugegriffen werden können. Außerdem sollten komplexe Vorgänge der Codegenerierung, wie z. B. die Validierung von Modelleigenschaften, nicht innerhalb der Templates erfolgen. Darunter würde die Übersichtlichkeit der Templates leiden. Die Lösung dieser Probleme erfolgt in AndroMDA durch Metafacades.

Der Begriff „Metafacade“ ist eine Kurzform für „Metamodel Facade“. Die so bezeichneten Java-Klassen bieten Schnittstellen für den Zugriff auf Modellelemente. Sie werden zusätzlich zu den vom MDR erzeugten Objekten instantiiert. Dieses Vorgehen stellt die Unabhängigkeit von einer konkreten Version des Metamodells sicher, da aus Templates heraus nur auf Metafacades zugegriffen werden kann. Im AndroMDA-Kern sind bereits Metafacades für die UML-Modellelemente implementiert. Diese stellen die grundlegenden Methoden für den Zugriff auf UML-Modelle zur Verfügung. Zusätzlich kann eine Erweiterung der Metafacades durch Cartridges erfolgen. Die durch eine Cartridge zur Verfügung gestellten Metafacades implementieren Funktionen, die spezifisch für die in der Cartridge enthaltenen Templates sind.

Folgendes Beispiel soll den Sachverhalt weiter illustrieren: In Abbildung 3 sind Metafacades, die für «Entity»-Klassen Verwendung finden, vereinfacht dargestellt.

Die Klasse `Classifier` ist Teil des UML-Metamodells und dementsprechend mit dem Stereotypen «metaclass» gekennzeichnet. Diese Klasse ist u. a. Oberklasse zur UML-Metaklasse `Class`, die für jede Klasse des Modells beim Einlesen instantiiert wird.

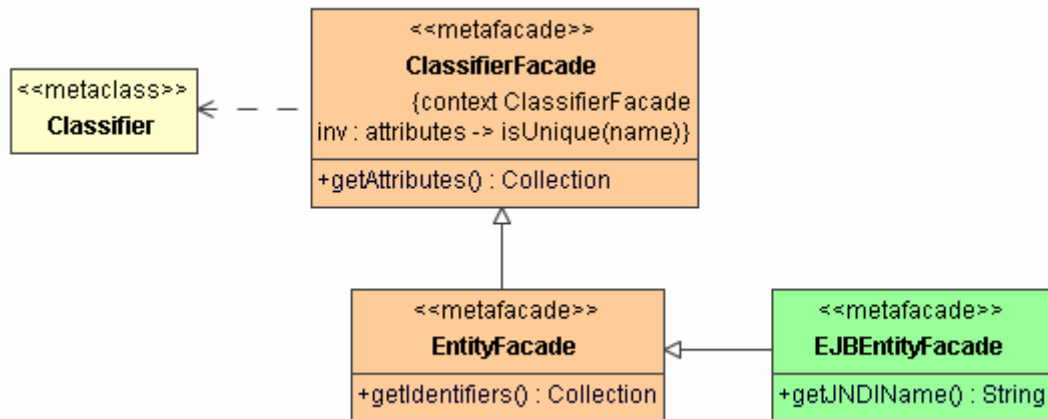


Abbildung 3: Beispiel für Metafacades

Die Klasse `ClassifierFacade` ist die Metafacade zur Metaklasse `Classifier` und Teil der bereits im AndroMDA-Kern implementierten Metafacades. Sie ermöglicht Templates den Zugriff auf alle Klassen, die in einem von AndroMDA verarbeiteten Modell vorkommen. Dafür wird neben einem `Class`-Objekt für jede Modellklasse auch ein `ClassifierFacade`-Objekt erzeugt. Die Methoden dieses Objekts können aus den Templates heraus aufgerufen werden. Ein Beispiel ist die in der Abbildung dargestellte Methode `getAttributes()`, die eine Liste aller Attribute einer Klasse zurückgibt.

Die Klasse `EntityFacade` ist eine Unterklasse von `ClassifierFacade`. Für jede `«Entity»`-Modellklasse wird ein `EntityFacade`-Objekt erzeugt. Dadurch stehen in Templates zusätzliche Methoden für `«Entity»`-Klassen zur Verfügung. Ein Beispiel ist die Methode `getIdentifiers()`, die eine Liste aller Schlüsselattribute zurückgibt. Die Klasse `EntityFacade` ist ebenfalls im AndroMDA-Kern implementiert, da `«Entity»`-Klassen von mehreren Cartridges verarbeitet werden.

Die Klasse `EJBEEntityFacade` ist wiederum Unterklasse von `EntityFacade`. Sie ist nicht Teil des AndroMDA-Kerns, sondern Teil einer Cartridge für die Generierung von Enterprise JavaBeans (EJB). Sie erweitert die Klasse `EntityFacade` um Methoden, die spezifisch für die verwendete Plattform sind. Ein Beispiel ist die Methode `getJNDIName()`. Diese ermöglicht es, innerhalb der Templates den JNDI-Namen der EJB zu ermitteln. JNDI steht hierbei für „Java Naming and Directory Interface“.

Der in Abbildung 3 dargestellte Modellausschnitt enthält außerdem ein Beispiel für die Modellvalidierung in AndroMDA. Im Kontext der Klasse `ClassifierFacade` ist ein OCL-Ausdruck definiert. Dieser ist durch `inv` als Invariante gekennzeichnet, die Bedingung muss damit jederzeit für alle Instanzen dieser Klasse erfüllt sein. Der Ausdruck

überprüft, ob alle Attribute einer Modellklasse unterschiedliche Namen tragen. Sollten zwei oder mehr Attribute denselben Namen haben, so wird der Ausdruck zu `false` ausgewertet. AndroMDA bricht in diesem Fall die Modellvalidierung mit einer entsprechenden Fehlermeldung ab.

Die Entwicklung von Metafacades wird durch die AndroMDA-Meta-Cartridge unterstützt. Dabei wird ein Modell, welches das UML-Metamodell und die bereits im AndroMDA-Kern implementierten Metafacades umfasst, als Ausgangspunkt für Erweiterungen genutzt. Eine detaillierte Darstellung zur Verwendung dieser Cartridge findet sich in der AndroMDA-Metafacades-Dokumentation unter [META05].

Die Templates als Schablonen für den zu generierenden Quellcode sind die zweite Art von Komponenten innerhalb der AndroMDA Cartridges. Die Template-Engine und damit die verwendete Templatesprache sind als austauschbare Komponente des Generators vorgesehen. Momentan steht jedoch ausschließlich die Velocity Template-Engine zur Verfügung. Die offizielle Homepage und Dokumentation dieses Apache Jakarta Projekts findet sich unter [VEL05].

Der Velocity-Sprachumfang ist im Wesentlichen auf Variablenoperationen, Schleifen, Verzweigungen und Zeichenkettenoperationen beschränkt. Attraktiv wird die Verwendung dieser Sprache für AndroMDA-Templates durch die Möglichkeit, Java-Methoden aus den Templates heraus aufzurufen. Dadurch ist ein Zugriff auf die Methoden der Metafacades möglich.

Der folgende Quellcode ist ein Beispiel für ein AndroMDA Template. Dieses Template generiert für jede Klasse, die mit einem entsprechenden Stereotyp gekennzeichnet ist, eine JavaBean. Für alle Attribute werden dabei Zugriffsmethoden erstellt, im Klassendiagramm modellierte Methoden werden nicht ausgewertet. Die im Folgenden in blauer, fetter Schrift dargestellten Quellcodefragmente sind Velocity-Anweisungen oder Metafacade-Methodenaufrufe. Java-Quellcode, der unverändert in die Zielformat übernommen wird, ist in schwarzer Schrift gedruckt.

```
// JavaBean, von AndroMDA generiert
## Paket deklarieren, falls im Modell vorgesehen
#if (!$class.packageName.equals(""))
package $class.packageName;
#end

## Klassendeklaration
public class $class.name
```

```

## falls Oberklasse existiert
#if($class.generalization)
    extends ${class.generalization.fullyQualifiedName}
#end

## Serialisierbarkeit
implements java.io.Serializable
{
## leerer Konstruktor
public ${class.name}()
{
}

## für alle Attribute: Deklaration, get-Methode, set-Methode
#foreach ($attribute in $class.attributes)
    ## Deklaration
    private $attribute.getterSetterTypeName $attribute.name;

    ## get-Methode
    $attribute.visibility $attribute.getterSetterTypeName
    ${attribute.getName}()
    {
        return this.${attribute.name};
    }
    ## set-Methode
    $attribute.visibility void
    ${attribute.setterName} ($attribute.getterSetterTypeName
    $attribute.name)
    {
        this.${attribute.name} = $attribute.name;
    }
#end
}
    
```

Das Template verwendet ausschließlich Methoden der Standard-Metafacades ClassifierFacade und AttributeFacade, wobei diese über class bzw. attribute angesprochen werden. Dabei wird von einer Kurzschreibweise Gebrauch gemacht, die in Velocity für Methodenaufrufe vorgesehen ist: Der Aufruf der Methode \$class.getName() wird bspw. zu \$class.name verkürzt.

Damit AndroMDA eine Cartridge verwenden kann, müssen die Templates und eventuell vorhandene Metafacades in XML-Deskriptoren beschrieben werden. Die Beschreibung der verwendeten Templates erfolgt in der cartridge.xml. Ist das obige Beispiel-Template unter dem Dateinamen JavaBean.vsl gespeichert, so sieht eine minimale cartridge.xml wie folgt aus:

```

<cartridge>
  <template
    path = "templates/JavaBean.vsl"
    outputPattern = "{0}/{1}.java"
    outlet = "java-beans"
  >
    
```

```
        overwrite = "true">
        <modelElements variable = "class">
            <modelElement stereotype = "JavaBean"/>
        </modelElements>
    </template>
</cartridge>
```

path verweist hierbei auf das Template, outputPath gibt ein Muster für die Namen der Ausgabedateien vor. Durch outlet wird ein Bezeichner definiert, mit dem projektspezifisch in der `andromda.xml` festgelegt werden kann, in welchem Verzeichnis die Ausgabedateien generiert werden. Das Attribut `overwrite` legt fest, ob der Quellcode bei erneuter Generierung überschrieben wird. Da im Beispiel keine manuelle Implementierung nötig ist, kann der Quellcode überschrieben werden. In `modelElements` werden die Modellelementtypen aufgelistet, auf die das Template angewendet werden soll, hier alle Klassen mit dem Stereotypen «JavaBean». Alternativ zur Angabe eines Stereotypen kann einem Template auch eine Metafacade-Klasse zugewiesen werden. Alle Modellelemente, für die diese Metafacade instantiiert wird, werden dann durch das Template verarbeitet.

Damit die Cartridge vom AndroMDA-Kern gefunden werden kann, muss ein eigener Namensraum mit Verweis auf die `cartridge.xml` definiert werden. Dies erfolgt in der Datei `namespace.xml`. Hier werden auch Konfigurationsparameter (`properties`) festgelegt, denen innerhalb eines Projekts Werte zugewiesen werden können. Diese Werte können aus den Templates heraus ausgelesen werden. Die Stereotype, auf die sich die Templates beziehen, sind in der Datei `profile.xml` festgelegt. Eventuell vorhandene Metafacaden werden in der `metafacades.xml` beschrieben. In diesem Dokument erfolgt u. a. die Zuordnung von Metafacades und Stereotypen. Auf Grund dieser Zuordnung erfolgt die Instantiierung von Metafacade-Klassen für Modellelemente durch den AndroMDA-Kern.

## 4 Entwicklung von J2EE-Anwendungen mit AndroMDA

### 4.1 AndroMDA J2EE-Cartridges

Die Java 2 Enterprise Edition (J2EE) hat weite Verbreitung als technologische Basis für die Entwicklung von verteilten Softwaresystemen gefunden. Sie steht jedoch in dem Ruf, komplex zu sein und ein Übermaß an plattformspezifischen Quellcode zu erfordern, der zusätzlich zur Fachlogik geschrieben werden muss. Ein Beispiel sind die Inter-

faces und Deployment-Deskriptoren, die zusätzlich zu einer Enterprise JavaBean (EJB) entwickelt werden müssen. Um dieser Komplexität Herr zu werden, werden parallel zwei Ansätze verfolgt: Einerseits finden Frameworks weite Verbreitung, die in Form einer API Modelle zur Anwendungsentwicklung vorgeben und Bibliotheken für die unterschiedlichen Schichten einer Anwendung zur Verfügung stellen. Einen Überblick über die wichtigsten Open-Source-Frameworks und ihre Einsatzgebiete bietet [RJ05]. Gleichzeitig werden Codegeneratoren für die J2EE immer beliebter. Einen Überblick über Codegenerierungsansätze bietet [JaHe03].

Im Rahmen des AndroMDA-Projekts werden Cartridges entwickelt, die alle Schichten einer J2EE-Anwendung abdecken. Diese generieren jeweils Quellcode für einzelne Frameworks. Im Folgenden soll ein Überblick über diese Cartridges gegeben werden, bevor in Abschnitt 4.2 die Entwicklung einer Beispielanwendung mit einer Auswahl dieser Cartridges dargestellt wird.

Für die Generierung der Datenschicht stellt AndroMDA zwei Cartridges zur Verfügung. Die erste Cartridge generiert CMP-EJBs (Container-Managed-Persistence Enterprise JavaBeans), die zweite generiert Quellcode für den Objekt-Relational-Mapper Hibernate, zu finden unter [Hib05].

Für die Modellierung der Geschäftslogikschicht dienen alternativ die EJB-Cartridge, die neben CMP-Beans auch Session-Beans generiert, oder die Spring-Cartridge, die Quellcode für das unter [SPR05] zu findende Spring Framework generiert.

Die Generierung der Präsentationsschicht ist auf Weboberflächen beschränkt. Hierfür gibt es die Bpm4Struts-Cartridge zur Verwendung mit dem Apache Struts Framework, zu finden unter [STR05]. Eine Cartridge zur Generierung von JavaServer-Faces-Code befindet sich aktuell noch in einem frühen Entwicklungsstatus und ist noch nicht Teil des stabilen AndroMDA-Release.

Weiterhin stehen Cartridges für die JBoss jBPM Workflow-Engine (siehe [jBPM05]) sowie für Web-Services mit Apache Axis (siehe [Axis05]) zur Verfügung.

Für allgemeine Java-Klassen, z. B. Exceptions oder JavaBeans, die übergreifend im Zusammenspiel mit den anderen Cartridges benötigt werden, wird die Java-Cartridge verwendet.

## 4.2 Entwicklung einer J2EE-Beispielanwendung

An Hand einer J2EE-Anwendung soll die Entwicklung mit AndroMDA beispielhaft dargestellt werden. Anforderung an die Anwendung ist insbesondere, die typischen drei Schichten einer verteilten Unternehmensanwendung zu umfassen. Die Anwendung verwendet dabei als Datenbank MySQL und als Applikationsserver JBoss AS. Dabei werden die Hibernate-Cartridge für die Datenschicht, die Spring-Cartridge für die Geschäftslogikschicht sowie die Bpm4Struts-Cartridge für die Webschicht eingesetzt.

Als Beispiel dient eine vereinfachte Version einer Praktikumsbörse, wie sie von Unternehmen auf ihren Webseiten betrieben werden: Ein Nutzer kann die offenen Praktikumsstellen als Liste betrachten und die Stellenausschreibungen in Form von PDF-Dokumenten herunterladen. Um sich auf eine Stelle zu bewerben, lädt er seine Bewerbungsunterlagen im PDF-Format hoch.

Aus Administratorsicht können neue Stellen ausgeschrieben und abgelaufene Ausschreibungen gelöscht werden. In der Übersichtsliste wird für jede Stelle die Anzahl der Bewerber angezeigt. Der Administrator kann eine Stelle auswählen, um sich die Bewerberliste anzeigen zu lassen. Von hier aus kann er die Unterlagen einzelner Bewerber herunterladen.

Aus Platzgründen werden im Folgenden nur Ausschnitte aus den Modellen zu dieser Anwendung dargestellt. Auch können nicht alle Details der Beispielanwendung beschrieben werden. Informationen über die weiteren Funktionen der verwendeten Cartridges finden sich auf der AndroMDA-Projekthomepage unter [And05].

Die Modellierung der Datenschicht für die Hibernate-Cartridge erfolgt als Klassendiagramm. Zur Speicherung der Stellen- und Bewerbungsdaten werden zwei Klassen mit dem Stereotypen «Entity» modelliert, die in Abbildung 4 dargestellt sind. Jeweils ein Attribut ist durch den Stereotypen «Identifier» als Schlüsselattribut gekennzeichnet. Ohne diese Kennzeichnung wird von der Cartridge ein Schlüsselattribut `id` angelegt.

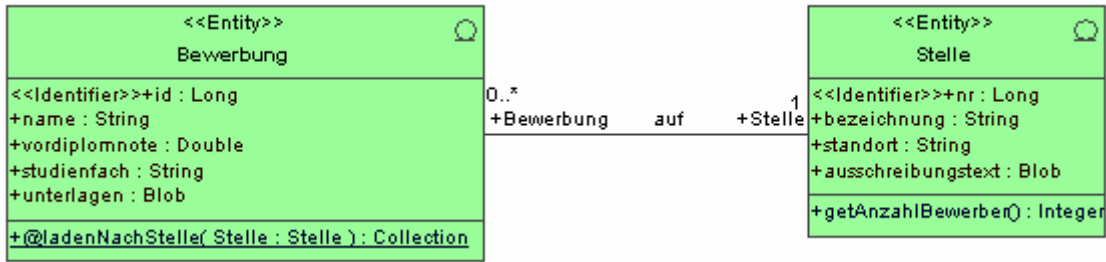


Abbildung 4: Modellierung der Datenschicht

Die Hibernate-Cartridge generiert aus diesem Modellausschnitt mehrere Dateien für jede Klasse. Es wird jeweils eine Java-Klasse mit Zugriffsmethoden für die Attribute generiert. Die Assoziation zwischen beiden Klassen resultiert dabei in entsprechenden Attributen. Um die Klassen mit Hibernate verarbeiten zu können, werden die Hibernate-Mapping-Dateien `Stelle.hbm.xml` und `Bewerbung.hbm.xml` generiert.

Da die Methode `getAnzahlBewerber()` der Klasse `Stelle` nicht generiert werden kann, muss diese von Hand in der Datei `StelleImpl.java` implementiert werden. Diese Klasse ist eine Unterklasse zur abstrakten Klasse `Stelle`, in der die betreffende Methode als abstrakt deklariert ist. Die Methode gibt die Größe der `Collection` `Bewerbung` zurück, die aus der Assoziation zwischen den Klassen generiert wird.

Um den Zugriff auf die persistenten Klassen zu erleichtern, generiert die Cartridge Zugriffsklassen nach dem Data-Access-Object (DAO) Muster, beschrieben in [Sun05a]. Für jede «Entity» Klasse wird dabei ein Interface, eine Klasse mit generierten Methoden (`DaoBase`) und eine Klasse als Rahmen für handgeschriebene Methoden (`DaoImpl`) generiert. Vollständig generiert werden Methoden zum Erzeugen, Laden, Löschen und Speichern der jeweiligen Objekte. Sie stehen sowohl für Einzelobjekte wie auch für eine `Collection` der Objekte zur Verfügung.

Die Methode `ladenNachStelle` ist auf Modellebene als `finder`-Methode mit dem Gültigkeitsbereich `classifier` deklariert. Die Hibernate-Cartridge erzeugt daraus eine Methode der Klasse `BewerbungDaoBase`, die die entsprechende Abfrage ausführt. Für komplexere Abfragen gibt es neben der Formulierung in OCL, die bereits in Abschnitt 3.2 dargestellt wurde, noch zwei weitere Möglichkeiten: Die Abfrage kann bereits auf Modellebene in der Hibernate-Query-Language (HQL) formuliert werden oder es wird die `Criteria`-API dafür verwendet. Die Umsetzung der zweiten Alternative erfolgt mittels des Stereotypen «Criteria» zur Kennzeichnung von Klassen. Die Attribute einer solchen Klasse fungieren als Abfragekriterien. Weitere Abfrageoptionen,

z. B. zur Sortierung der Ergebnisse, lassen sich über Eigenschaftswerte beeinflussen. Die beiden letztgenannten Varianten verringern jedoch die Plattformunabhängigkeit des Modells gegenüber einer Lösung mit der OCL, da zumindest die Datenbankabfragen spezifisch für die Hibernate-Cartridge sind. Weitere Informationen zur HQL und zur Criteria-API bietet [BK05, Kapitel 7].

Beide «Entity»-Klassen verfügen über ein Attribut zur Speicherung von Dateien als `byte-Array`. Zum Anzeigen einer Liste werden diese Daten nicht benötigt, trotzdem müssten sie durch die Schichten der Anwendung transportiert werden. Dies wird durch Verwendung des Value-Object-Musters, beschrieben in [Sun05c], umgangen. Auf Modellebene werden zwei weitere Klassen mit dem Stereotyp «ValueObject» erstellt. Eine Abhängigkeit von einer «Entity»-Klasse zu einer «ValueObject»-Klasse erzeugt die passende Zuordnung. AndroMDA generiert hieraus die entsprechende Java-Bean. Eine Methode zur Umwandlung der Objekte muss von Hand in der jeweiligen `DaoImpl`-Klasse ausgefüllt werden. In dieser Methode werden die Attribute des Value-Objects passend gesetzt. Die Methoden der `DaoBase`-Klasse werden automatisch dahingehend erweitert, dass durch Angabe eines zusätzlichen Parameters die Rückgaben in entsprechenden Value-Objects transformiert werden können.

Die Geschäftslogikschicht der Anwendung wird mit der Spring-Cartridge generiert. Von den in [WB05, S. 10] beschriebenen Funktionalitäten des Spring Frameworks nutzt die Cartridge die Kernfunktionalitäten zur Dependency-Injection sowie die Anbindung von ORM-Frameworks wie Hibernate. Die Modellierung für die Spring-Cartridge erfolgt mittels Klassen, die den Stereotyp «Service» tragen. Abbildung 5 stellt die entsprechende Klasse `PBoerseService` der Beispielanwendung dar.

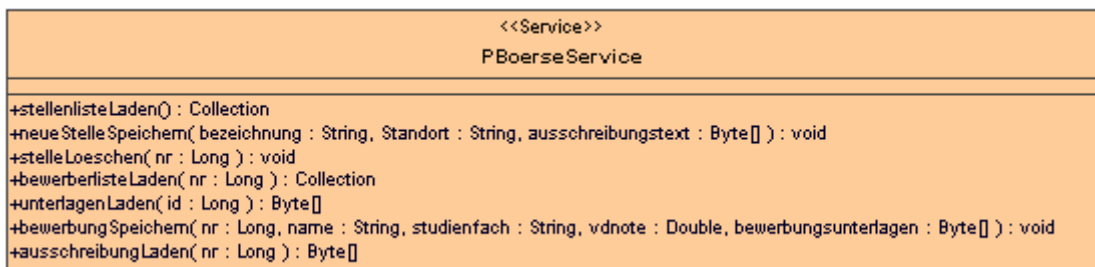


Abbildung 5: Modellierung der Geschäftslogikschicht

Für diese Schicht kann weniger Quellcode generiert werden als für die Datenschicht. Dies liegt daran, dass die Methodenrumpfe mit dieser Cartridge nicht generiert werden können. Generiert werden die Spring-Deskriptoren zur Definition der verwendeten

Klassen und Datenquellen, die EJB zur Kapselung der Service-Klasse und das Interface `PBoerseService`. Letzteres wird implementiert von der generierten Klasse `PBoerseServiceBase`. Hier wird für jede «Entity»-Klasse, zu der eine Abhängigkeit modelliert wurde, eine Zugriffsmethode für das entsprechende Data-Access-Object generiert. Die Klasse `PBoerseServiceImpl` wird als Rahmen für die Implementierung der Methoden der Service-Klasse generiert. Sie ist Unterklasse der `Base`-Klasse.

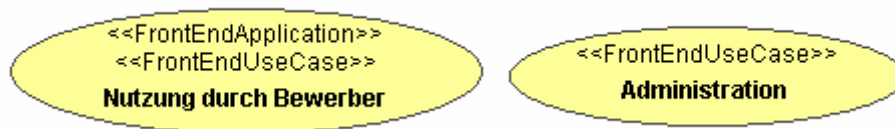
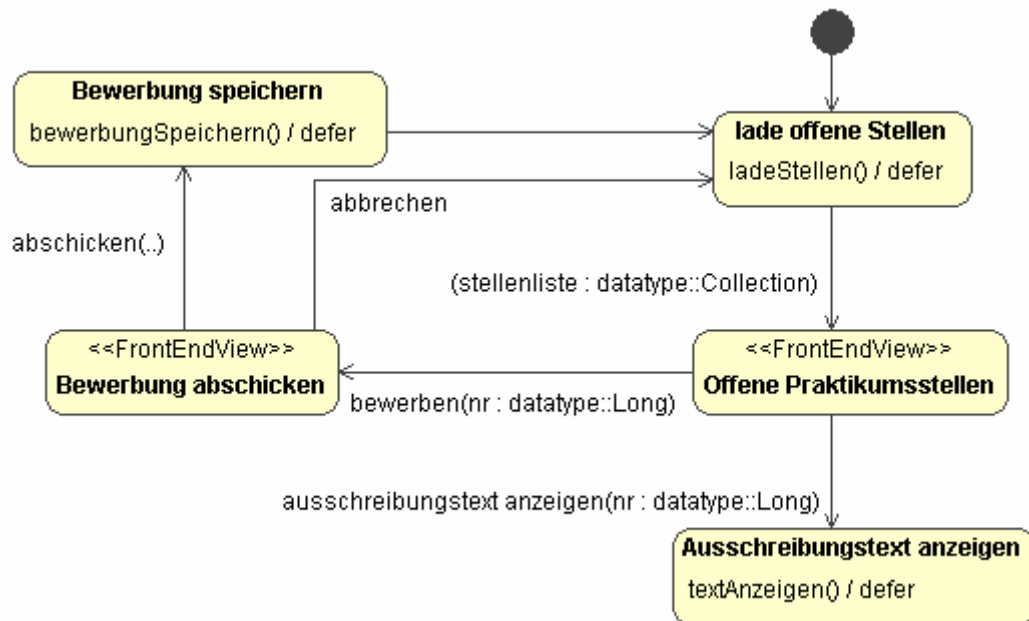


Abbildung 6: Anwendungsfalldiagramm der Beispielanwendung

Die für die Präsentationsschicht genutzte `Bpm4Struts-Cartridge` (Business Process Modeling for Struts) adressiert nicht nur die Struktur, sondern auch die dynamischen Aspekte der Anwendung. Ausgangspunkt der Modellierung ist die Unterteilung in Anwendungsfälle, dargestellt in Abbildung 6. Der Stereotyp «`FrontEndUseCase`» kennzeichnet einen Anwendungsfall für die Nutzung durch die Cartridge, «`FrontEndApplication`» kennzeichnet genau einen Anwendungsfall als Einstiegspunkt der Anwendung. Eine in diesem Beispiel nicht genutzte Möglichkeit ist die Generierung von Code für *Container-Managed-Security* durch Modellierung von Akteuren. Die Rechte von Nutzergruppen ergeben sich dabei aus Assoziationen zwischen Akteuren und Anwendungsfällen.

Jedem Anwendungsfall wird ein Aktivitätsdiagramm und diesem eine Controller-Klasse zugewiesen, Stereotype sind hierfür nicht nötig. Im Aktivitätsdiagramm wird der Anwendungsfluß beschrieben. Abbildung 7 stellt das Aktivitätsdiagramm zum Anwendungsfall *Nutzung durch Bewerber* dar. Hierbei fehlt aus Platzgründen die Darstellung der Eigenschaftswerte und einiger Parameter.

Jedes Aktivitätsdiagramm hat genau einen Anfangszustand. Endzustände tragen den Namen anderer Anwendungsfälle, zu denen eine Weiterleitung erfolgen soll. Es werden zwei Arten von Aktionszuständen unterschieden: Die mit «`FrontEndView`» gekennzeichneten Zustände führen zu einer clientseitigen Ausgabe, die Zustände ohne Stereotyp werden nur serverseitig ausgeführt.


 Abbildung 7: Aktivitätsdiagramm zu *Nutzung durch Bewerber*

Die Cartridge generiert für jeden «FrontEndView»-Zustand eine JavaServer Page (JSP), ein Cascading Style Sheet (CSS) zur Formatierung der Seite, eine Hilfeseite aus der Modelldokumentation, eine JavaBean für das Formular, Eintragungen in den Resource-Bundle-Dateien, Validierungsregeln, eine Struts-Action-Klasse und die entsprechenden Eintragungen in den Struts-Konfigurationsdateien.

Die Aktionszustände ohne Stereotyp werden dazu verwendet, auf Methoden der Geschäftslogikschicht zuzugreifen. Hierzu dienen die Methoden der Controller-Klasse, in diesem Fall der Klasse `NutzungController`, auf die in den Zuständen verwiesen wird. Die Cartridge generiert u. a. eine Klasse `NutzungControllerImpl`, in der diese Methoden von Hand implementiert werden müssen. Der Zugriff auf die Klasse `PBoerseService` erfolgt dabei nach dem Service-Locator-Muster, beschrieben in [Sun05b]. Hierfür muss eine Abhängigkeit von der Controller-Klasse zu der Service-Klasse modelliert sein.

Die Übergänge zwischen den Zuständen können über Signale mit Parametern versehen werden. Parameter, die in einen «FrontEndView»-Zustand eingehen (*page variables*), stehen in der generierten JSP-Seite zur Verfügung. Parameter, die von einem «FrontEndView»-Zustand ausgehen (*event parameters*), werden zu Parametern eines Formulars der Seite. Dabei wird ein Formular für jeden ausgehenden Übergang generiert. Für serverseitige Zustände werden eingehende Parameter an die jeweilige Methode übergeben. Ausgehende Parameter werden zu Parametern des Formulars des

folgenden Zustands und können somit aus der Controller-Methode heraus übergeben werden.

Wird einem «FrontEndView»-Zustand ein Parameter des Typs `Collection` übergeben, so wird mit der `Display Tag Library`, siehe [DTL05], eine Tabelle aus den Objekten gerendert. Über Eigenschaftswerte kann die Darstellung der Tabelle beeinflusst werden und ausgehende Übergänge können mit der Tabelle verknüpft werden. Im Beispiel wird die Liste der offenen Stellen über die Controller-Methode `ladeStellen` geladen und dem folgenden Zustand übergeben. In der Tabelle hat der Nutzer die Möglichkeit, sich über eine Schaltfläche auf eine Stelle zu bewerben oder über die Ausschreibungsnummer den Ausschreibungstext zu laden.

In den Aktivitätsdiagrammen stehen drei Möglichkeiten zur Verfügung, Verzweigungen zu modellieren. Die erste ist die oben bereits dargestellte Möglichkeit, mehrere von einem «FrontEndView»-Zustand ausgehende Übergänge zu modellieren. Die zweite ist die Bindung von Übergängen an Exceptions mittels des «FrontEndException»-Stereotyps. Weiterhin gibt es die Möglichkeit, Entscheidungspunkte zu modellieren, an denen die Rückgabe einer Controller-Methode ausgewertet wird. Verschiedene Rückgabewerte werden dabei verschiedenen ausgehenden Übergängen zugeordnet.

Zusammenfassend ergibt sich das folgende Bild bezüglich der vorliegenden J2EE-Cartridges: Für die Datenschicht sind die Methoden zur Umwandlung der Value-Objects und die Methode `getAnzahlBewerber` zu schreiben. Die Methoden der Geschäftslogikschicht müssen vollständig von Hand implementiert werden. In der Präsentationsschicht ist in jedem Fall eine Implementierung der Controller-Methoden notwendig. Zusätzlich kann eine Anpassung der JSP- und CSS-Dateien nötig sein. Dies betrifft sowohl die Anpassung des Layouts als auch die Erweiterung um Funktionalitäten, die über die Anzeige von Tabellen und Formularen hinausgehen. In vielen Fällen ist auch die Bearbeitung der Resource-Bundle-Dateien zur Anpassung der Ausgaben unumgänglich. Vorteilhaft ist, dass die Message Keys generiert und mit Standard-Werten vorbelegt werden.

Die Erweiterung um handgeschriebenen Quellcode kann schrittweise erfolgen, da alle Cartridges Platzhalter generieren. So bleibt die Anwendung zu jeder Zeit lauffähig. Soll der Anteil des generierten Quellcodes erhöht werden, so müssen die Cartridges erweitert werden.

## 5 Zusammenfassung, Bewertung und Ausblick

In der vorliegenden Ausarbeitung wurde das Open-Source-Tool AndroMDA vorgestellt. Nach einer Übersicht über die MDA folgte die Darstellung des Codegenerators sowie dessen Anpassung an verschiedene Plattformen mittels Cartridges. Da für die Entwicklung von J2EE-Anwendungen bereits ausgereifte Cartridges vorliegen, wurden diese im zweiten Hauptteil der Ausarbeitung untersucht.

AndroMDA stellt eine ausgereifte Lösung zum Einsatz der Model Driven Architecture in Softwareprojekten dar. Um über den Einsatz des Tools zu entscheiden, ist insbesondere eine Einschätzung des Aufwands zur Entwicklung der Cartridges notwendig. Hierbei ist zu prüfen, ob existierende Cartridges erweitert und neu entwickelte Cartridges in Folgeprojekten wieder verwendet werden können.

Da in dieser Ausarbeitung nur exemplarisch ein MDA-Tool vorgestellt wurde, können die Erkenntnisse nicht auf andere Tools übertragen werden. Insbesondere bleibt die Fragestellung offen, welche Vorteile der zweistufige MDA-Ansatz unter Einbezug eines plattformspezifischen Modells bietet.

Für die AndroMDA-Entwickler stehen, neben der Entwicklung weiterer Cartridges, die Integration mit anderen Tools und die Verbesserung der Bedienung momentan im Vordergrund: Unter der Projektbezeichnung Android wird eine grafische Benutzeroberfläche als Eclipse-Plug-In entwickelt. Diese integriert alle Funktionalitäten bis auf das Modellierungstool. Eine grafische Nutzeroberfläche befreit den Entwickler insbesondere von der Auseinandersetzung mit den interdependenten XML-Konfigurationsdateien. Außerdem wird der integrierte Template-Editor die Cartridge-Entwicklung erleichtern.

Im Bezug auf die Weiterentwicklung des AndroMDA-Kerns planen die Entwickler die Unterstützung der UML 2.0. Die bessere Ausführbarkeit der Modelle könnte die Möglichkeiten des Tools deutlich erweitern.

## Literaturverzeichnis

- [And05] AndroMDA: *Projekthomepage*.  
URL: <http://www.andromda.org>, Abrufdatum: 05. November 2005.
- [Axis05] Apache Axis: *Projekthomepage*.  
URL: <http://ws.apache.org/axis>, Abrufdatum: 05. November 2005.
- [BK05] Christian Bauer, Gavin King: *Hibernate in Action*, Manning, 2005.
- [BS03] Matthias Bohlen, Gernot Starke: *MDA entzaubert*, OBJEKTSpektrum 03/2003, S. 52-56, SIGS-DATACOM, 2003.
- [DTL05] Display Tag Library: *Projekthomepage*.  
URL: <http://www.displaytag.org/>, Abrufdatum: 05. November 2005.
- [Hib05] Hibernate: *Projekthomepage*.  
URL: <http://www.hibernate.org>, Abrufdatum: 05. November 2005.
- [JaHe03] Jack Herrington: *Code Generation in Action*, Manning, 2003.
- [jBPM05] JBoss jBPM: *Projekthomepage*.  
URL: <http://www.jboss.org/products/jbpm>, Abrufdatum: 05. November 2005.
- [KWB03] Anneke Kleppe, Jos Warmer, Wim Bast: *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison–Wesley, 2003.
- [MAV05] Apache Maven: *Projekthomepage*.  
URL: <http://maven.apache.org>, Abrufdatum: 05. November 2005.
- [MDR05] Netbeans Metadata Repository: *Projekthomepage*.  
URL: <http://mdr.netbeans.org>, Abrufdatum: 05. November 2005.
- [META05] AndroMDA: *Metafacades-Dokumentation*.  
URL: [www.andromda.org/andromda-metafacades/developing.html](http://www.andromda.org/andromda-metafacades/developing.html),  
Abrufdatum: 05. November 2005.
- [OMG03] Object Management Group: *Technical Guide to Model Driven Architecture: The MDA Guide*, v1.0.1, 2003.
- [OMG05a] Object Management Group: *OCL 2.0 Specification*, Version 2.0, 2005.
- [OMG05b] Object Management Group: *MOF 2.0 / XMI Mapping Specification*, v2.1, 2005.
- [RJ05] Rod Johnson: *J2EE Development Frameworks*, IEEE Computer 38 (1), S. 107-110, 2005.
- [SB03] Thorsten Sturm, Marko Boger: *Softwareentwicklung auf Basis der Model Driven Architecture*, HMD – Praxis der Wirtschaftsinformatik 231, S. 38-45, dpunkt.verlag, 2003.

- [SPR05] Spring Framework: *Projekthomepage*.  
URL: <http://www.springframework.org>, Abrufdatum: 05. November 2005.
- [STR05] Apache Struts: *Projekthomepage*.  
URL: <http://struts.apache.org>, Abrufdatum: 05. November 2005.
- [Sun05a] Sun Developer Network: *Core J2EE Patterns – Data Access Object*.  
URL: <http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>, Abrufdatum: 05. November 2005.
- [Sun05b] Sun Developer Network: *Core J2EE Patterns – Service Locator*.  
URL: <http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html>, Abrufdatum: 05. November 2005.
- [Sun05c] Sun Developer Network: *Sun Java Center J2EE Patterns – Value Object*.  
URL: <http://java.sun.com/j2ee/patterns/ValueObject.html>,  
Abrufdatum: 05. November 2005.
- [VEL05] Velocity: *Projekthomepage*.  
URL: <http://jakarta.apache.org/velocity/>, Abrufdatum: 05. November 2005.
- [WB05] Craig Walls, Ryan Breidenbach: *Spring in Action*, Manning, 2005.