
**TU Wien**


**Business Informatics Group**
Institut für Softwaretechnik und Interaktive Systeme

---

Thomas Hiebler

Fallgasse 1/11, 1060 Wien
thomas.hiebler@gmx.net

# Modellgetriebene Entwicklung von Webanwendungen aus Anforderungsspezifikationen

Magisterarbeit zur Erlangung des akademischen
Grades eines Magister der Sozial- und Wirtschaftswissenschaften

Betreuung:
o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Univ.-Ass. Mag. Manuel Wimmer

Wien, 13. Dezember 2007

# Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wien, 16. August 2007

Thomas Hiebler

# Danksagung

Mein erster Dank richtet sich an meine Familie (Erich, Heinrike, Michael und Erich), die mich sowohl finanziell als auch moralisch unterstützt haben. Ohne dies wäre das Studium nicht möglich gewesen. Einen besonderen Dank auch an meine langjährige Freundin Birgit, die sich oft genug meine 'Problemchen' anhören durfte und mich während meiner Studienlaufbahn tatkräftig unterstützt hat.

Weiters möchte ich mich bei meinen Betreuern o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel und Univ.-Ass. Mag. Manuel Wimmer bedanken, die mir genügend Freiraum gelassen und mich mit einer hervorragenden Kompetenz unterstützt haben. In diesem Sinne will ich mich auch beim Institut für Rechnergestützte Automation - Arbeitsgruppe INSO (Arr!) bedanken, das mich durch die langjährige Arbeit als Tutor geprägt und auf zukünftige Projekte vorbereitet hat.

Auch ein besonderer Dank geht an meine Studienkollegen und die 'Sinos', die mir während meiner Studienlaufbahn enormes Feedback gegeben haben und mich in erholsameren Zeiten gestützt, gehalten und falls notwendig auch heimgebracht haben. Special Thanks auch an Gertschi, der oft genug mein 'Gejammere' ausgehalten hat und mir in schwierigen Phasen der Arbeit mit Rat und Tat zur Seite gestanden hat.

Danke auch an alle, die ich jetzt an dieser Stelle vergessen habe.

# Kurzfassung

Betrachtet man den Trend der vergangenen Jahre, so ist die *modellgetriebene Softwareentwicklung*, kurz *MDSD*, der nächste logische Schritt in der Anwendungsentwicklung. Die leichte Verständlichkeit und die rasche Anpassung an wechselnde Implementierungstechnologien werden als großer Vorteil von MDSD gesehen.

In derzeitigen Softwareentwicklungsprozessen trifft man bei der Entwicklung von Anwendungen auf eine Vielzahl von Problemen. Die Dominierung von Fachlichkeit durch Technik, die Divergenz der Änderungszyklen und der methodische Bruch zwischen Analyse, Design und Implementierung sind einige nennenswerte Beispiele. Der modellgetriebene Ansatz versucht diese Probleme zu vermindern, indem er die ohnehin für die Entwicklung notwendingen Softwaremodelle als Sourcecode behandelt.

Diese Arbeit zeigt eine mögliche Lösung dieser Probleme durch die Verwendung eines modellgetriebenen Ansatzes. Das Ziel ist es, eine modellgetriebene Entwicklungsumgebung für Modellierung und Generierung von Webanwendungen auf Basis des JBoss Seam Frameworks, welches gängige Java-Technologien miteinander kombiniert, anzubieten.

Neben der Generierung der benötigten Backend-Datein aus UML 2.0 Strukturdiagrammen wird weiters die Generierung einer prototypischen Benutzeroberfläche aus UML Verhaltensdiagrammen gezeigt. Diese kann für Tests und als Ausgangsbasis für das Design der Webanwendung herangezogen werden. Neben dem Frontend und dem Backend der Anwendung wird auch der Qualitäts-Aspekt in der Arbeit behandelt. Hierbei werden Möglichkeiten gezeigt, wie Testinformationen aus den Modellen gewonnen werden können. Dies soll den Bereich der Softwarequalitätssicherung abdecken und den effektiven Testaufwand in einer realen Entwicklung reduzieren. Als Referenzanwendung dient ein Web-DVD-Shop, der für die Erstellung der Generatoren und Modelle herangezogen wird. Anhand dieser Referenzanwendung wird auch der derzeitige Automatisationsgrad durch MDSD besprochen.

# Abstract

Considering the major trends in the software engineering discipline, Model Driven Software Development (MDSD) is a promising advancement concerning the process of software development. MDSD supports both rapid adjustments regarding alternating implementation technologies and easy comprehensibility.

The development of software applications is frequently accompanied by a variety of problems like domination of expertise in technical matters, divergence of changing processes, and methodical gaps between analysis, design and implementation. By interpreting software models as source code, the model driven approach tries to reduce these problems.

This work explores a potential solution of the above mentioned problems by providing a model driven development environment for modelling and generating Web applications based on the JBoss Seam Framework. This framework has been chosen because it combines established Java technologies.

Besides receiving the required backend files from UML 2.0 structure diagrams, the prototypical user interface is automatically generated from the behaviour diagrams. This interface may be used as starting point for the design process of web applications. In addition to the automatic development of the frontend and backend of the application, this work also meets various quality aspects by showing possibilities of automatically extracting test cases from the developed models. The intention is to reduce the manual effort of testing throughout the whole development process. Exemplifying the development of the generators and models, a Web DVD store has been developed, which also exhibits the degree of automation by MDSD.

# Inhaltsverzeichnis

# 1. Einleitung

## 1.1. Kontext der Arbeit

Diese Arbeit befindet sich im Kontext der modellgetriebenen Softwareentwicklung. In dieser Arbeit wird die Kurzform MDSD angewandt, welche für den englischen Begriff *Model Driven Software Development* steht. Dabei wird versucht, möglichst viel Informationen aus den Modellen zu extrahieren, um diese architekturgerichtet für die Generierung der Anwendung zur Verfügung zu stellen. Die dafür zuständigen Generatoren besitzen meist Sprachkonstrukte zur Definition der einzelnen Arbeitsschritte, die für die Implementierung der Software notwendig sind.

Neben der Generierung der Anwendung stellt die Modellierung ein wichtiges Themengebiet im Bereich von MDSD dar. Diese Modelle auf Basis von Metamodellen sind der Antrieb der Generatoren und versuchen, die Inhalte der Anwendung auf abstrakte Weise zu erfassen. Die dazu nötigen Sprachelemente werden durch das verwendete Metamodell definiert.

## 1.2. Problemstellung

Im Durnstkreis des MDSD sind eine Vielzahl von Methodiken, Werkzeuge und Anwendungsszenarien entstanden. Die meistverwendetsten Standardlösungen fokusieren jedoch nicht auf die Erstellung einer vollständigen Applikation sondern decken nur Teilprobleme ab. Weiters bieten sie oft nicht die Möglichkeit von Einstellungen und Anpassungen an die gewünschte Umgebung. Als zusätzliches Problem sind Versionskonflikte zu nennen, da sich die Technologien meist schneller weiterentwickeln, als die dazugehörigen Generatoren.

Derzeit gibt es keinen Generator und kein Cartridge[1], das die Entwicklung einer JBoss Seam Applikation [JBo07f] ermöglicht. Zwar können Teilaspekte wie Java Beans automatisch erzeugt werden, jedoch existiert keine Möglichkeit der Generierung der Anwendung inklusive einer Benutzeroberfläche für die Bedienung. Aus diesem Grund versucht diese Arbeit, eine Komplettlösung für Seam Webanwendungen zu erzeugen, die neben der Erstellung der Anwendung auch zeigen soll, welche Open Source Möglichkeiten die modellgetriebene Entwicklung für konkrete Problemstellungen innerhalb spezieller Umgebungsanforderungen liefern kann.

## 1.3. Lösungsvorschlag

Als Lösung für das bereits beschriebene Problem wird eine Komplettlösung in Form von Generatoren angeboten, die aus bestehenden Diagrammen auf Basis von UML 2.0 eine Anwendung erzeugen. Die Generierung umfasst sowohl den Backend-Bereich für die Datenverwaltung und Steuerung der Anwendung, als auch den Frontend-Bereich mit der Benutzeroberfläche, die den Zugriff des Benutzers über den Webbrowser ermöglicht.

Die Verwendung von UML 2.0 [OMG07d] stützt sich auf die Tatsache, dass dies ein weitverbreiteter Industriestandard ist, der neben einer guten Werkzeugunterstützung auch eine Vielzahl von Diagrammen, welche über eine etablierte Notation verfügen, bietet. Zudem bietet UML Erweiterungsmöglichkeiten, wodurch zusätzliche Sprachelemente definiert werden können, die für die Modellierung von Webanwendungen notwendig sind. Mittels Profilen lassen sich die Basiselemente der MOF [OMG07a] zusätzlich um Elemente erweitern, sodass die Möglichkeiten in der Anwendung nahezu unbegrenzt sind.

Eines der Hauptprobleme von UML ist es, dass kein explizites Diagramm für die Darstellung von Webinhalten oder Seitenflüsse exisitert. Daher wird diese Arbeit in diesem Bereich ein mögliches Konzept eingeführt, wie man aus bestehenden Verhaltensmodellen trotzdem grundlegende Benutzeroberflächen generieren kann. Deren Erweiterung und manuelle Implementierung begrenzt sich ausschließlich auf visuelle Aspekte, die ohnehin für jede Problemstellung unterschiedlich sind.

[1] Cartridge.: Ein Cartridge ist ein Stück eines Generators, das zu einem kompakten Paket zusammengefügt wurde und so einfach exportiert und innerhalb anderer Umgebungen verwendet werden kann.

## 1.4. Wissenschaftlicher Beitrag

Die Beiträge zur Lösung der in 1.2 vorgestellten Probleme bietet diese Arbeit folgene Möglichkeiten:

- Seam-Backend-Generator für die Erzeugung der Daten- und Persistenzschicht sowie deren Zugriffsmechanismen.
- Seam-Frontend-Generator, der eine auf die funktionalen Elemente beschränkte Benutzeroberfläche erstellen lässt. Dieser ist als Prototyp zu sehen, dessen Konzept sich einfach erweitern und somit auf viele unterschiedliche Problemstellungen ausrichten lässt.
- Überlegungen hinsichtlich eines Qualitätsmanagement oder Testszenarien. Diese sollen die Entwicklung bei der Einhaltung der Qualitätsanforderungen unterstützen.
- Eine Analyse über die Arbeit sowie weitere Möglichkeiten der Erweiterung der einzelnen Generatoren. Zudem wird gezeigt, in welchem Bereich des Automatisierungsgrades eine sinnvolle Entwicklung im Sinne der MDA möglich ist.

## 1.5. Methode

Die Implementierung des Generators mit den Templates basiert auf einen zweigleisigen, iterativen Entwicklungsprozess, der später in Abschnitt 2.4 näher betrachtet wird. Dabei wird die Anwendungs- von der Generatorenentwicklung getrennt und separat behandelt. Nach der erfolgreichen Generierung der Anwendung werden die Ergebnisse analysiert. Die gewonnenen Erkenntnisse fließen als Feedback in die nächste Iteration.

Zur Ableitung der ersten Grundtemplates dient eine kurze Hello-World-Implementierung, die zudem die Machbarkeit des Generators demonstriert. Die Erkenntnisse aus dieser ersten Testimplementierung fließen in die erste Iteration der Generatorenentwicklung ein. Als nächstes wird eine Referenzapplikation analysiert und mit den Generatoren bestmöglich abgebildet. Nach jeder Iteration der Entwicklung werden die Ergebnisse betrachtet. Die gewonnenen Informationen fließen in die nächste Generatorversion ein. Hierbei wir nicht nur der Backend-

, sondern auch der Frontend-Bereich betrachtet. Zuletzt werden noch Konzepte aufgelistet, die das Qualitätsmanagement betreffen und ein modellgetriebenes Testen unterstützen sollen.

Die Referenzapplikation ist eine Anwendung auf Basis von JBoss Seam, die einen DVD-Store abbildet und eine Vielzahl von Technologien verwendet. Neben Frameworks wie Java Server Pages und Java Server Faces werden auch Facelets oder Java Business Process Management für die Erzeugung des Shops verwendet. Näheres zu den technischen Details wie Werkzeuge oder Frameworks wird später in der Arbeit vorgestellt.

Seitens der Industrie gibt es eine Vielzahl von Möglichkeiten und Werkzeuge in der modellgetriebenen Entwicklung. Neben Werkzeugen wie Rational Software Architect, openArchitectureWare oder AndroMDA gibt es weitere Produkte, die die Generierung auf Basis von Modellen unterstützen. Diese Arbeit basiert auf die Verwendung des Open Source Produkts openArchitectureWare [OAW07], welches in Kapitel 3 näher vorgestellt wird. Für die Implementierung und Modellierung werden Eclipse [Ecl07] und MagicDraw [Mag07] verwendet.

## 1.6. Verwandte Arbeiten

Im Kontext der modellgetriebenen Entwicklung von Webanwendungen gibt es einige nennenswerte Ansätze, die ebenfalls darauf abzielen, eine möglichst vollständige Anwendung aus den Modellen zu generieren. Ein direkte Vergleich mit dieser Arbeit ist jedoch nicht möglich, da sich die Methodik und der Modellierungsumfang doch deuchtlich von einander unterscheiden. In Folgenden werden zwei alternative Modellierungsansätze vorgestellt.

- **WebML:** Web Modeling Langage [Web07], kurz WebML, basiert auf einem eigenen Metamodell und generiert eine Anwendung über vier verschiedene Modelle. Dabei handelt es sich nach [Ald01] um einen datenzentrierten Ansatz, wobei die Struktur der Anwendung als Basis fungiert und die Web- und Präsentationslayer darauf aufbauen. Ein wichtiges Element von WebML ist somit das ER-Diagramm, welches die Objekte der Anwendung zeigt.

  Die Entwicklung mittels WebML besteht aus den folgenden vier Modellen:

  - Data model

- Hypertext model
- Presentation model
- The WebML process

Durch die Verwendung von WebML ist es möglich, den selben Inhalt für verschiedene Benutzer unterschiedlich aufzubereiten. Eine hundertprozentige Generierung der Anwendung aus den Modellen ist durch die unterstützte Erzeugung der Benutzerschnittstelle möglich. Die WebML bietet somit einen soliden Ansatz für die Generierung von Webanwendungen.

- **UWE:** UML-based Web Engineering, kurz UWE, stützt sich auf die Erweiterung der UML, um mehr Informationen in die benötigten Modelle zu bringen. Dabei werden spezielle Profile verwendet, die die gewünschten Funktionalitäten bieten. Für die Entwicklung werden eine Vielzahl von Modellen benötigt. Neben einem Konzeptions-, einem Navigations- wird noch ein Präsentationsmodell benötigt, um eine Anwendung daraus zu generieren. Die letzten zwei Modellarten werden die Hypertext-Ebene, also der logischen Verbindung der einzelnen Objekte, benötigt. Nähere Informationen zum Ansatz finden sich in der Literatur [Nor02], [Nor01].

  Obwohl der Ansatz sicher viele Möglichkeiten bietet, scheint er aufgrund der hohen Modellanzahl auf den ersten Blick sehr aufwendig zu sein. Eine genaue Evaluierung des Ansatzes findet sich beispielsweise in folgender Arbeit [Mar06].

Als mögliche Alternative zu den bereits vorgestellten Modellierungsansätzen lässt sich noch das Werkzeug ArcStyler [InO07] nennen, das ebenfalls eine Generierung von Webinhalten und Benutzerschnittstellen unterstützt. Neben der Generierung ist auch das sogenannte *'reverse engineering'* möglich, das aus bestehendem Java-Sourcecode Modelle ableiten kann. Die Generierung basiert auf einer eigenen UML- und MDA-Engine. Die Modellierung lässt sich mittels der Tool-Adapter-Standard-API definieren, sodass externe Modellierungsquellen herangezogen werden können.

## 1.7. Abgrenzung

Der Fokus der Arbeit richtet sich auf die Entwicklung der Generatoren für das Backend und das Frontend, wobei letzteres nur prototypisch erfasst werden soll. Dennoch soll das Konzept der Benutzeroberfläche verständlich sein und die Möglichkeit bieten, es für die unterschiedlichsten Anwendungsgebiete zu erweitern.

Zusätzlich sollen neben den Generatoren auch Möglichkeiten der Qualitätssicherung gezeigt werden. Dies umfasst primär ein Konzept, wie sich aus den bereits vorgestellten Diagrammen Testfälle und Systemtests ableiten und erzeugen lassen.

Aufgrund der Komplexität des Themas würde die Erstellung einer Komplettlösung für JBoss Seam Anwendungen den Rahmen dieser Arbeit sprengen. Lediglich der Backend-Bereich soll möglichst vollständig abgebildet werden; die Benutzeroberfläche und die Testmechanismen sind als Prototyp zu betrachten.

# 2. Einführung in die modellgetriebene Entwicklung

Dieses Kapitel soll die grundlegenden Aspekte der modellgetriebenen Softwareentwicklung näherbringen und dabei die wichtigsten Begriffe detailliert beschreiben.

Im Laufe der Jahre wurde Software und deren Entwicklung immer komplexer und dynamischer. Als Resultat der steigenden Komplexität wurden nach [Vol06] im Jahr 2003 noch immer zwei Drittel der durchgeführten Softwareprojekte nicht wie geplant vollendet. Um diesem Trend entgegenzuwirken und die Projekte erfolgreicher abzuschließen, wurden die angewandten Programmiersprachen und Methoden stets verbessert und an aufkommende Probleme angepasst. Durch die Weiterentwicklung der Programmierungsparadigmen ist es zu einem immer höheren Abstraktionsgrad gekommen, wobei die modellgetriebene Entwicklung, kurz MDSD, nun mit der Einbindung von Modellen dem aktuellen Streben in der Softwareentwicklung nach höherer Abstraktion folgt. Der Begriff des Modells wird in [Pet03] wie folgt beschrieben:

*'Ein Modell ist eine abstrakte Repräsentation von Struktur, Funktion oder Verhalten eines Systems.'*

In der Literatur sind viele Definitionen des MDSD verfügbar, welche den Kern der modellgetriebenen Softwareentwicklung mehr oder weniger detailliert beschreiben. Im Wesentlichen steht MDSD für Techniken zur automatischen Erzeugung von lauffähiger Software unter Verwendung von formalen Modellen. Diese werden direkt in den Entwicklungsprozess eingebunden, wobei deren Wertigkeit dem finalen Sourcecode gleichgestellt wird. Abbildung 2.1 zeigt, wie Modelle bereits jetzt in Lebenszyklus einer Anwendung verankert sind.

Unter formalen Modellen versteht man Modelle, welche einen Aspekt der Software möglichst vollständig beschreiben. Da sich nicht sämtliche Bereiche einer Software beschreiben lassen, muss sich das formale Modell an definierten Regeln

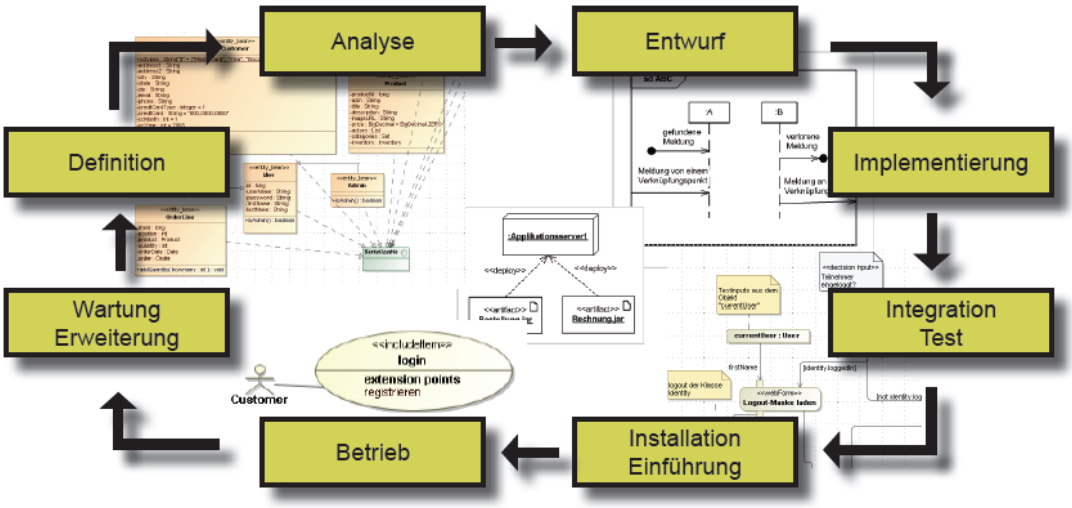


Abbildung 2.1.: Modelle im Softwarelebenszyklus [Vol06]

orientieren, die aufzeigen, welche Eigenschaften der Software dargestellt und welche vernachlässigt werden können. Eine weitere Eigenschaft von Modellen ist ihr abstrakte Charakter. Es werden Details ausgelassen, welche sich direkt auf die Implementierung beziehen. Obwohl es neue Editoren und Entwicklungswerkzeuge zulassen würden, ist es nicht im Sinne der MDA, direkte Implementierungen in die Modellierung zu bringen. Ein Einbetten dieser technischen Details würde die eigentlichen Informationen in den Diagrammen verfälschen und nicht in den eigentlichen Arbeitsbereich eines Modellierers passen. In einem solchen Szenario wären Probleme mit den Entwicklern und Programmierern unvermeidbar.

## 2.1. Unterschied zwischen MDA und MDSD

Im Zuge der modellgetriebenen Softwareentwicklung werden die Begriffe MDA und MDSD oftmals als Synonym verwendet. Dies ist aber nur bedingt möglich, da die beiden Begriffe teils unterschiedliche Aspekte der modellgetriebenen Entwicklung bezeichnen.

MDA steht für *Model Driven Architecture* und bezeichnet einen von der OMG [OMG07b] definierten Standard für die Erstellung von Software unter der Trennung von Spezifikation der Software und deren technischen Umsetzung. Dieser Standard [Sun07c] soll die Erstellung von Werkzeugen unterstützen, die folgende Funktionalitäten bieten:

- Spezifikation eines Systems unabhängig von dessen Plattform

- Spezifikation von Plattformen
- Auswahl einer bestimmten Plattform für das System
- Transformation einer Spezifikation in eine plattformabhängige Spezifikation

Während die MDA den modellgetriebenen Rahmen vorgibt, orientiert sich der Begriff des MDSD vielmehr an der eigentliche Entwicklung und der Vorgehensweise bei der Erstellung der Anwendung. Dabei stellt die MDSD einen Oberbegriff dar, der sich der konzeptionalen Aspekte der MDA bedient und diese erweitert. Neben Best Practices Ansätzen bezieht sich die MDSD auch auf mögliche Entwicklungsmodelle und praxisnahe Richtlinien bei der Entwicklung.

## 2.2. Vorteile und Ziele des MDSD

In der MDA-Spezifikation der OMG [Sun07c] werden drei wesentlichen Vorteile des modellgetriebenen Einsatzes in der Entwicklung genannt. Neben der Portabilität und der Plattformunabhängigkeit wird der Begriff der Wiederverwendung genannt. Nachdem MDSD einen weitgreifenden Aspekte im Bereich der modellgetriebenen Entwicklung abdeckt, wird an dieser Stelle auf die wichtigsten Vorteile beim Einsatz von MDSD eingegangen, wobei diese aufgrund der Nähe zur praktischen Entwicklung in einigen Punkten von der MDA Spezifikation abweichen können.

- **Hohe Abstraktionsebene:** Der hohe Abstraktionsgrad des MDSD ist nach [Pet03] einer der wichtigsten Entscheidungskriterien für die Verwendung. Die Modelle erlauben die Beschreibung der Applikation mit größeren, verständlicheren Elementen als es durch die bloße Verwendung einer Programmiersprache möglich wäre. Dies erweitert die Trennbarkeit von Modell und Implementation, da der fachliche Zusammenhang der Anwendung zwar im Modell dargestellt wird, die Implementierung jedoch mithilfe von Generatoren erfolgt, welche sich nur auf das Zusammenfügen von Bausteinen kümmern müssen. Die Generatoren bleiben somit frei von applikationsspezifischer Logik.

  Als Beispiel dient das Modellieren von 'Entitäten' anstatt von 'Klassen', welche abhängig von der jeweiligen Programmiersprache sind.

- **Architekturzentrierung:** Durch die Verwendung von Generatoren werden ar-

chitektonische Aspekte der Anwendung zentral verwaltet - im Generator und in den dazugehörigen Templates. Alle Elemente im Modell werden auf die selbe Art und Weise in die Programmiersprache übersetzt, wodurch die Architektur fixiert ist und sich auch während der Generierung nicht ändert. Dies erlaubt neben der einfachen Dokumentation der Softwarearchitektur auch das Vertrauen in dessen korrekten Einsatz. Brüche und Fehler in diesem Bereich werden somit ausgeschlossen.

Eine Änderung der Architektur ist ebenfalls leicht möglich, sodass diese gegebenenfalls an zentraler Stelle schnell verändert und übernommen werden kann.

- **Wiederverwendbarkeit:** Durch die Erstellung der Generatoren für die Transformationen werden wichtige Schritte im Entwicklungsprozess gespeichert. Diese lassen sich auf ähnliche Anwendungen, wie etwa Produktfamilien oder Systemlinien, anwenden, was ein enormes Potential im Sinne der Wiederverwendung bringt. Ein Generator kann für beliebig viele Anwendungen definiert werden, wodurch sich der Aufwand auf mehrere Projekte aufteilen lässt.

- **Interopabilität und Plattformunabhängigkeit:** Durch MDSD ist es möglich, von Herstellern und Plattformen weitmöglichst unabhängig zu bleiben. Die Verwendung der gebotenen Standards verringert die Bindung an Hersteller und erlaubt das Übertragen der Applikation in eine andere Umgebung.

  Die Vision der OMG der Plattformunabhängigkeit ist in der Regel nicht zu 100 Prozent durchführbar. Der Grund dafür ist in einer leichten Verschmelzung von Zielplattform und Modell zu sehen, da die Plattform auf die eine oder andere Art Einfluss auf das Modell hat.

- **Qualität-Aspekt:** Die Qualität der Anwendung kann mithilfe von MDSD gesteigert werden, da das System einheitlich und über den Generator zentral verwaltbar ist. Weiters hilft der modellgetriebene Aspekt bei der Dokumentation, da die Modelle eine zentrale Rolle spielen und diese ein wichtiger Bestandteil der Dokumentation sind.

  MDSD ist kein Garant für eine hohe Qualität, da einige Elemente zu der gängigen Entwicklung hinzukommen. Fehlerhafte Generatoren sind sehr schädlich, da sie die Fehler gegebenenfalls über die gesamte Anwendung verteilen, was die Fehlersuche erschwert und mehr Schaden verursacht, als eine

herkömmliche Entwicklung. Somit ist gerade der Bereich der Generatoren und Transformationen ein starker Risikofaktor im Sinne des Qualitätsmanagements.

- **Zeitersparnis:** Ein großer Vorteil von MDSD ist die mögliche Produktivitätssteigerung, die erzielt werden kann. Die Generierung von Sourcecode kann eine zeitliche Ersparnis bringen, wobei zu bedenken ist, das das Tippen von Quelltext in der Praxis einen eher geringen Teil der Entwicklung ausmacht. Vielmehr lässt sich über die Wiederverwendung der Generatoren sparen. Ein optimierter Generator steigert die Effizienz der Entwicklung und lässt große Teile der Anwendung ohne viel Aufwand erstellen.

  Weiters ist die Einbettung von Expertenwissen in den Generator sehr nützlich, da dieses Wissen über den Generator für alle Mitarbeiter verfügbar wird.

Nachdem nun die wesentlichen Vorteile genannt wurden, zeigt der nächste Abschnitt einen Einblick in die Methodik der modellgetriebenen Softwareentwicklung.

## 2.3. Allgemeine Konzepte des MDSD

Der Bereich der MDA wird seitens der OMG mit einer Vielzahl von Empfehlungen, Spezifikationen und Vorschlägen abgedeckt. Durch diese Verschmelzung kommt es zu einer Vielfalt an Begriffen und möglichen Konzepten im Dunstkreis des MDSD, welche später innerhalb dieses Abschnittes gezeigt werden.

Die Grundidee der modellgetriebenen Entwicklung besteht darin, die einzelnen Elemente der zukünftigen Anwendung bestmöglich zu trennen und zu einem geeigneten Zeitpunkt zu dem fertigen Produkt zu kombinieren. Im Gegensatz zu herkömmlichen Entwicklungsprozessen rücken die Modelle stark in den Vordergrund und werden als wichtiges Artefakt innerhalb des finalen Sourcecodes gesehen. Dabei gibt es verschiedene Sichten der Modelle, auf die jedoch später noch näher eingegangen wird.

Modelle beinhalten die fachlichen Informationen über die Problemstellung, lassen jedoch die implementierungsspezifischen Details vorweg. Somit kommt es zur klaren Trennung zwischen Fachlogik und Sourcecode. Diese Modelle sind nicht nur

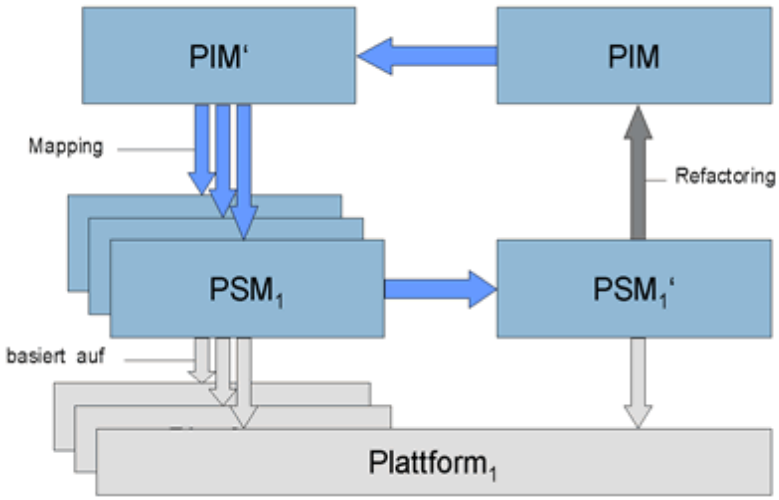


Abbildung 2.2.: Transformationszyklen der Modelle [Pet03]

abstrakt sondern weisen durch die Anbindung an eine sogenannte 'Domain Specific Language', kurz DSL oder Domänensprache, auch einen formalen Charakter auf. Diese kann beispielsweise über UML-Profile erstellt werden, die die Elemente der MOF [OMG07a] erweitern. Die DSL definiert die Ausdrucksmöglichkeiten der Modelle und ist an ein Metamodell der Domäne geknüpft.

Neben der Beschreibung der Anwendung mit Modellen ist die Generierung von Sourcecode der wohl wichtigste Teil in der modellgetriebenen Entwicklung. Dies erfolgt mithilfe von Generatoren und Interpretern, welche die Informationen aus den Modellen einlesen, auswerten, analysieren und schlussendlich Sourcecode erzeugen. Es gibt eine Vielzahl von Transformationsmöglichkeiten innerhalb der MDA. Neben der reinen Modell-zu-Code Transformation sind auch Modell-zu-Modell und Code-zu-Modell Transformationen in beliebiger Reihenfolge und Kombination möglich, wobei letztere meist schwieriger in der Durchführung sind, da der Code oft nicht viele abstrakten Informationen zur Erstellung der Modelle bietet. Abbildung 2.2 zeigt grob einen Einblick in die Möglichkeiten der Transformationen.

Der Generator ist mit dem Metamodell der DSL verknüpft und kann somit auf verwendete Sprachelemente zugreifen, was die Erstellung und Bedienung stark vereinfacht. Weiters bedient sich der Generator der Informationen der Plattform, welche die Anwendung stützen. Die klare Definition der Zielplattform hilft den Generatoren, da sie Bausteine für die Anwendung anbieten kann.

Nachdem nun auf die Zusammenhänge und die Grundkonzepte innerhalb die-

ses modellgetriebenen Ansatzes eingegangen wurde, werden einige Begriffe im Folgenen näher betrachtet.

### 2.3.1. Metamodell

Ein Metamodell ist eine Sammlung von Elementen, die innerhalb untergeordneter Modelle verwendet werden können. Es bildet somit eine Ansammlung von Möglichkeiten der Modellnotation. Die MDSD beschäftigt sich mit der formalen Beschreibung von domänenspezifischen Aspekten und muss in der Lage sein, die Struktur einer Domäne zu erfassen. Dies geschieht mittels Metamodellen, die Regeln festlegen, wie die benötigten Sprachelemente in den Modellen zu verwenden sind.

### 2.3.2. Domäne

Eine Domäne ist ein abgrenzbares Wissensgebiet. Im Sinne der modellgetriebenen Entwicklung ist sie durch das Vorhandensein von speziellen Sprachelementen von großer Bedeutung. Innerhalb des Kontextes der Domäne gibt es gewisse Sprachausdrücke, Methoden oder Begebenheiten, welche durch das Metamodel abgebildet werden. Hierfür gibt es den Begriff der *Domain Specific Language*.

Domänen können auch in kleinere, sogenannte Subdomänen unterteilt werden. Dies ermöglicht eine noch deutlichere Kapselung der DSL und erlaubt einen besseren Überblick bei komplexen Themen.

### 2.3.3. Plattform

Als Plattform bezeichnet man die Umgebung, in der eine Applikation ausgeführt wird. Diese bietet neben der definierten Systemarchitektur auch mögliche Schnittstellen, die anderen Systemkomponenten zur Verfügung stehen. Durch die Verwendung von mehreren Plattformen lässt sich ein sogenannter Plattform-Stack definieren, wobei die einzelnen Schichten miteinander kommunizieren und als Basis für eine Applikation dienen. Die Definition der Plattform hilft bei der Erstellung des Generators, der die Funktionalitäten der darunterliegenden Plattform benutzen und sich bei der Transformation auf die wesentlichen Softwareteile konzentrieren kann.

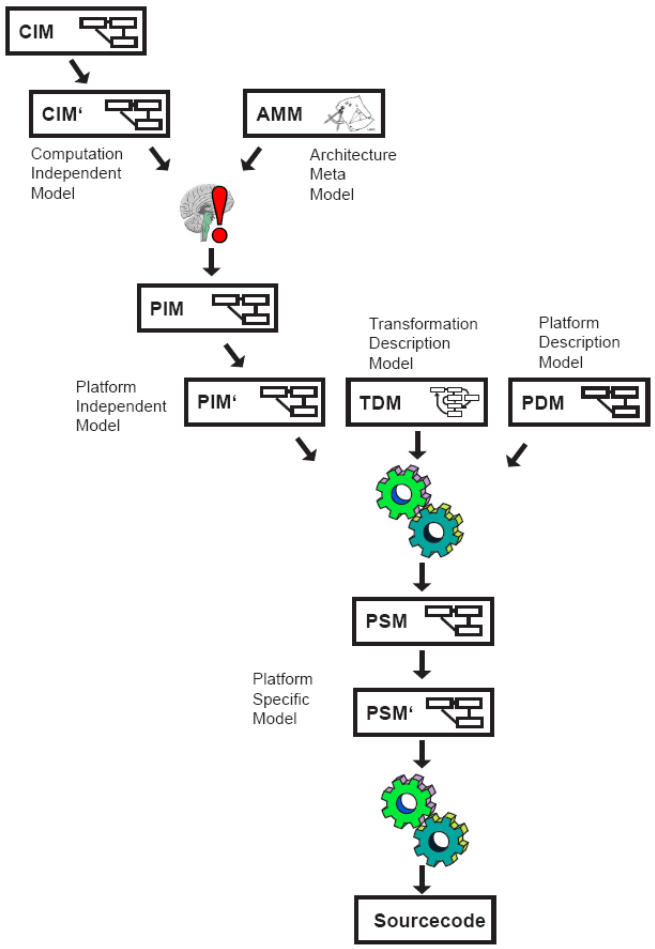


Abbildung 2.3.: Lebenszyklus von Modellen in der MDA

### 2.3.4. Abstraktionsebenen der MDA

Modelle sind der eigentliche Antrieb für die MDSD. Sie stellen die Anwendung möglichst formal dar, wobei sie sich den Elementen der Metamodelle und der DSL bedienen. Je nach Anbindung an eine Plattform und die Programmiersprache lassen sich die Modelle der MDA in drei unterschiedliche Sichten unterteilen. Basierend auf [Vol06] zeigt Abbildung 2.3 den Lebenszyklus der Modelle innerhalb der modellgetriebenen Entwicklung. Dabei beginnt die Systemabbildung mit dem *Computation Independent Model*, kurz CIM, und wird mit dem Wissen über die konkrete Zielarchitektur in ein *Platform Independent Model*, kurz PIM, umgewandelt. Nachdem die Plattform und die Transformationen beschrieben sind, werden diese Informationen mit dem PIM verschmolzen, wobei das *Platform Specific Model*, kurz PSM, entsteht.

Näheres zum Modellbegriff findet sich im MDA Guide [Sun07c] der OMG. Eine kurze Einführung in die unterschiedlichen Sichten bietet der folgende Abschnitt.

#### CIM - Computation Independent Model

Dieses Modell stellt einen Gesamtüberblick über das System dar. Es befinden sich keinerlei implementationsspezifische Details innerhalb dieser 'vogelperspektifischen' Darstellung. Durch seinen allgemeinen Charakter findet diese Art der Modellierung kaum eine direkte Anwendung in der modellgetriebenen Entwicklung. Sie dient vielmehr der Darstellung der Umwelt und der externen Umgebung zu anderen Systemen und unterstützt die Dokumentation der Anwendung. Oftmals wird diese Art der Modelle auch Business- oder Domainenmodell genannt.

#### PIM - Platform Independent Model

Das plattformunabhängige Modell stellt einen Schritt zwischen dem Domänenmodell und dem konkreten, auf die Implementierung bezogenen PSM dar. Hier finden sich die fachliche Logik und die Struktur der Anwendung wieder, wobei diese unabhängig von den darunterliegenden Systemkomponenten sind. Dies soll die Möglichkeit des Wechsels der verwendeten Plattform fördern.

Diese Modelle können in UML oder einer anderen, für die konkrete Domäne passenderen Modellierungssprache dargestellt werden.

#### PSM - Platform Specific Model

Durch das Einbringen von plattformspezifischen Details wird das PIM in ein PSM umgewandelt. Darin sind werden Details zur Implementierung möglich, welche später direkt über den Generator angesprochen werden können. Das PSM stellt somit den letzten Schritt innerhalb der Modellierungsebene dar.

Üblicherweise werden PSM in UML modelliert, wobei UML-Profile zur Anpassung der Sprache an die gewünschte Plattform verwendet werden. Alternativ sind auch ausführbare Modelle möglich, die direkt von Interpretern gelesen und verarbeitet werden können.

### 2.3.5. Modelltransformation

Die Transformation der Modelle stellt einen der Kernpunkte des MDSD dar. Hierbei werden Modelle in andere Modelle umgewandelt, wobei Sourcecode ebenfalls

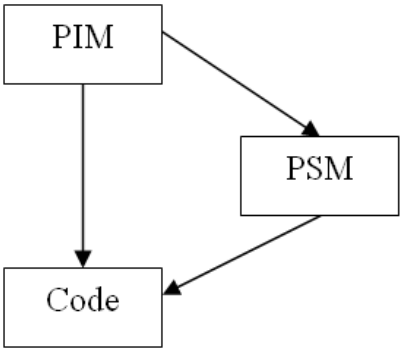


Abbildung 2.4.: Mögliche Abweichung der Modelltransformation

als Modell mit geringem Abstraktionsgrad angesehen werden kann. Im Allgemeinen unterscheidet man jedoch aus Gründen der Verständnis zwischen Modellen und ausführbarem Sourcecode. Je nach Zielmodell der Transformation definiert man Modell-zu-Modell, Modell-zu-Code und Code-zu-Modell Transformationen.

Bei der Umwandlung von Modellen kann es von Vorteil sein, wenn das Ursprungsmodell mit zusätzlichen Informationen angereichert wird. Dies kann über Kommentare, Annotationen oder Marken erfolgen, welche dabei helfen, das Abstraktionsniveau auf eine andere Ebene zu bringen.

Obwohl die OMG Spezifikation [Sun07c] eine Transformation von Modell-zu-Modell beschreibt, wird dies in der Praxis oftmals vernachlässigt, da die Automatisierung in diesem Bereich noch nicht ausgereift ist und nur von wenigen Werkzeugen angeboten wird. Stattdessen wird oftmals die Möglichkeit angewandt, das PIM zu annotieren und mit Informationen zu überladen, was die OMG in der Spezifikation nicht explizit ausschließt. Abbildung 2.4 zeigt diese Abkürzung in der Modelltransformation schemenhaft. Als Resultat reduziert sich der Modellierungsaufwand um eine Modellebene und der Sourcecode wird direkt aus dem annotierten PIM abgeleitet. Diese Methodik erleichtert somit die Entwicklung, da die Werkzeugunterstützung für diesen speziellen Fall bereits gegeben ist. Jedoch birgt dieser Ansatz die Gefahr, dass die Wartbarkeit und die Wiederverwendung darunter leiden und erschwert werden können. Nähere Informationen dazu finden sich unter dem Begriff *MDA-light* in der Literatur [Vol06].

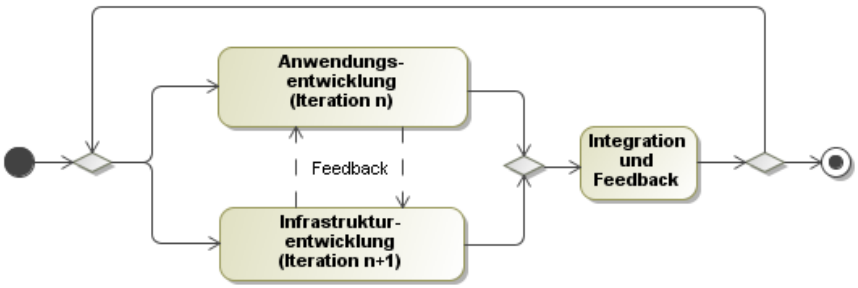


Abbildung 2.5.: Iterative, zweigleisige Entwicklung [Tho07]

## 2.4. Modellgetriebener Entwicklungsprozess

Im Laufe der Jahre haben sich im Bereich der Softwareentwicklung eine Vielzahl von unterschiedlichen Prozessen zur Erstellung der Software hervorgetan. Neben dem Rational Unified Process [RUP07] oder dem V-Modell [Mar05a] gibt es auch eine breite Palette von agilen Prozessen. Aufgrund der Tatsache, dass die modellgetriebene Entwicklung eine neue Art und Weise der Softwareerstellung darstellt, sind mögliche Kombinationen mit bereits vorhandenen Entwicklungsmodellen schwer zu bewerten. Iterative Modelle scheinen sich auf den ersten Blick besser in die Infrastrukturentwicklung des MDSD zu integrieren, wobei es mit Sicherheit auch möglich wäre, beide Bereiche mit einer agilen Arbeitsweise abzudecken. In der Literatur [Tho07] wird jedoch die iterative, zweigleisige Entwicklung empfohlen.

Diese Arbeit bedient sich einer iterativen Entwicklung, da diese meist überschaubarer und vorsichtiger ist. Agile Prozesse sind schwerer abzuschätzen und benötigen mehr oftmals Erfahrung in der Entwicklung. Beim verwendeten Entwicklungsmodell wird die Anwendung und die Infrastruktur schrittweise weiterentwickelt. Somit ergibt sich die Trennung in zwei Entwicklungsbereiche des Projektes, welche später näher betrachtet werden:

- Anwendungsentwicklung
- Infrastrukturentwicklung

Das Resultat liefert wichtige Erkenntnisse über beide Bereiche und hilft dabei, diese in der nächsten Iteration zu verbessern. Dies steigert die Sicherheit des Prozesses, da die Schritte überschaubar und leicht zurückführbar sind. Abbildung 2.5 zeigt diese iterative, zweigleisige Entwicklung. Dabei tritt die Feedback-Schleife in den Vordergrund, die wesentliche Kenntnisse über die Qualität des MDSD liefert.

### 2.4.1. Anwendungsentwicklung

Die Anwendungsentwicklung bezieht sich auf die eigentliche Erstellung der Anwendung, wobei sich die Entwickler an die definierte Domänenarchitektur halten. Sie kann in selbst in Iterationen ablaufen und durchläuft dabei nach [Tho07] folgende Schritte:

- **Anwendungsanalyse:** Hier werden die Requirements für die neue Iteration erarbeitet. Gegebenenfalls kann auch eine bereits erstellte Version der Anwendung analysiert und ausgewertet werden. Die gewonnenen Informationen werden als Input für das Design und die Modelle verwendet - um Probleme zu beheben oder Erweiterungen zu definieren.

- **Formale Modellierung und Design:** Dieser Schritt beschäftigt sich mit den Modellen und der Darstellung der Anwendung. Es treffen die Informationen der Analyse und jene der Domänendefinition aufeinander, was zufolge hat, dass neben der reinen Modellierung auch Optimierungsmöglichkeiten der DSL gefunden werden können. Als Output dieses Prozessknotens werden die formalen Modelle an die Transformationen weitergegeben. Weiters sind gefundene Verbesserungen der DLS an die Infrastrukturentwicklung zu leiten, damit die Sprachelemente der DSL angepasst werden können.

- **Transformation:** Die Transformation ist ein automatisch ablaufender Prozess der die Generatoren der Infrastrukturentwicklung zur Erstellung der Anwendung benutzt. Es wird keine neue Information in Bezug auf die Modelle oder die DSL gewonnen.

- **Manuelle Implementierung:** Da die Fachlogik meist nicht durch die Transformationen automatisierbar ist, muss sie nachträglich in die generierte Applikation eingefügt werden. Hierbei gibt es verschiedene Möglichkeiten um den manuellen vom generierten Code zu trennen. Möglichkeiten zu dieser Trennung finden sich in der Literatur ( [Vol06], [Rol06]) und würden an dieser Stelle den Rahmen sprengen.

Nachdem die Anwendung diese Schritte durchlaufen hat, kann sie ins System integriert und getestet werden. Die Feedback-Schleife bringt die Entwicklung in die nächste Iteration, wodurch sich der Reifegrad der Software erhöht.

### 2.4.2. Infrastrukturentwicklung

Die Infrastrukturentwicklung schafft das Umfeld der Anwendungsentwicklung. Es wird die DSL erzeugt, Referenzen entwickelt und nach Möglichkeiten der Wiederverwendung gesucht. Sie erstellt die Basis für die eigentliche Anwendung und stellt neben der Methodik auch die zu verwendenden Sprachelemente zur Verfügung.

In der Literatur sind hierzu verschiedene Modelle oder Prozesse zu finden, wobei die einzelnen Schritte ähnlich oder gleich sind. Im Folgenden werden die wichtigsten Schritte bei der Erstellung der nötigen Infrastruktur kurz vorgestellt. Diese lassen sich in drei Bereiche unterteilen, welche miteinander in Beziehung stehen. Details zu den einzelnen Punkten finden sich in der Literatur [Tho07].

- **Bereich der Domäne:** Dieser Bereich beschäftigt sich mit der Modellierung und der anzuwendenden Sprachelemente. Hier werden die Anforderungen analysiert, eine DSL definiert und die Modellierung anhand eines Referenzmodelles durchgeführt. Das Referenzmodell und das erstellte DSL Metamodell stehen in enger Beziehung mit den Transformationen des Generators, welcher sich auf die Darstellung und die verwendeten Modellierungselemente bezieht. Weiters werden mögliche Änderungen der DSL in Betracht gezogen, wodurch die Definition einer Schnittstelle zur DSL nötig wird. Dies kann über UML-Profile, XML-Dateien oder über einen eigens erstellten Editor geschehen, welcher in hochspezialisierten Domänen nützlich ist.

- **Bereich der Transformationen:** Neben den Transformationen und Generatoren ist dieser Bereich auch für die Erstellung einer Referenzimplementation zuständig, welche sich am Referenzmodell orientiert. Diese Implementation ist ein sehr wichtiges Instrument bei der Definition der Transformationen und liefert Kenntnisse über die Plattform, welche im nächsten Bereich entwickelt wird. Vor der Erzeugung der Referenzapplikation ist es durchaus üblich, einen kleinen Prototyp zu implementieren, der einer grob umrissenen Plattformdefinition entspringt und erste Erfahrungen sammeln lässt. Weiters kann man mithilfe eines Prototypen schnell architektonische Fehler erkennen, noch bevor viel Aufwand in die Referenz und die Generatoren gesteckt wurde.

- **Bereich der Plattform:** Der Bereich der Plattformen definiert die Zielplattform der Anwendung. Zuerst werden mögliche Frameworks und Komponenten definiert, welche im Prototypen verwendet werden. Die Informatio-

nen dieses ersten Implementationsversuches fließt mitunter ein in die finale Plattformentwicklung, welche im Zusammenhang mit der Referenzimplementation steht.

Wichtig bei der Entwicklung einer Anwendung ist es, den Infrastruktur-Iteration mindestens ein Mal durchlaufen zu haben, da sich diese mit den Grundelementen beschäftigt, welche die Implementation stark beeinflussen. Ohne Informationen, eine Zielplattform oder erste Transformationsableitungen ist es nicht möglich, die Applikationsentwicklung zu betreiben.

### 2.4.3. Vorgehensweise in dieser Arbeit

Wie bereits erwähnt richtet sich diese Arbeit nach dem zweigleisigen, iterativen Entwicklungsprozess.

Zur Erweiterung der DSL werden UML-Profile erstellt, da diese sehr simpel in der Handhabung sind und leicht erstellt werden können. Da nicht allzuviele Anpassungen nötig sind, wird kein eigener DSL-Editor entwickelt.

Als Prototyp für die Plattform dient eine simple Hello-World-Implementierung, welche die wichtigsten Elemente der Webapplikation, die Ein- und Ausgabe von Daten, abdeckt. Weiters lassen sich erste Transformationsregeln ableiten und Erfahrungen mit der Architektur sammeln, die der Abbildung der Referenzapplikation, die in Kapitel 5 vorgestellt wird, zu Nutze kommen.

Auf Basis der Referenzimplementierung wir das Referenzmodell erzeugt, welches laufend verbessert wird. Für die Lösung von auftretenden Problemen werden sowohl der Generator, als auch das Referenzmodell angepasst.

Nach der ersten vollständigen Iteration des Entwicklungsprozesses wird die Anwendungsentwicklung eingeleitet. Informationen aus dieser Entwicklung dienen als Feedback für die Modelle und Generatoren und helfen dabei, diese zu optimieren und auftretende Probleme zu beheben. Jede weitere Iteration verbessert die Generatoren und Templates und erhöht den Automatisierungsgrad, welcher das Verhältnis zwischen generierten und manuell entwickelten Codezeilen anzeigt.

# 3. Überblick über openArchitectureWare

openArchitectureWare ist eine auf Java basierende, Open Source Toolsammlung für den Bereich der modellgetriebenen Softwareentwicklung. Bei der Verwendung von Java wird bei der Entwicklung auf eine strenge Trennung der einzelnen Komponenten geachtet, wodurch openArchitectureWare modular und leicht auf die Systemumgebung anpassbar ist. Das Framework liegt derzeit in der Version 4.1 vor, welche im Zuge dieser Arbeit zum Einsatz kommt. Eine Weiterentwicklung auf Version 4.2 mit besserer Unterstützung bekannter Entwicklungsumgebungen und besseren Möglichkeiten der Modell-zu-Modell-Transformation ist bereits in Arbeit. Wichtig ist, dass es sich bei diesem Framework nicht um einen reinen MDA-Generator, sondern vielmehr um eine modulare MDSD-Plattform handelt, die sich anpassen und verändern lässt. Aus diesem Grund gibt es bislang nur wenige Cartridges, bereitgestellte DSLs mit den entsprechenden Templates, für den Generator. Als entscheidende Funktionalitäten lassen sich die folgenden vier Punkte bezeichnen:

- Einfache Integration in bestehende Umgebungen durch modulare Erweiterungsmöglichkeiten
- Open Source Softwaresammlung mit Unterstützung gängiger Formate als Quelldatein
- Steuerung der Entwicklung über einen Workflow-Engine
- Eigens entwickelte Templatesprache und Möglichkeit der Modellvalidierung

Den Kern von openArchitectureWare stellt der *Open Generator* dar. Dieser lässt sich mit modularen Komponenten verbinden und arbeitet die einzelnen Schritte der Entwicklung ab. Alle zusätzlichen Komponenten sind Standardimplementierungen und lassen sich einfach austauschen oder erweitern. Dadurch


Abbildung 3.1.: Aufbau des Generator-Frameworks [Rol06]

ist die Integration in bereits bestehende MDSD-Umgebungen möglich. Zusätzlich zum Generator existiert eine Workflow-Engine, die den Ablauf der Entwicklung über XML-Konfigurationsfiles steuern lässt.

Abbildung 3.1 zeigt den Aufbau und den Ablauf des Generator-Frameworks. Als erster Schritt wird ein Modell mithilfe des Parsers eingelesen. Dabei spielt das Format eine nebensächliche Rolle, da UML, EMF, XML, Microsoft Visio und viele andere Formate unterstützt werden. Es lässt sich innerhalb des Workflows über einen passenden Instantiator definieren. Das eingelesene Modell wird mithilfe des Metamodells instanziert - also jedes Element wird im Speicher als Blatt eines Objektbaumes abgelegt und lässt sich somit später nach beliebigen Kriterien iterieren.

Objekte im Speicher lassen sich mittels einer oAW-Komponente auf deren Gültigkeit überprüfen. Dies hat den Vorteil, dass Modellierungsfehler bereits gefunden werden, noch bevor sie durch den Generator verarbeitet werden.

Als nächster Schritt wird der eigentliche Sourcecode erstellt. Dabei werden die Templates durchlaufen, welche auf die Elemente der instanzierten Objekte zugreifen. Je nach Implementierung innerhalb der Templates werden die entsprechenden Textfragmente erzeugt.

Im folgenden Abschnitten werden die einzelnen Bereiche von openArchitectureWare kurz erläutert, um einen Überblick über dessen Funktionalitäten zu geben.

## 3.1. Workflow Steuerung

Die openArchitectureWare 4 Workflow Engine erlaubt nach [OAW06] die deklarative Steuerung des modellgetriebenen Entwicklungsablaufs. Die Steuerungselemente werden innerhalb einer eigenen Java Virtual Machine sequentiell als soge-

nannte *Workflow component* abgearbeitet. Ein Generator-Workflow arbeitet die einzelnen Teile ab, wobei es unterschiedliche Workflow-Komponenten wie Parser, Validatoren oder Transformatoren gibt.

Zur Definition des Ablaufes können die bereits implementierten Workflow-Komponenten verwendet werden. Diese werden mit der Distribution von oAW mitgeliefert und reichen für die meisten Entwicklungsprozesse aus. Durch die Modularität des Frameworks ist es jedoch möglich, eigene Komponenten zu definieren. Diese müssen nur das Interface der Workflow-Komponente *org.openarchitectureware.workflow.WorkflowComponent* implementieren und können beliebig verändert und angepasst werden.

Als Beispiel für eine Workflow-Steuerung zeigt Listing 3.1 die beispielhafte Workflow-Definition aus der Workflow-Referenz [OAW06]. Obwohl diese Workflowdefinition nicht lauffähig ist, zeigt sie dennoch die wesentlichen Aspekte des Ablaufes. Die definierten 'Properties' verweisen auf die Dateistruktur und geben die Quelle der Modelle und das Ziel der Generatoren an. Die erste Komponente verweist auf die Klasse des XMI-Readers und ist für das Einlesen des Modells zuständig. Der erzeugte Objektbaum wird mit der Generator-Komponente verarbeitet, dessen Resultate in den dafür definierten Ordner gespeichert werden.

```
<workflow>
    <property name='genPath' value='/home/user/target'/>
    <property name='model' value='/home/user/model.xmi'/>
    <component class='oaw.emf.XmiReader'>
        <model value='${model}'/>
    </component>
    <component class='oaw.xpand2.Generator'>
        <outlet>
            <path value='${genPath}'/>
        </outlet>
    </component>
</workflow>
```

Listing 3.1: Beispiel einer Workflow-Definition

Neben Parametern oder globalen Variablen können auch andere Workflows aufgerufen werden. Dies erlaubt das Einbinden von externen Workflows für die Verwendung von vordefinierten Cartridges.

## 3.2. Metamodelle und Profile

Die Metamodelle sind ein wesentlicher Bestandteil des MDSD. Wie bereits in Kapitel 2 gezeigt wurde, lassen sich Metamodelle mithilfe von Profildefinitionen einfach und modular erweitern. Mittels des openArchitectureWare-Konfigurationsfiles lässt sich die UML mit Profilen erweitern, ohne dabei von der UML-Spezifikation abzuweichen.

OpenArchitectureWare unterstützt eine Vielzahl von Eingabeformaten, welche im Workflow des Generators definiert werden müssen. Üblicherweise werden diese Modelle mithilfe von externen Modellierungswerkzeugen erstellt. oAW bietet jedoch auch die Definition mittels Java-Klassen an.

Listing 3.2 zeigt einen Ausschnitt des erstellten Workflows, welcher die Verwendung der Metamodelle und des Profiles zeigt. Im Beispiel werden zuerst das EMF und das UML2 Metamodell geladen. Zuletzt wird instanzierte Metamodell im Speicher um das Profil im UML2 Format erweitert, bevor es später weiterverarbeitet wird.

```
<component id="generator" class="oaw.xpand2.Generator"
   skipOnErrors="true">
  <metaModel class="oaw.type.emf.EmfMetaModel"
     metaModelPackage="org.eclipse.emf.ecore.EcorePackage
     "/>

    <metaModel class="oaw.uml2.UML2MetaModel"/>

    <metaModel id="profile" class="oaw.uml2.profile.
       ProfileMetaModel">
        <profile value="model/dvdstoreV16/profile.profile.
           uml"/>
    </metaModel>

    <!--  restliche Abläufe -->

</component>
```

Listing 3.2: Workflowabschnitt des Generators

Weiterführende Informationen zur Verwendung von Profilen oder der Erzeugung eigener Metadaten finden sich in [Rol06] oder [OAW07].

## 3.3. Xpand Templatedefinition

Im Zuge der Entwicklung des openArchitectureWare-Frameworks wurde eine eigene Templatesprache geschaffen, welche die Ausgaben des Generators steuert. Dabei verhält sich das Template recht flexibel und ist schwach typisiert - es überprüft die Typeninformationen der Ausdrücke erst zur Laufzeit und daher werden Typenfehler durch entsprechende Laufzeitfehler dargestellt. Ein Vorteil der Templates ist es, dass sei beliebig kombinierbar sind. Dies lässt eine modulare Aufteilung der einzelnen Verantwortlichkeiten zu und erleichtert die Übersicht und die Wiederverwendung.

Listing 3.3 zeigt einen Ausschnitt eines Templates zur Erzeugung einer Javaklasse. Das Beispiel wurde der Sprachreferenz der oAW entnommen [OAW07]. Es zeigt die wichtigsten Elemente wie Imports, Extensions und Elementdefinitionen.

```
«IMPORT meta::model»
«EXTENSION my::ExtensionFile»
«DEFINE javaClass FOR Entity»
    «FILE fileName()»
        package «javaPackage()»;
        public class «name» {
            // implementation
        }
    «ENDFILE»
«ENDDEFINE»
```

Listing 3.3: Struktur eines Templates

Eine Templatedatei besteht im Wesentlichen aus folgenden Sprachelementen:

- **Templatedefinitionen:** Innerhalb der Templates können Imports und Erweiterungen definiert werden. Imports lassen auf die Elemente des Namespaces zugreifen, auf die referenziert wird. Somit wird das Template anschaulicher, da lange Bezeichnungen verkürzt werden. Neben den Imports besteht auch die Möglichkeit, externe Objekte oder Funktionen in das Template einzubinden. Dies geschieht über das Element 'EXTENSION'.

- **Elementdefinitionen:** Der wichtigste Inhalt der Templates stellt die Definition der Elemente dar. Dies geschieht über das Wort 'DEFINE' und bietet Polymorphismus an. Sollte es zwei Definitionen mit gleichen Namen und unterschiedlichen Typen existieren, wird die passendste zur Laufzeit gewählt

und verarbeitet.

- **Aufrufe und Verzweigungen:** Diese Elemente rufen die verschiedenen Elementdefinitionen auf und steuern den Fluss durch den Objektbaum im Speicher. Neben 'IF' und 'FOREACH' gibt es noch eine Vielzahl von weiteren Möglichkeiten, deren genauen Erklärung in der Referenz [OAW07] zu finden ist.

In der Implementierung der Templates ist es durchaus sinnvoll, Teile oder Funktionen auszulagern, um diese zentraler zu verwalten oder mittels Java oder einer anderen Sprache manuell zu implementieren. Hierbei bietet sich die Verwendung von Xtend-Sprachelementen an, die im Folgenden Abschnitt behandelt werden.

## 3.4. Xtend Erweiterungen

Xtend stellt eine Erweiterung der Templates bereit, um Funktionen oder Methoden zu definieren. Die sind entweder schwer in der Templatesprache Xpand darzustellen oder werden an vielen Stellen der Templates benötigt, was für die zentrale Verwaltung innerhalb von Xtend-Dateien spricht.

Um statischen Javacode innerhalb der Templates zu verwenden, ist es nötig, ein Mapping zwischen Java und dem Template herzustellen. Dies geschieht über ein Xtend-File, welches in diesem Fall die Funktion der Schnittstelle annimmt. Folgendes Beispiel (siehe Listing 3.4), der Referenz des openArchitectureWare-Frameworks [OAW07] entnommen, zeigt, wie ein Xtend-File aussehen kann. Im Beispiel wird ein externes Metamodell und eine weitere Extension importiert, eine Expression an einem Stringparameter ausgeführt und eine Methode in Java aufgerufen.

```
import my::metamodel;
extension other::ExtensionFile;
/**
* Documentation
*/
anExpressionExtension(String stringParam) :
    doingStuff(with(stringParam))
;
/**
* java extensions are just mappings
```

```
*/
String aJavaExtension(String param) :
    JAVA my.JavaClass.staticMethod(java.lang.String
;
```

Listing 3.4: Erweiterungsbeispiel mit Xtend

Mithilfe von Xtend-Files lässt sich die Modularität erhöhen und auf Methoden in der Sprache Java zugreifen. Eine häufige Anwendung sind sogenannte *helper-classes*, die sich beispielsweise um ein Typenmapping zwischen Modell und Zielplattform kümmern oder Eigenschaften aus dem Metamodell extrahieren, die mittels Xpand nicht ermittelbar sind. Auch das Umgehen von Bugs und problematischen Bereichen des openArchitectureWare-Frameworks ist ein sehr beliebtes Anwendungsgebiet der Xtend-Sprache.

## 3.5. Check Modellvalidierung

In der modellgetriebenen Entwicklung ist es von entscheidender Bedeutung, dass die eingelesenen Modelle nicht nur korrekt, sondern auch valide gegenüber dem Metamodell sind. Denn nur valide Modelle können vom Generator richtigen Output liefern, da dieser sich ebenfalls den Informationen des Metamodells bedient. Aufgrund der Tatsache, dass viele Modellierungswerkzeuge nicht über eine Modellvalidierung verfügen, wurde diese in das oAW-Framework integriert.

Die Validierung erfolgt über separate Datein, die mittels Expressions-Sprachelementen die den bereits beschriebenen Xtend-Files ähneln. Wichtig bei der Erstellung der Check-Expressions ist es, den gültigen Zustand zu definieren und über Regeln auszudrücken. Listing 3.5 gibt einen kurzen Einblick in die Arbeit der Check-Modellvalidierung. Dieses Beispiel wurde der Referenz [OAW07] entnommen und zeigt, wie überprüft wird, ob eine Entität über einen Namen verfügt.

```
import my::metamodel;
extension other::ExtensionFile;
/**  Entities without a name are useless /
context my::Entity ERROR 'an entity must have a name!':
    this.name != null;
```

Listing 3.5: Beispiel eines Validierungs-Checks

Check-Befehle folgen dabei einem speziellen Syntax, der wie folgt aufgebaut ist:

- **Context-type:** Dieser Type stellt die Verbindung zu den Elementen des Metamodells her. Wird ein Element in dem übergebenen Modellelementen gefunden, welches dem geforderten Typ entspricht, wir der Check-Befehl auf dieses Element angewandt.

- **Guard-predicate:** Dieses Prädikat kann die Elemente über einen 'if'-Befehl weiter filtern. Unnötige Elemente lassen sich dadurch bereits im Vorfeld überspringen.

- **Warning/Error:** Hierbei handelt es sich um die Definition der Reaktion der Validierung. Bei einer 'Warning' wird am Ende der Generatorausführung eine Warnung in die Konsole geschrieben, die auf das Element referenziert und die eingestellte Nachricht ausgibt. Ein 'Error' hingegen stoppt den Generator vor der Ausführung und weist auf den Fehler hin.

- **Message-expression:** Hierbei handelt es sich um die Ausgabemeldung, die bei Warnungen oder Fehlern ausgegeben wird.

- **Predicate:** Das Prädikat ist das eigentliche Validierungsinstrument und überprüft das eingehende Element. Fällt das Ergebnis der Überprüfung negativ aus, werden die definierten Folgemaßnahmen ausgeführt.

## 3.6. Arbeit mit oAW

Das openArchitectureWare-Framework lässt sich vollständig in die IDE Eclipse [Ecl07] integrieren, was über die in Eclipse übliche Installation eines Plug-Ins möglich ist. Die Installationsdatein können entweder über die Homepage von oAW oder direkt in Eclipse über die Update-Plattform geladen werden. Durch die Installation werden Bearbeitungseditoren und Viewer für Xpand-, Xtend- und Check-Files zur Verfügung gestellt, die innerhalb der Eclipse-Plattform funktionieren. Neben diesen Editoren gibt es auch noch Beispielprogramme und Wizards, die den Einstieg und die ersten Schritte erleichtern. (siehe Abbildung 3.2)

Neben dem eigentlichen openArchitectureWare-Plugins werden noch einige weitere, frei verfügbare Systemkomponenten gebraucht, bevor oAW einsatzbereit ist. Hierzu zählen das UML- und weitere Modellierungs-Plugins. Eine Auflistung der

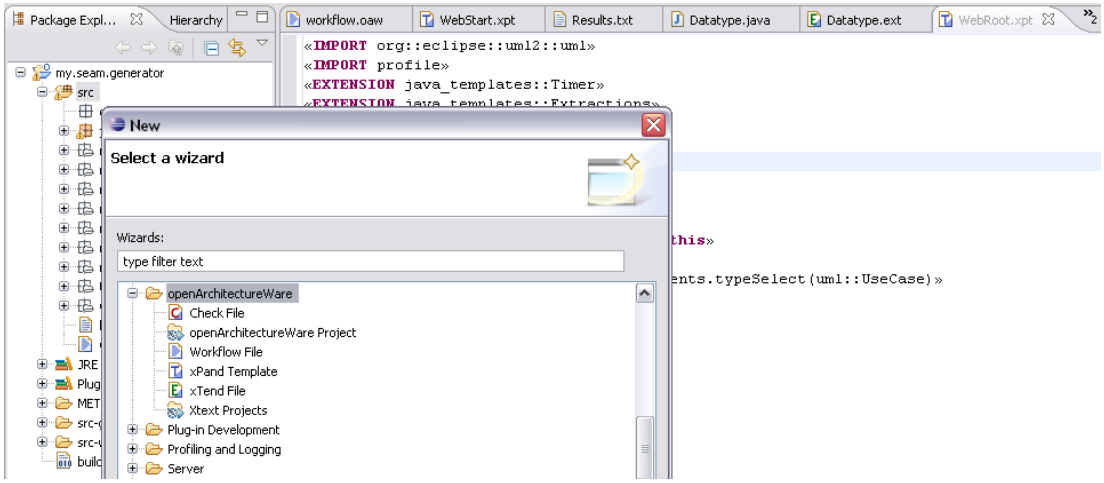


Abbildung 3.2.: Eclipse Wizards und Template-Ansicht

verwendeten Plugins befinden sich auf dem beigefügten Eclipse-Workspace im Anhang dieser Arbeit.

Für die Erstellung dieser Arbeit wurde openArchitectureWare in der Version 4.1 verwendet. Hierbei wurden die Templates des Generators mit der zur Verfügung gestellten Templatesprache Xpand erstellt. Während der Entwicklung der Templates wurde besonders auf die Modularität und auf die strickte Trennung der Bereiche des Front- und Backends der Applikation geachtet. Dies ermöglicht eine separate Nutzung der Generatoren was nicht nur das Testen und die Fehlerbehandlung erleichtert, sondern auch mögliche Erweiterungen problemlos durchführen lässt.

Zusätzlich zu den Xpand-Files wurde auf die Xtend-Möglichkeit zugegriffen, um statische Java-Methoden für die verschiedensten Problemlösungen zu verwenden.

Die erstellten Check-Files runden die Arbeit ab und erlauben, das eingelesene Modell zu validieren und die wichtigsten Problemstellen im Vorfeld aufzudecken. Es wurden dabei jedoch nur einige Fälle behandelt, da zusätzlich ein Abfangen von Fehlern zur Laufzeit in den Java-Methoden stattfindet.

Die Steuerung des Arbeitsablaufes funktioniert über das erstellte Workflow-File, welches die Bereiche des Front- und Backends trennt und somit eine Fehlersuche beziehungsweise einen separaten Einsatz erleichtert. Nähere Informationen zu den Bereichen und den Details der Arbeit mit openArchitectureWare findet sich in den entsprechenden Kapiteln 6 und 7.

# 4. Einführung in die UML 2.0 Modellierung

Dieses Kapitel beschäftigt sich mit der Modellierung der Anwendung. Hierfür wird die Unified Modelling Language, kurz UML, in der Version 2.0 verwendet, was entscheidende Vorteile hat:

- Die OMG Spezifikation sieht im Bereich der modellgetriebenen Entwicklung die Verwendung von UML vor.
- UML leitet sich von Metamodellen ab, welche die Erstellung von eigenen Modellierungssprachen erlauben. Dies ist durch das MOF Metamodell möglich.
- UML ist ein etablierter Standard seitens der Industrie und wurde in der aktuellen Version um wichtige Aspekte und Diagramminformationen erweitert, die des MDSD dienlich sind.
- Die Werkzeugunterstützung ist durch die weite Verbreitung von UML sehr ausgereift.

UML ermöglicht die Modellierung einer Problemstellung und anwendungsspezifischer Eigenschaften. Es umfasst mehr als die bloße Darstellung und die extakte Notation der Elemente, sondern erlaubt die Spezifikation und Visualisierung von Begebenheiten über vordefinierte Diagramme. Dabei gibt es die unterschiedlichsten Modelle für eine Vielzahl von Problemstellungen. Dabei kann man die Diagramme in Struktur- und Verhaltensdiagramme unterteilen, wobei in Summe dreizehn unterschiedliche Diagrammarten zur Auswahl stehen.

- **Strukturmodelle:** Diese umfassen das Klassen-, das Kompositionsstruktur-, das Komponenten-, das Verteilungs-, das Objekt- und das Paketdiagramm.

- **Verhaltensmodelle:** Hier unterscheidet man in das Anwendungsfall-, das Ak-

tivitäts-, das Interaktionsübersichts-, das Zeit-, das Sequenz-, das Kommunikations und das Zustandsdiagramm.

MDSD erlaubt den Einsatz jeglicher Modelle, die sich innerhalb einer definierten Metaebene bewegen. Das Metamodell entscheidet über den Einsatz von visuellen Objekten in der Darstellung und erlaubt den Zugriff des Generators auf die unterschiedlichen Elemente. Um neue Bausteine innerhalb eines Modells zu verwenden, gibt es verschiedene Möglichkeiten.

- **Anpassen der Meta Object Facility:** Die sogenannte *Meta Object Facility*, kurz MOF, stellt das Metamodell von UML dar. Hier werden die grundlegenden Elemente definiert, die in jedem Diagramm verwendet werden können, welches die MOF als Metamodell nutzt. Eine Änderung innerhalb der MOF-Metaebene ist jedoch nicht empfehlenswert, da einfachere Möglichkeiten gibt, die weniger Probleme wie etwa die Inkompatibilität mit externen Werkzeugen oder Modellen mit sich führen.

- **Definition innerhalb der UML-Metaebene:** Das Metamodell der UML lässt sich ebenfalls anpassen, um neue Bausteine zu erzeugen. Hierbei ist jedoch wichtig, dass das neue Metamodell eine nicht-standardisierte Version der UML darstellt. Als Probleme könnten sich eine fehlende Werkzeugunterstützung oder Probleme in der Datenverarbeitung ergeben. Auch ein besseres Wissen über die Metamodellierung wird bei der Änderung des Metamodells von nöten. Dieser Ansatz wird in der Literatur als *Heavyweight* bezeichnet [Rol06]

- **Definition von UML Profilen:** UML erlaubt die Erstellung von sogenannten Profilen, die das Metamodell erweitern. Es handelt sich hierbei um eine Ansammlung von Stereotypen, die auf die bestehenden Konstrukte des Metamodells aufbauen, Eigenschaften erben und Neue definieren lassen. Die Verwendung von Profilen ist modular, was es erlaubt, eine Vielzahl von unterschiedlichen Profilen je nach Verwendungszweck zu erstellen und bei Bedarf einzubinden.

Im Folgenden werden die verschiedenen Diagramme, die im Zuge dieser Arbeit verwendet wurden, in Anlehnung an [Mar05b] näher erklärt.

## 4.1. Klassendiagramm

Dieses Diagramm dient der Darstellung der Struktur, dient also der Spezifikation von Daten oder Objekten im System. Daher wird es oftmals auch mit dem Begriff des Objektdiagramms bezeichnet, was die Objektorientierung hervorhebt. Aus der Welt der objektorientierten Entwicklung ist dieses Diagramm nicht mehr wegzudenken, da es simpel in der Erstellung und sehr effektiv in der komplexen Darstellung ist. In Klassendiagrammen lassen sich neben Operationen und Attribute auch eine Vielzahl von weiteren Eigenschaften modellieren. Zusammenhänge von Objekten sind via Generalisierungen, Assoziationen oder anderen Beziehungs-Sprachelementen realisierbar. Der nächste Abschnitt liefert einen Einblick in die Modellierungselemente von Klassendiagrammen. Hierbei wurde aus Rücksicht auf den Umfang auf eine komplette Aufzählung verzichtet und nur die in der Arbeit verwendeten Elemente gezeigt. Weiterführende Informationen finden sich in der Literatur [Mar05b], [OMG07d].

### 4.1.1. Sprachelemente

#### Klassen

Klassen sind das primäre Modellierungselement des Klassendiagramms und stellen die einzelnen Objekte der Anwendung dar. Eine Klasse wird als Rechteck dargestellt und ist in horizontale Abschnitte unterteilt, wobei jeder Abschnitt für bestimmte Eigenschaftsdefinitionen steht.

Der obere Bereich definiert den Namen und zugewiesene Stereotypen der Klasse. Auch Annotationen oder klassenspezifische Eigenschaften finden sich in diesem Bereich. Der mittlere Abschnitt ermöglicht das Einfügen von Klassenattributen und deren Regeln, wie zum Beispiel Typdefinitionen, Multiplizitäten und Sichtbarkeitsregeln. Der letzte Bereich dient als Platzhalter für Operationen und Rückgabewerte.

Abbildung 4.1 zeigt die Klasse des Users der Referenzapplikation.

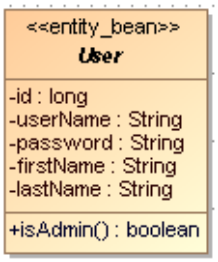


Abbildung 4.1.: User Darstellung im Klassendiagramm

### Generalisierungen

Diese Art von Verknüpfung von Klassen dient der Vererbung von Attributen oder Operationen. Sie werden als gerichteter Pfeil dargestellt, dessen Spitze ungefüllt ist. Generalisierungen sind ein wichtiges Instrument in der objektorientierten Entwicklung und erlauben das wiederverwenden von ähnlichen Objekteigenschaften.

### Constraints

Einschränkungen sind eine Erweiterungsmöglichkeit der UML und bietet die Möglichkeit, eigene Aspekte und Informationen zu definieren. Sie werden meist als Notiz oder Kommentar an ein gewünschtes Modellierungselement gehängt und lassen sich in OCL [OMG07c] oder natürlicher Sprache beschreiben.

### Assoziationen

Assoziationen stellen Beziehungen von Objekten dar, wobei diese in der Regel starr und unabhängig von durchgeführten Operationen der Objekte sind. In der Literatur wird oftmals das Wort 'langfristig' in diesem Zusammenhang erwähnt. Es gibt eine Vielzahl von unterschiedlichen Arten von Beziehungselementen, die unterschiedliche Komplexitäten in der Struktur darstellen lassen. Sie werden üblicherweise als Verbindungslinie zwischen Objekten modelliert und können je nach Art verschiedene Abänderungen in der Notation unterlaufen.

### Abhängigkeiten

Mit Abhängigkeiten lassen sich zwei unterschiedliche UML Modellelemente miteinander koppeln. Auch die Kopplung von mehreren Elementen ist möglich. Übli-


Abbildung 4.2.: Kundenverhalten im Anwendungsfalldiagramm

cherweise werden Klassen, Interfaces oder Pakete miteinander verknüpft, wobei auch jedes andere Element möglich ist. Es gibt drei Kategorien von Abhängigkeiten, wobei an dieser Stelle nicht näher auf die Unterschiede eingegangen wird. Abhängigkeiten werden als strichlierter Pfeil dargestellt.

#### Interfaces

Eine spezielle Art von Abhängigkeit stellt die Verwendung von Interfaces dar. Dieses Konzept wurde der komponentenorientierten Programmierung entnommen und stellt eine Erweiterung einer Klasse dar. Interfaces werden wie Klassen dargestellt, weisen jedoch den entsprechenden Interface-Stereotyp auf. In Anlehnung an Microsofts *Distributed Component Object Model* [MSD07] ist die Darstellung mittels Buchse und Stecker erlaubt. Hierbei handelt es sich um einen Stiel mit Kreis, der je nach Bedarf voll oder halb gezeichnet ist.

## 4.2. Anwendungsfalldiagramm

Das Anwendungsfalldiagramm dient der Beschreibung von Tätigkeiten oder Verhalten des Systems. Dabei kommt es zur Interaktion zwischen der Umgebung und den Elementen des Systems. Innerhalb der Systemgrenzen können Verhalten modelliert werden, die von externen Objekten, meist menschliche Akteure, verwendet oder bedient werden. Abbildung 4.2 zeigt die Darstellung eines Anwendungsfalldiagrammes anhand des Kunden der Referenzapplikation.

Zu beachten ist, dass die Verhalten nicht abhängig von zeitlichen Aspekten oder Abläufen sind. Für diese Art der Darstellung sind andere Modelle in der UML vorgesehen, wie das Kollaborationsdiagramm, das Sequenz- oder das Aktivitätsdiagramm. Anwendungsfalldiagramme bieten einen Überblick über die Verhal-

tensmöglichkeiten des gesamten Systems oder Subsystemen. Eine Verknüpfung oder Modularisierung der Diagramme ist möglich.

### 4.2.1. Sprachelemente

#### Akteure

Akteure sind Objekte, die das modellierte System benutzen. Sie befinden sich stets ausserhalb der Systemgrenze und stehen in Verbindung mit den dargestellten Anwendungsfällen. Diese können von den Akteuren ausgeführt werden, aber das System kann auch die Akteure benutzen, um Informationen einzuholen oder andere Aktionen anzustoßen.

Akteure werden durch ein Strichmännchen modelliert, dessen Bezeichnung unterhalb des Symbols angebracht ist. Es ist jedoch auch möglich, Akteure als Klassen darzustellen. Hierbei ist der Stereotyp 'actor' zu verwenden, der auf die Verwendung der Klasse als Akteur explizit hinweist.

Eine Zuweisung von Beziehungen zwischen verschiedenen Akteuren ist ebenfalls möglich. Somit lassen sich Generalisierungen abbilden, um beispielsweise Verhaltensweisen zu vererben. Weiter ist die Interaktion eines abstrakten Akteurs modellierbar, der repräsentativ für eine Menge von realen Akteuren mit dem System kommuniziert, was die Übersicht erhöhen kann.

#### Anwendungsfälle

Anwendungsfälle stellen bestimmte Verhalten oder Verhaltensmöglichkeiten des Systems dar. Die Summe aller Anwendungsfälle zeigt das vollständige Systemverhalten. Sie sind ähnlich wie Akteure auch dem Metatyp 'Classifier' zugeordnet, was es erlaubt, Eigenschaften zu definieren. Somit können Anwendungsfälle mit jeglicher Information ausgestattet werden, die auch im Klassenmodell möglich wären. Es sind dadurch auch Verschachtelungen oder Zuweisungen zu anderen Klassen möglich.

Es gibt unterschiedliche Möglichkeiten in der Darstellung von Anwendungsfällen. Gleich ist ihnen jedoch eine Ellipse in der Nähe der Bezeichnung des Anwendungsfalles.

### Beziehungen

Um die Zusammenhänge und Verbindungen zwischen Anwendungsfällen zu definieren, stellt die UML eine Vielzahl von Möglichkeiten bereit.

- **Include:** Diese Beziehung zwischen zwei Anwendungsfällen wird dazu verwendet, um ein Verhalten in einen anderen Anwendungsfall einzubinden. Das Resultierende Verhaltenskonstrukt wird ausgeführt, sobald das startende Verhalten angestoßen wird. Ein gerichteter Pfeil mit dem Schlüsselwort 'include' dient der Visualisierung.

- **Extend:** Eine Erweiterung des Anwendungsfalles gibt weitere Möglichkeiten im Verhalten preis. Diese können, müssen jedoch nicht verwendet werden. Die Darstellung erfolgt mit einem gerichteten Pfeil, der mit 'extend' gekennzeichnet ist.

- **Generalisierung:** Die Verwendung der Generalisierung erlaubt das Vererben von Verhaltensweisen. Die vererbten Informationen können vom erbenden Anwendungsfall übernommen, erweitert oder modifiziert werden. Als Visualisierungselement dient ein gerichteter Pfeil mit ungefüllter Spitze, wie er bereits in anderen Modellen verwendet wird.

- **Abstrakte Anwendungsfälle:** Diese Art von Anwendungsfällen dient der Strukturierung und der verbesserten Darstellung. Das Verbinden von identischen Aspekten von unterschiedlichen Anwendungsfällen kann hervorgehoben werden, sodass Gruppierungen die Übersicht erhöhen. Ein abstrakter Anwendungsfall wird mit dem Stereotypen 'abstract' versehen und mit den zu gruppierenden Anwendungsfällen über eine Generalisierungsbeziehung miteinander verbunden.

## 4.3. Aktivitätsdiagramm

Ein Aktivitätsdiagramm erlaubt die Definition von Aktivitäten, die mittels Kontrollknoten und Datenflüssen gesteuert werden. Die einzelnen Arbeitsschritte der Aktivität werden Aktionen genannt. Seit der UML Version 2.0 wurden Aktivitätsdiagramme um Konzepte von parallelen Abläufen, Tokensemantik aus Petrinetzen und weitere Verbesserungen erweitert. Viele neue Sprachelemente wurden zur Modellierung erstellt und erlaubt die verbesserte Darstellung von Komplexitäten.

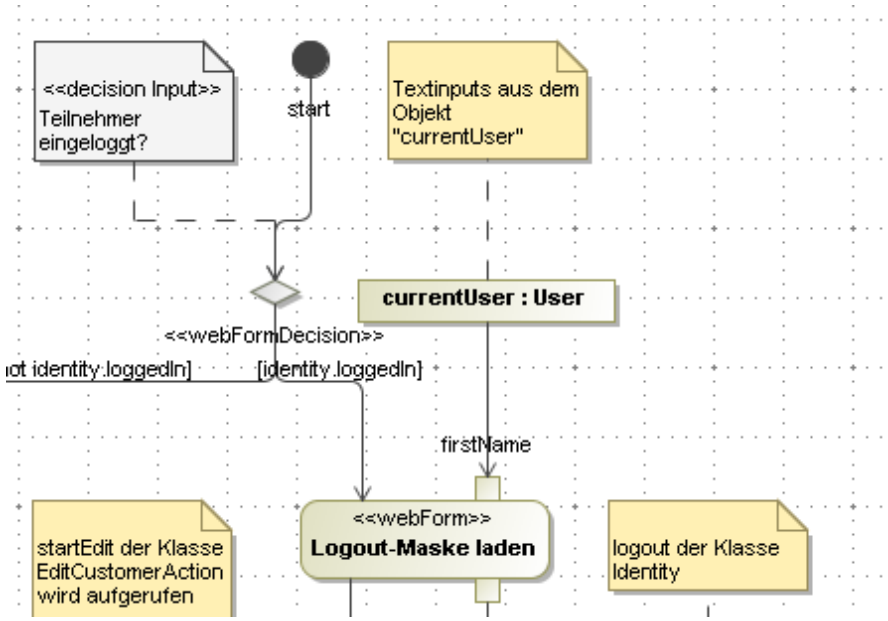


Abbildung 4.3.: Ausschnitt aus dem Aktivitätsdiagramm der Benutzeranmeldung

Auch Kommunikationsmechanismen wie Signale oder triggernde Ein- und Ausgabewerte lassen sich darstellen.

Abbildung 4.3 zeigt einen Ausschnitt aus dem modifizierten Aktivitätsdiagramm des Referenzmodells. Es zeigt einen Bereich aus dem Ablauf der Benutzeranmeldung im System.

In Anlehnung an [Mar05b] wird im Folgenden auf einige der Notationselemente der Modellierung eingegangen.

### 4.3.1. Sprachelemente

#### Aktivität

Seit UML 2.0 ist eine Aktivität nicht mehr der atomare Bestandteil der Verhaltensbeschreibung. Dieser wurde neu definiert und lässt sich durch Aktionen darstellen. Eine Aktivität kann je nach Betrachtungsweise eine Methode oder Operation oder aber auch Geschäftsprozesse darstellen. Aktivitäten verfügen über einen Namen und können verschiedenste Eigenschaften aufweisen. Desweiteren sind Vor- und Nachbedingungen und Parameter definierbar. Die Darstellung einer Aktivität erfolgt mit einem Rechteck mit gerundeten Ecken.

### Knoten

Knoten stellen den eigentlichen Inhalt einer Aktivität dar. Anhand dieses gerichteten Graphen wird die Aktivität abgearbeitet und die Verhaltensmuster und Aktionselemente durch Knoten visualisiert.

Es gibt drei verschiedenen Knotenarten, wobei jede ihre eigene Notation und Verwendungsregeln aufweist.

- **Aktionsknoten:** Als Aktion wird eine einzelne Handlung oder Ausführung im System betrachtet. Sie verrichten damit die Arbeit innerhalb der Verhaltensmodellierung. Sie empfangen Daten oder Informationen, rufen Methoden auf oder liefern Ausgaben für weitere Aktionen. Die werden als Rechteck mit abgerundeten Ecken dargestellt.

- **Kontrollknoten:** Diese Art von Knoten erlauben es, Aktionsflüsse zu regeln und bieten Konzepte für die Parallelisierung, Synchronisierung oder Entscheidungen. Auch die Start- und Endknoten der Aktivität fallen in den Bereich der Kontrollknoten. Die Notation unterscheidet sich je nach Art und Weise der Anwendung.

- **Objektknoten:** Objektknoten sind die Verbindung der Struktur und dem Verhalten. Hier lassen sich Objekte und Daten von anderen Modellen in das Verhaltensdiagramm einbeziehen. Jegliche Art von Information lassen sich über Objektknoten definieren. Die Notation erfolgt mit einem Rechteck, welches bei der Verwendung von Parametern auch direkt an den Rahmen der Aktivität geknüpft werden kann.

### Kanten

Sämtliche Knoten können mit Kanten miteinander verbunden werden. Obwohl unterschiedliche Kanten modelliert werden können, ist die Notation mittels eines Verbindungspfeils einheitlich. Die Kanten im gerichteten Graphen können über Eigenschaften oder Daten verfügen, und verbinden den Start- mit den Endknoten, wobei je nach Art der zu verbindenden Knoten Kontrollflusskanten oder Objektflusskanten zum Einsatz kommen.

### Pins

Pins erlauben die Darstellung von Daten oder Parametern direkt innerhalb der Aktionen. Durch einen Eingabepin wird eine Aktion nicht gestartet, dazu benötigt sie immer noch eine eingehende Kontrollflusskante. Zusätzlich zu Ein- und Ausgabepins gibt es weiters ein Konzept des Bufferknotens, der als Objektknoten zwischen den Pins dargestellt wird. Pins werden mittels kleiner Quadrate an Aktionen geheftet und können über kleine Pfeile verfügen, die visuell darstellen, ob es sich um einen Ein- ode Ausgabepin handelt.

## 4.4. Anpassungen

Die in Zuge der Arbeit erstellten Änderungen der UML Elemente wurden mittels UML Profile realisiert. Obwohl eine modulare Aufteilung der Elemente innerhalb des Profiles möglich wäre, wurde aufgrund der Komplexität und Menge der Anpassungen bewusst darauf verzichtet, um die Übersicht zu bewahren und die Änderungen möglichst einfach und verständlich zu halten. Abbildung 6.2 zeigt dieses Profil.

Das Profil stellt sich aus einer Ansammlung von verschiedenen Stereotypen zusammen. Datentypdefinitionen wurden nicht in das Profil übernommen, da für diese ein eigenes Konzept erarbeitet wurde.

# 5. Referenzapplikation

Eine Referenzapplikation ist nach [Tho07] ein wesentlicher Bestandteil der Generatorenentwicklung im Sinne des MDSD. Sie dient dabei als praktische Anwendung und Umsetzung der Domäne und hilft dabei, die Generatoren zu Entwickeln und zu Testen. Dabei ist der fachliche Gehalt der Referenzapplikation nebensächlich. Die manuell erstellte Referenzimplementierung wird analysiert und von der Fachlogik getrennt, welche nicht durch Generatoren erstellbaren Fragmente der Applikation enthält. Sobald der Generator mit den nötigen Transformationen implementiert ist, wird der Rahmen für die Anwendung aus dem Referenzmodell abgeleitet und um die losgelöste Fachlogik ergänzt. Die Referenzapplikation wird laut [Tho07] nach der ersten Erstellung der Templates der Generatoren in den nachfolgenden Iterationen laufend erweitert und angepasst. Sie ist aus diesem Grund nicht nur für die ersten Versionen der Templates, sondern auch für die weiteren Schritte im Projekt von zentraler Bedeutung.

In Zuge der Arbeit wird eine JBoss Seam [JBo07f] Beispielapplikation als Refererenz herangezogen. Hierbei handelt es sich um einen DVD-Store, der durch eine einfache Fachlogik und ein möglichst breites Spektrum an verwendeten Technologien als Referenz hervorsticht. Im nächsten Abschnitt wird dieser DVD-Store näher betrachtet.

## 5.1. JBoss Seam DVD-Store

Der JBoss Seam DVD-Store ist ein komplexes DVD-Verkaufssystem welches als Referenzarchitektur und Anwendungsbeispiel dient. Er ist eine einfach aufgebaute Webapplikation die innerhalb eines EJB 3.0 unterstützenden Server lauffähig ist. Der Store ermöglicht die Bestellung von DVDs über das Internet und besitzt neben der Benutzersicht einen Administrationsteil, der Einstellungen über die Verkaufsmodi ermöglicht.

Die Wahl der Referenz fiel aus folgenden Gründen auf den Seam DVD-Store:

- Der Store bedient sich einer Vielzahl von unterschiedlichen Technologien. Neben den Facelets und der Beans wird auch jBPM [JBo07d] verwendet, was einen weiteren interessanten Aspekt hinzufügt.
- Durch die guten Tutorials ist die Einarbeitungszeit in das Framework sehr gering.
- Die Einführungsbeispiele der JBoss Seam sind im Gegensatz zu vielen anderen Frameworks sofort lauffähig und der Sourcecode leicht verständlich.
- Seam findet breiten Anklang innerhalb der Sourcecode Entwickler-Community und bietet guten Support bezüglich Toolunterstützung und Hilfestellungen.

In den folgenden Abschnitten werden die Features und verwendeten Technologien näher besprochen.

## 5.2. Sicht der Benutzer

Dem Benutzer ist es möglich, innerhalb gewisser Auswahlkriterien im DVD-Store nach Begriffen zu suchen. Das dafür erstellte Suchformular ist in Abbildung 5.1 zu sehen.


Abbildung 5.1.: Die Suchfunktion des DVD-Stores

Nach der Suche können eine oder mehrerer DVDs selektiert und in einen Wa-

renkorb gelegt werden. Dieser wird nach dem ersten Eingang im Seitenbereich angezeigt und begleitet den Benutzer während des Einkaufs. Die Suchfunktion ist jederzeit möglich und bedarf keiner Anmeldung des Benutzers am System. Für den Kauf der Waren des Warenkorbs muss der Benutzer angemeldet sein, wobei der Warenkorb noch nach Bedarf angepasst werden kann. Neben der Änderung der Menge der Artikel ist ein nachträgliches Entfernen möglich. Die Kaufabwicklung erfolgt nach einer Bestätigung und bedient sich dabei bei den Bezahlungsmodalitäten, die der Benutzer bei der Registrierung angegeben hat.

Neben dem Durchsuchen und Bestellen von DVDs ist dem Benutzer auch eine Registrierung am System möglich. Dies erfolgt über die Eingabe der Benutzer- und verkaufsrelevanten Daten, welche über Formulare eingelesen werden. Hierbei werden die Account-, die Kontakt- und die Kreditkartendaten benötigt, um den Benutzer anzulegen und für Bestellungen im Store freizuschalten.

Eine Übersicht der Möglichkeiten des Benutzers sind in Abbildung 5.2 als UML Anwendungsfalldiagramm dargestellt.

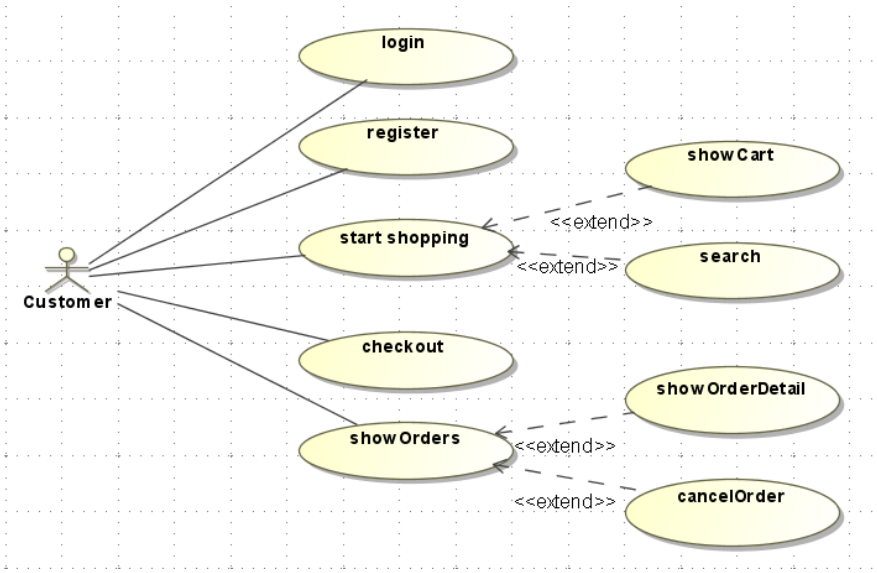


Abbildung 5.2.: Anwendungsfalldiagramm des Benutzers

## 5.3. Sicht der Administration

Die Administrationssicht des DVD-Stores ermöglicht es, die Verkaufsaktivitäten der Kunden zu beobachten und die Verkaufsprozesse zu steuern. Folgende Verkaufsmodi sind über eine Auswahlbox, welche in Abbildung 5.3 dargestellt ist, einstellbar:

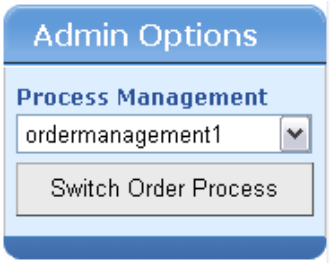


Abbildung 5.3.: Die Auswahl der Prozesse im Managementbereich

- **Order Process 1:** Alle Bestellung des Kunden werden akzeptiert. Der Manager ist lediglich für das Verschiffen des Auftrags und das Eintragen der sogenannten 'tracking number' zuständig.

- **Order Process 2:** Dieser Modus fügt dem ersten Prozess einen Bestätigungsschritt hinzu, in dem der Manager die Bestellungen ablehnen kann, bevor die Ware verschifft wird.

- **Order Process 3:** In diesem Prozess erfolgt die Entscheidung der Annahme der Bestellung über das Bestellvolumen. Aufträge unter 100$ werden automatisch vom System bestätigt und direkt wie in Prozess 2 behandelt. Bei Bestellungen jenseits dieses Betrages bedarf es einer Entscheidung des Managers, ob die Bestellung akzeptiert wird.

Die Einstellungen der Verkaufsprozesse verändert nicht nur die internen Programmroutinen, sondern erzeugen auch unterschiedliche Ausgaben bei der Kundenbestellung. Der Kunde kann über seine Bestellungsdetails den aktuellen Status seiner Bestellung betrachten. Als weiteres Feature kann der Manager die laufenden Prozesse in aufgelisteter Form betrachten. Jede Bestellung wird dabei als eigener Prozess angezeigt.

Einen Überblick der Möglichkeiten des Managers gibt Abbildung 5.4, welches die Anwendungsfälle durch ein UML Anwendungsfalldiagramm abbildet.

## 5.4. Technologischer Hintergrund

Der DVD-Store ist innerhalb der Domäne der webbasierten E-Business Verkaufssysteme angesiedelt. In den letzten Jahren haben sich viele technische Möglichkeiten entwickelt, die diese Domäne der webbasierten Anwendungen bedienen. Als die nach [Ger04b] wohl bekanntesten Vertreter der Programmiersprachen sind

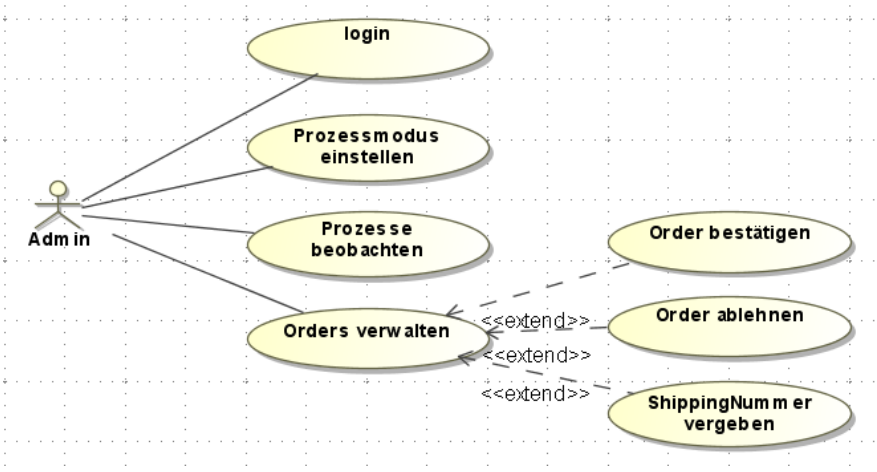


Abbildung 5.4.: Anwendungsfalldiagramm des Managers

Microstofts .NET Framework, SUN Microsystems Java Enterprise Edition, PHP, Perl und Python zu nennen, da dies die meistverwendeten Technologien in diesem Bereich sind. Neben den Programmiersprachen haben sich eine Vielzahl von Frameworks und Entwicklungshilfen für die einzelnen Technologien gebildet.

Für die Umsetzung dieser Arbeit wurde als Referenz eine auf Java EE [Sun07a] basierende Applikation herangezogen. Als Hauptgrund für diese Wahl ist die freie Verfügbarkeit der Technologien zu nennen, welche als Open Source Produkte im Internet frei erhältlich sind. Weiters sticht die Programmiersprache Java durch die hohe Portabilität und Umgebungsfreiheit hervor. Als Applikations-Framework für die Umsetzung wurde JBoss Seam [JBo07f] benutzt, welches ebenfalls frei verfügbar ist und entscheidende Vorteile gegenüber anderen Frameworks besitzt. Es folgt eine Auflistung einiger nennenswerter Vorteile:

- JBoss Seam kann in jeder J2EE Umgebung genutzt werden.
- Seam erlaubt komplexes Kontextmanagement.
- Das Framework ist frei erhältlich.
- JBoss Seam ist leicht verständlich und bietet funktionierende Tutorials an.
- Seam erlaubt die einfache Verwendung von Work- und Pageflows.
- Es bietet neben Unit Tests auch komplexere Integrationstests an, die komplette Benutzerinteraktionen testen können.
- Die Unterstützung von multiplen Browserfenstern und Tabs sowie die 'Back-

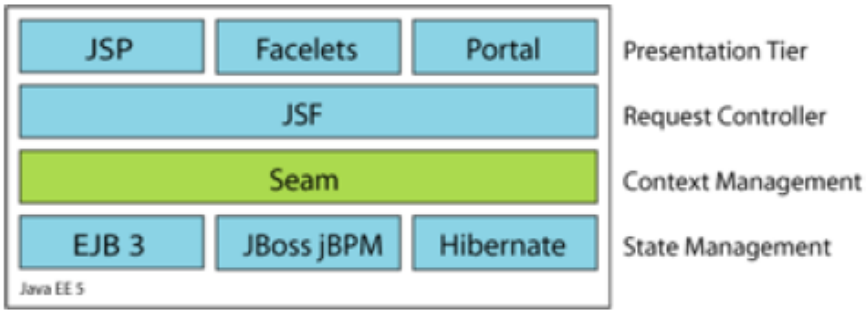


Abbildung 5.5.: Die Schichtenarchitektur von JBoss Seam

Button'-Navigation ist gegeben.

Der als Referenz betrachtete DVD-Store basiert auf JBoss Seam in der Version 1.2.1, welcher auf der JBoss Seite zu finden ist. Weiters existiert eine Live Demonstration dieses Stores auf der JBoss Seite. Im Zuge der Arbeit wird der *JBoss Applikation Server* [JBo07a] in der Version 4.05GA verwendet, in den der verwendbare Sourcecode geladen wird.

Der DVD-Store basiert auf die Verwendung der durch Seam unterstützten Technolgien, die in Abbildung 5.5 als Schichten innerhalb des Java EE Containers dargestellt sind. Die Möglichkeit der Portal-Erstellung innerhalb der Präsentationsschicht ist durch die Verwendung von JBoss Seam vorhanden, wird aber innerhalb der Arbeit nicht durchgeführt. Die Aufteilung in Schichten erhöht die Übersicht und erlaubt eine Trennung von Verantwortlichkeiten.

Im Folgenden wird auf die einzelnen in der Referenzapplikation verwendeten Technolgien näher eingegangen. Begleitend werden Teile des DVD-Stores als Beispiele angeführt und erläutert.

### 5.4.1. Java EE

Java Enterprise Edition ist ein etablierter Industriestandard im Bereich der Softwareentwicklung. Aufbauend auf den soliden Java Standard Edition bietet es nach [Sun07a] zusätzliche Möglichkeiten um service-orientierte Architekturen zu unterstützen. Die Java EE schafft eine Trennung der Applikation in einzelne Komponenten, die selbst eigenständige Softwareeinheiten darstellen. Um gemeinsam als Anwendung zu dienen, verfügen die Komponenten über Schnittstellen und bieten einander Dienste an. Innerhalb eines Applikationsservers können diese Einzelteile geladen und bedient werden, wobei dieser die Verwaltung und Steue-

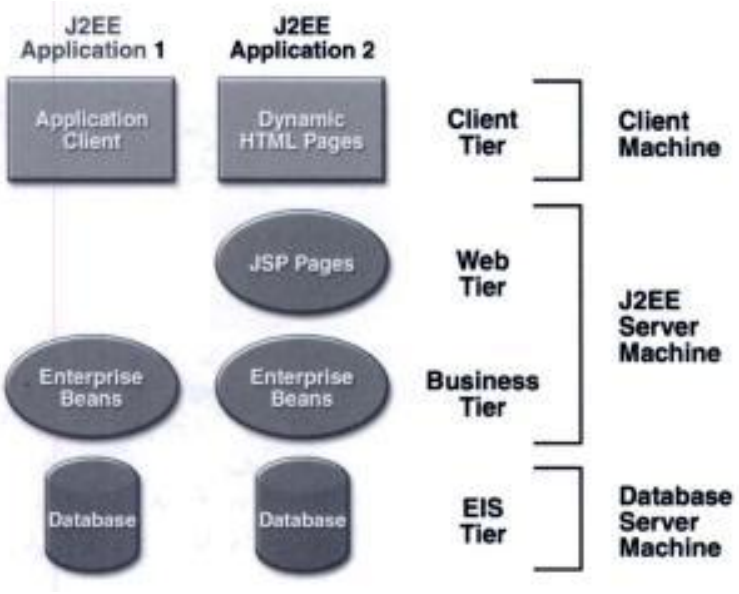


Abbildung 5.6.: Applikationen mit mehreren Schichten [Ste02]

rung übernimmt. Die Trennung der Verantwortlichkeiten werden in Abbildung 5.6 deutlich. Diese sich auf eine frühe Version der Java EE beziehende Grafik zeigt die klare Trennung der Applikation in einzelne Schichten.

Für den DVD-Store wird Java Enterprise Edition in der Version 5 verwendet, da dieser von JBoss Seam unterstützt wird. Weiters sind nach [Sun07a] mit der Verwendung der neuen Version im Gegensatz zu dessen Vorgängern Einsparungen im Bereich der Entwicklung möglich, da die benötigten Klassen und die Codezeilen reduziert wurden. Dies hilft dabei, den Code schlanker und überschaubarer zu halten und reduziert dadurch die Fehleranfälligkeit und die Entwicklungszeit.

### 5.4.2. EJB

#### Allgemeines

Enterprise Java Beans wurden von Sun Microsystems und der Java Community entwickelt und nach [Ric06] im Jahre 1998 als erste Spezifikation veröffentlicht. Nach weiteren Entwicklungen befinden sich die Enterprise Java Beans in der Version 3.0, welche im Zuge dieser Arbeit verwendet werden.

Enterprise Java Beans sind eine besondere Art von Java Beans, die für die Ausführung serverseitiger Operationen dient. Es handelt sich dabei um einen komponentenorientierten Ansatz, der die Verteilung der Aufgaben innerhalb der

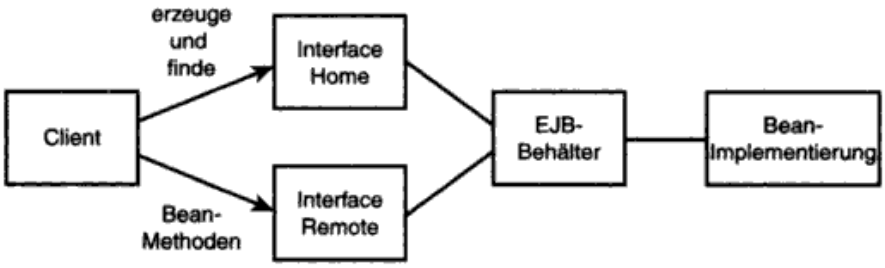


Abbildung 5.7.: Kommunikation mit EJB via Interfaces [Mar01]

Applikation unterstützt und dadurch erlaubt, diese modular und übersichtlich zu gestalten. (vgl. [Tho01], [Sun07a], [Ric06])

Die Verteilung der Komponenten erlaubt eine Aufteilung der Anwendung auf mehrere Applikationsserver, jedoch wird für die Übersicht in der Referenzapplikation nur ein Server, der JBoss Application Server, verwendet.

Ein nennenswerter Vorteil der EJBs ist es, dass der Entwickler bei vielen serverseitigen Aufgaben entlastet wird. Besonders die Bereiche der Datenanbindung an Datenbanken und das Sessionhandling wurden durch die Einführung von EJB erleichtert [Tho01].

> *EJB is a distributed component framework that provides services for transactions, specifically security and persistence in a distributed multi-tier environment.* [Jia03]

Eine Enterprise Java Bean ist eine abstrakte Klasse, die deren Funktionalitäten via Interfaces anbieten. Dies erhöht die Modularität und erleichtert die Verteilung der Objekte. Weiters sind die Informationen des Beans gekapselt und von den Client-Applikationen nur über die vordefinierten Interfaces erreichbar, was einen wesentlicher Sicherheitsvorteil darstellt und zur Robustheit des Systems beiträgt. Die Anzahl der Interfaces ist unbeschränkt wodurch für jedes erdenkliche Zugriffszenario ein eigenes Interface angeboten werden könnte. Man unterscheidet dabei zwischen Remote interfaces, Local interfaces, Endpoint interface und Message interface, die unterschiedliche Zugriffsmöglichkeiten bieten. Abbildung 5.7 zeigt den Zugriff des Clients auf die Interfaces des Beans. Wesentlich ist dabei, dass der Client zu keiner Zeit auf das Enterprise Java Bean direkt zugreift, sondern nur mit den Schnittstellen interagiert. Nähere Informationen zu den Interfaces sind in der Literatur zu finden. (vlg. [Ric06])

Neben den verschiedenen Interfaces gibt es auch eine Unterschiede in den Arten

von Enterprise Java Beans. Das JBoss Seam Framework der Referenzapplikation unterstützt dabei folgende Beans [JBo07f]:

- **EJB 3.0 Entity Beans:** Entity Beans stellen serverseitige Komponenten dar, die persistent sind und direkte Datenverarbeitung unterstützen. Sie repräsentieren jene Objekte, die sich auch in den Datenbanken wiederfinden. Als Faustregel nach [Ric06] gilt für die Erstellung dieser Entitäten, dass man Entitäten stets als Nomen innerhalb des Business-Model Konzeptes wiederfindet. Diese Art von Beans sind flexibler als Java-Klassen die das EntityBean-Interface implementieren. Sie sind der Spezifikation nach *plain old Java objects*, kurz POJOs, und können wie jedes andere POJO serialisiert, zugewiesen und durchs Netzwerk gesendet werden.

  Im Gegensatz zu anderen Enterprise Java Beans unterscheiden sich ein Entity Bean von der Art des Zugriffs. Entity Beans werden von der Applikation direkt angesprochen, benötigen daher kein Interface für die Interaktion. Den eigenen Persistenz-Status kann das Entity Bean selbst oder durch den Container managen. Nähere Informationen dazu sind in der Spezifikation [Sun07a] zu finden.

  Der Zugriff auf Entity Beans erfolgt über den *EntityManager*, der eine Query-API und Lifecycle Methoden für die Entität anbietet.

  Innerhalb JBoss Seam bieten Entity Beans keine Möglichkeit der *Bijection* oder *Kontextbegrenzung*. Der Begriff Bijection steht für *bidirectional injection* und erlaubt das dynamische Laden und Ausladen von Objekten innerhalb definierter Bereichsgrenzen. Beispielsweise lassen sich diese Objekte direk über die Benutzeroberfläche ansprechen, wobei dies durch das *Kontextmanagement* und dem definierten *Scope* ermöglicht wird. Mittels der Annotationen *@In* und *@Out* lässt sich der Mechanismus direkt in den Beans regeln.

  Üblicherweise werden die Entitäten über Session Beans angesprochen, die über den EntityManager auf das Bean zugreifen.

- **EJB 3.0 Session Beans:** Während die Entity Beans die Entitäten und Objekte innerhalb der Applikation darstellen, sind die Session Beans für die eigentliche 'Arbeit' zuständig. Sie beinhalten die Logik der Anwendung. Nach der bereits erwähnten Faustregel sind sie durch Verben im Konzept darstellbar und bilden das Verhalten ab.

> *Session beans act as agents that manage business processes or tasks for the client; they're the appropriate place for business logic. [Ric06]*

Man unterscheidet zwischen *Stateless* und *Stateful Session Beans*, wobei die Bezeichnungen bereits Aufschluss auf deren Unterscheidbarkeit gibt.

Stateless Session Beans speichern keinen Zustand der Interaktion mit dem Benutzer und sind vergleichbar mit konventionellen Methoden in Java-Klassen, die eingehende Parameter verarbeiten und Ergebnisse liefern. Dadurch dass jeder Aufruf unabhängig vom Zustand ist, ist die Leistungsfähigkeit sehr hoch und es können eine Unzahl von Clients bedient werden, ohne Probleme zu verursachen.

Stateful Session Beans speichern den Zustand des Benutzers im Speicher ab und sind in der Lage, sich Parameter und Objekte zu 'merken'. Im Gegensatz zu anderen Frameworks bietet hier JBoss Seam eine benutzerfreundliche Art der Zustandsverwaltung an. Information wird nicht über die HttpSession[1] übergeben sondern innerhalb der Session Bean festgehalten. Dies erlaubt JBoss Seam den Status der Interaktion für den Benutzer zu speichern und Kollisionen zu vermeiden. Üblicherweise werden Session Beans in Seam für die JSF Interaktion mit dem Benutzer verwendet. Zum Beispiel bei der Übergabe von ein oder mehreren Formularen an den Server.

- **Java Beans:** Java Beans unterscheiden sich von Enterprise Java Beans in ihrer Funktionsweise. Sie verhalten sich zwar wie Session Beans, bieten aber selbst keine Möglichkeiten wie Session Management, Persistenz- und Securityfeatures und weitere Funktionen an.

  Eine Einsatzmöglichkeit ist dennoch von entscheidender Bedeutung. Es ist möglich Java Beans anstelle von Session Beans zu verwenden, wenn der Applikationsserver kein EJB3 unterstützt. Zwar sind in diesem Fall Leistungseinbußen zu verzeichnen, dennoch ist diese Verwendung gerade in Bezug auf ältere Systeme von großem Vorteil.

- **EJB 3.0 Message-driven Beans:** Message-driven Beans verhalten sich sehr unterschiedlich zu anderen Bean-Arten [Ric06]. Sie bieten kein Interface für einen Zugriff sondern reagieren auf Nachrichten im System. Nachrichten

[1] HttpSession.: Möglichkeit zur Identifikation des Benutzers über mehrere Seitenaufrufe hinweg, um Informationen über den Benutzer zu speichern. [Sun07a]

von JMS, Legacy Systemen oder Webservices werden asynchron abgearbeitet. Message-driven Beans sind weder persistent, noch können sie eine Session speichern, was ihren Einsatz sehr begrenzt. Jedoch besitzen sie die Möglichkeit der Bijection innerhalb des Seam Frameworks.

Nachdem die verschiedenen Beans erläutert wurden, befasst sich der nächste Abschnitt mit der Bean-Anwendung der Referenzapplikation.

### EJBs in der Referenzapplikation

Der Seam DVD-Store basiert auf die Verwendung von EJB 3.0 Entity und Session Beans. Andere Arten von Beans kommen nicht zum Einsatz.

Der Store verfügt über neun Entitäten, wobei der die Entität 'User' abstrakt ist. Dies hat den Hintergrund, dass zwei verschiedene Arten von Benutzern im System unterschieden werden müssen, die Kunden und der Administrator. Die Entitäten verfügen über Getter- und Setter-Methoden und weisen nur eine geringe Anzahl von Anweisungen auf, die Fachlogik beinhaltet. Abbildung 6.4 zeigt einen Einblick in das UML Diagramm der Entity Beans.

Neben den Entitäten gibt es noch neun Session Beans, wobei zwei davon Stateless Session Beans sind. Diese zwei bedienen die Authentifikation und dem Aufrufen vom Informationen für den Manager. Eine Auflistung der Session Beans ist auf dem beiliegenden Modellen zu finden; einen kurzen Einblick in das Modell zeigt Abbildung 6.5. Jedes der Session Bean verfügt über ein Interface, welches den Zugriff auf dessen Funktionalität definiert.

Zuletzt verwendet der DVD-Store noch fünf Hilfsklassen, die Informationen aufbereiten oder Prozesse bedienen.

Die Bezeichnung der Beans erfolgt nach folgendem Schema:

- Entitäten werden nach dem Informationsobjekt benannt, welches durch die Klasse dargestellt wird. Zum Beispiel wird der Kunde durch die Entität Customer.java beschrieben und in der Tabelle 'CUSTOMER' persistiert.

- Session Beans sind nach der Aufgabe oder der Tätigkeit benannt. Das Bean EditCustomerAction.java verwaltet den Kunden und dessen Daten.

- Intefaces von Session Beans werden nach dem Session Bean benannt, wobei

das Wort 'Action' ausgelassen wird. Das Interface EditCustomer.java bietet somit den Zugriff auf das Bean EditCustomerAction.java.

- Hilfsklassen werden möglichst treffend nach der Hilfestellung bezeichnet. Als Beispiel ist hier die Klasse ProcessDefinitionSwitcher.java zu nennen, die bei der Veränderung der Prozessdefinitionen unerstützt.

Somit ergeben sich folgende Häufigkeiten, die hier tabellarisch dargestellt sind:

| Klasse | Anzahl |
|---|---|
| Entity Beans | 9 |
| Stateful Session Beans | 7 |
| Stateless Session Beans | 2 |
| Interfaces | 9 |
| Hilfsklassen | 5 |
| Summe | 32 |

### 5.4.3. JBoss jBPM

#### Allgemeines

JBoss jBPM [JBo07d] ist ein flexibles, auf XML-basierendes Framework zur Definition von Prozessen. jBPM steht für Java Business Process Management und stellt eine Java-Library dar, was jBPM in jeder Java-fähigen Umgebung lauffähig macht. Abbildung 5.8 zeigt, wie BPM in der Java-Library verankert ist und über Zugriffservices verfügt, die die Prozesse verwalten und ausführen lassen.

Durch jBPM innerhalb einer JBoss Seam Umgebung ist es Entwicklern möglich, komplexe Workflows und Interaktionsszenarios für den Benutzer zu generieren [JBo07f].

Zur Generierung von jBPM-Workflows gibt es einen auf Eclipse basierenden Designer, der über eine graphische Oberfläche verfügt und die Definition von Prozessen erleichtert. Neben Tasks und Nodes lassen sich komplexere Expressions anlegen.

jBPM Prozesse verfügen über einen Start- und über einen oder mehrere Endknoten, verbunden über Transaktionen und Aktivitäten. Listing 5.1 zeigt ein einfaches Beispiel eines Prozesses mit einem Start-, einem End- und einem Akti-

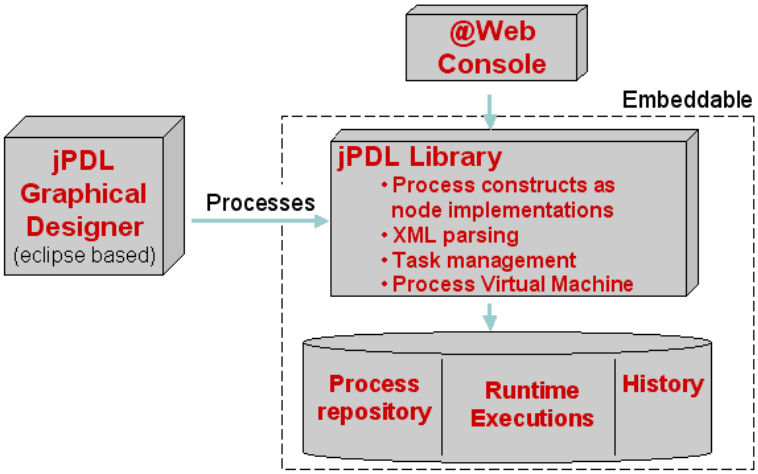


Abbildung 5.8.: Übersicht der jBPM Library [JBo07f]

vitätsknoten.

```
<process-definition name="process">
    <start-state name="start">
        <transition to="activity"/>
    </start-state>

    <task-node name="activity">
        <task name="task">
            <!-- ... Inhalt der Aktivität ... -->
        </task>
        <transition to="end"/>
    </task-node>

    <end-state name="end"/>
</process-definition>
```

Listing 5.1: Beispiel einer Prozessdefinition

Nähere Informationen zu JBoss jBPM mit Beispielen finden sich auf der entsprechenden JBoss Seite [JBo07d].

### JBoss jBPM in der Referenzapplikation

Der DVD-Store verwendet jBPM um einfache Benutzertransaktionen zu definieren. Es wurden dazu fünf Prozesse erstellt, die folgende Aufgaben erfüllen:

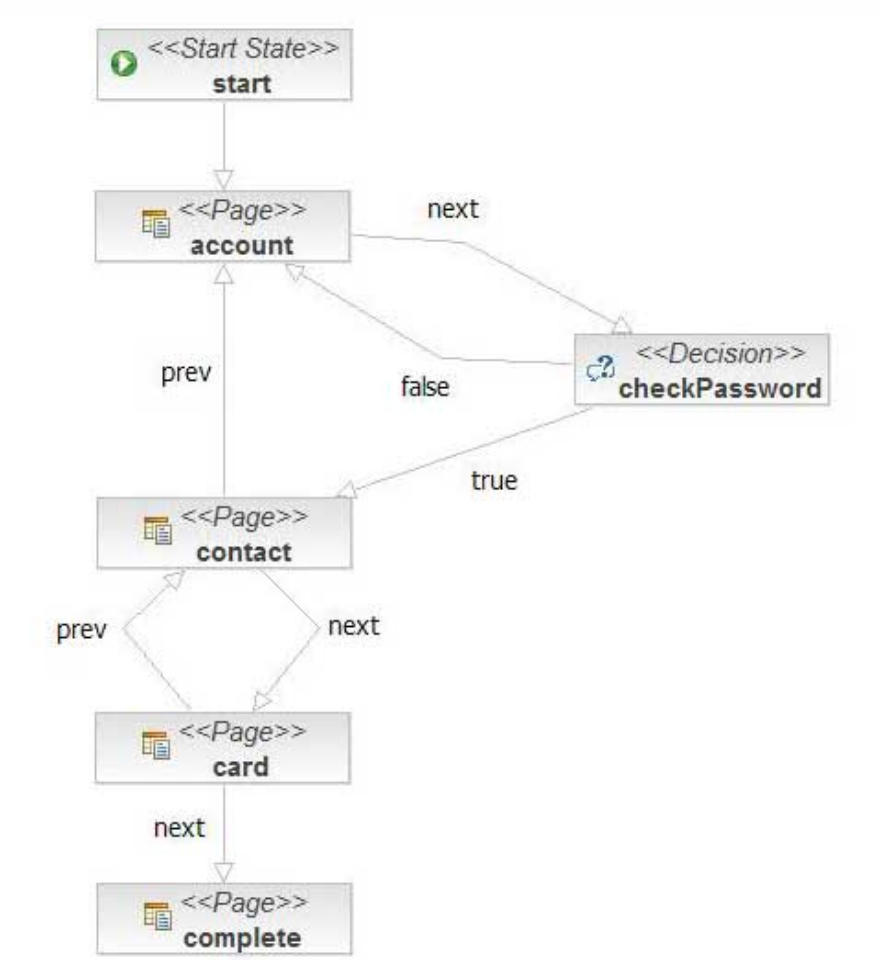


Abbildung 5.9.: Prozessdarstellung zur Registrierung eines Benutzers

- Anlegen eines neuen Benutzers
- Bestellen und Bezahlen der Waren des Warenkorbs
- 3 unterschiedliche Verkaufsmodi, die bereits in 5.3 vorgestellt wurden

Als Beispiel für einen in der Referenzapplikation verwendeten Workflow zeigt Abbildung 5.9 die Benutzerregistrierung.

### 5.4.4. Hibernate

#### Allgemeines

Hibernate ist ein Open Source Framework zur Persistierung von Objekten. Mithilfe der Open Source Community geschaffen wurde das Projekt 2003 von der JBoss Gruppe [JBo07a] übernommen und weiterentwickelt. Momentan befindet sich Hibernate in Version 3 und ist auf über 15 Datenbanken verwendbar [Mic04], wobei jede Java EE-Umgebung und EJB 3.0 unterstützt wird.

Der Grundgedanke hinter Hibernate ist es, ein Objekt als ein solches innerhalb der Datenbank zu speichern. Hierfür existieren sogenannte Mapping-Dateien, welche die Objektstruktur beinhalten und Objekte aus den persistenten Datensätzen erzeugen können. Somit lassen sich die Objekte von der Datenhaltung entkoppeln, was Vorteile in der Entwicklung und der Performance mit sich führt.

> *Hibernate's goal is to relieve the developer from 95 percent of common data persistence related programming tasks, compared to manual coding with SQL and the JDBC API.* [JBo07c]

Das Hibernate Projekt hat sich mittlerweile zu einem der gängigsten Frameworks im Bereich der Persistenzschicht entwickelt und bietet neben der Datenhaltung auch viele Möglichkeiten der Datenbearbeitung an. Als Beispiele läßt sich die Suchfunktion, Validierung und Verwaltung von Daten über den Hibernate EntityManager nennen.

Mithilfe der Hibernate Annotations können Hibernate Funktionen innerhalb der Java-Klassen verpackt und ausgeführt werden, was die Anzahl der Projektdateien reduziert und bessere Übersicht bietet.

#### Hibernate in der Referenzapplikation

Die Anwendung von Hibernate im DVD-Store zeigt sich in den Annotationen der Java Beans, den Konfigurationsfiles und den Validierungs-Annotationen innerhalb der Entity Beans, welche die eingegebenen Daten auf deren Gültigkeit überprüft.

Die Konfiguration der Persistenz der Daten des Stores zeigt Listing 5.2.

```
<persistence-unit name="dvdDatabase">
```

```
  <provider>org.hibernate.ejb.HibernatePersistence</
     provider>
  <jta-data-source>java:/dvdDatasource</jta-data-source>
  <properties>
    <property name="hibernate.hbm2ddl.auto" value="create-
       drop"/>
    <property name="hibernate.cache.use_query_cache" value
       ="true"/>
    <property name="hibernate.show_sql" value="true"/>
    <property name="hibernate.cache.provider_class"
        value="org.hibernate.cache.HashtableCacheProvider
           "/>
    <property name="jboss.entity.manager.factory.jndi.name"
        value="java:/dvdEntityManagerFactory" />
  </properties>
</persistence-unit>
```

Listing 5.2: Konfigurationsfile für die Persistenz der Applikation

Innerhalb der Hibernatekonfiguration kann man den Verweis auf das Persistenzfile erkennen. Listing 5.3 zeigt diese Verbindung als Property-Inhalt. Zu erkennen ist die Referenz der Datenquelle auf die konfigurierte Persistenzschicht, die mittels *java:/dvdDatasource* angesprochen wird.

```
<property name="connection.datasource">java:/dvdDatasource
   </property>
```

Listing 5.3: Datenquelle der Applikation innerhalb der Hibernate-Konfiguration

Neben der Persistenz wird auch die Validierung der Benutzereingaben von Hibernate übernommen. Das Paket *org.hibernate.validator* bietet diese Funktionalitäten an. Als Beispielvalidierung zeigt Listing 5.4 die Annotationen für die Passworteingabe des Benutzers. Das Passwort darf nicht 'NULL' sein und die Länge ist zwischen 6 und 50 Zeichen. Sollte die Eingabe fehlerhaft sein, wird eine Fehlermeldung eingeblendet, die durch ein Properties-File definiert werden kann. Dieses kann gegebenenfalls auf die Werte der Annotation zugreifen sodass eine Fehlermeldung wie: 'Die Eingabe muss zwischen 6 und 50 sein' möglich wird.

```
@NotNull
@Length(min=6,max=50)
```

Listing 5.4: Validierungsannotation des Passwortfeldes der Benutzerklasse

Für die Definition der persistenten Felder werden Hibernate Annotations verwendet. Listing 5.5 zeigt beispielhaft die Anwendung innerhalb der Benutzerklasse. Die @Table Annotation signalisiert die Verwendung der Tabelle 'USERS' während die @Column Annotation auf die Spalte 'USERNAME' des Tables hinweist, wobei das Feld nicht 'NULL' sein darf, eindeutig ist und die Länge des Inhaltes kleiner als 50 Zeichen sein muss. Um dies zu gewährleisten werden Überschreitungen und Fehler durch die Validierung abgefangen, die wie bereits erwähnt, genau diese Fehler abfängt.

```
@Table(name="USERS")
@Column(name="USERNAME",unique=true,nullable=false,length
   <=50)
```

Listing 5.5: Annotations für die Persistenz von Benutzerdaten

### 5.4.5. Seam Context Management

Neben den bereits erwähnten Seam Komponenten, den Enterprise Java Beans, gibt es ein weiteres Konzept innerhalb von Seam, den Kontext. Nach [JBo07f] ist jedes Bean mit einem Kontext verbunden, der über einen Namen referenzierbar ist. Durch den Zugriff über den Namen im Kontext ist eine Komposition von Komponenten innerhalb der Applikation möglich.

Das Seam Kontext Management erzeugt und verwaltet die Kontextobjekte, wobei kein Zugriff über Methoden oder Java APIs möglich ist. Lediglich die Steuerung von Kontextverhalten über Java Annotations wird bedient.

Folgende Kontexte werden innerhalb des Seam-Frameworks verwendet:

- Stateless context
- Event (or request) context
- Page context
- Conversation context
- Session context
- Business process context

- Application context

Eine genauer Erklärung zu den verschiedenen Kontextmöglichkeiten findet sich auf der JBoss Seam Seite [JBo07f].

### Seam Context in der Referenzapplikation

Der DVD-Store greift auf verschiedene Arten des Kontextmanagements zurück. Während übliche Datenbankzugriffe wie das Speichern eines Benutzers über den Session-Kontext läuft werden die Daten über den Persistenzkontext des EJB-Packages 'javax.persistence' geladen. Das Listing 5.6) zeigt Ausschnitte der Speicherung eines Kunden des Session Beans 'EditCustomer.java'.

```
import org.jboss.seam.contexts.Context;
import javax.persistence.PersistenceContext;

@Stateful
@Name("editCustomer")
public class EditCustomerAction implements EditCustomer {

    @PersistenceContext
    EntityManager em;

    @Resource
    SessionContext ctx;

    @In
    Context sessionContext;

    public String saveUser() {
            em.persist(customer); // Zugriff auf den
               Persistenz-Kontext
            sessionContext.set("currentUser", customer); //
                Setzen des Kunden im Session-Kontext
            return "success"; // Stringausgabe zur
               Bestätigung
    }
}
```

Listing 5.6: Kontextzugriff zur Speicherung

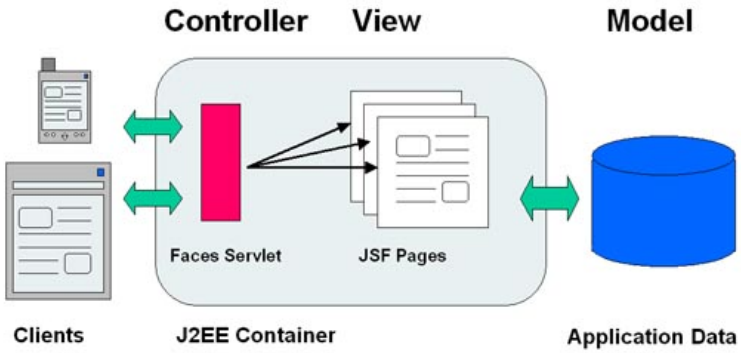


Abbildung 5.10.: Trennung der Darstellung in 3 Bereiche nach MVC

Das Beispiel (Listing 5.6) zeigt gut, wie der EntityManager durch die EJB Annotation @PersistenceContext injected wird, um direkten Zugriff zu gewähren. Neben dem EntityManager wird auch der aktuelle Session-Kontext injected. In diesen wird der aktuelle Benutzer mit der Zugriffsbezeichnung 'currentUser' gesetzt. Somit ist es möglich, den aktuellen Benutzer über diese Variable des Kontexts in jeder anderen Klasse zu verwenden.

### 5.4.6. Java Server Faces

Java Server Faces [Sun05] ist ein Framework zur Darstellung von Inhalten einer Webapplikation. Dabei muss der Entwickler sich nicht mehr direkt um die Ausgabeseite in HTML kümmern, sondern setzt sich die Ausgabe mithilfe von einzelnen, frei definierbaren Komponenten zusammen. Hierbei wird die Trennung der Darstellung nach dem MVC-Schema [Chr05] unterstützt, welches eine Trennung der grafischen Benutzeroberfläche in 3 Bereiche vorsieht. Diese sind in Abbildung 5.10 dargestellt und zeigt die Teilung der Benutzerschnittstelle in das Model und die View- und Steuerungselemente. Dadurch ist es möglich, dass die Entwickler die Oberfläche per 'drag and drop' gestalten können, ohne sich um die Fachlogik der Applikation zu kümmern. [Chr05]

Java Server Faces bringen nach [Sun05] viele Entwicklungsvorteile in folgenden Bereichen:

- Spezifikation der Seitennavigation
- Standard Komponenten für Felder, Buttons oder Links
- Validierung der Benutzereingaben

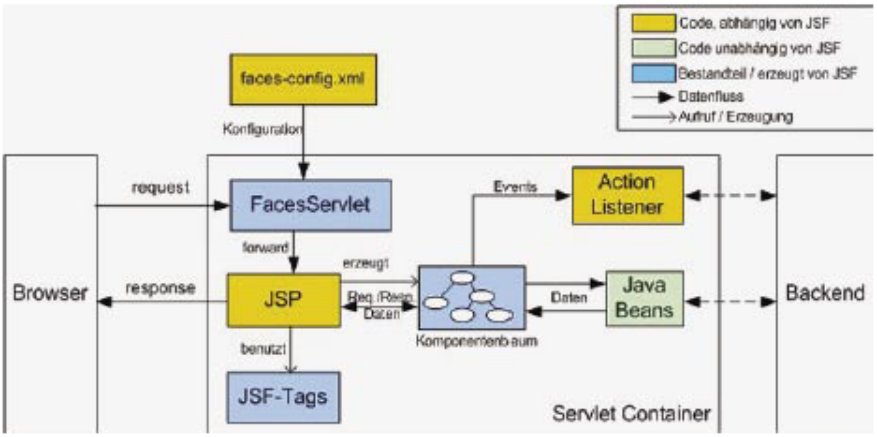


Abbildung 5.11.: Grober Aufbau über eine JSF-Anwendung [Ger04a]

- Event- und Fehlerbehandlung
- Java Bean Management
- Unterstützung von Internationalisierungsmaßnahmen

Abbildung 5.11 zeigt die Funktionsweise einer JSF-Anwendung. Der Browser greift auf die Applikation zu und erzeugt einen Request, welcher vom JSF Faces-Servlet entgegengenommen und an die Darstellungsseite weitergeleitet wird. Im Beispiel ist dies eine JSP-Seite, die mithilfe der angebundenen Library mit dem Aufbau des Komponentenbaumes beginnt und sich die dafür benötigten Daten aus den Beans zu holen. Die Bean greifen dabei über Schnittstellen auf die Daten zu, wobei die dazu verwendete Technologie für die JSF-Komponenten irrelevant ist. Neben der Darstellung von Daten aus den Beans werden auch gegebenenfalls ActionListener an die grafische Benutzeroberfläche gebunden, die auf Benutzeraktionen wie dem Pressen von Buttons reagieren. Sobald die JSP-Seite erzeugt und mit den Backend-Inhalten verknüpft ist, wird sie als Response des Servers an den Client des Benutzers zurückgesendet.

### JSF in der Referenzapplikation

Die Referenzapplikation verwendet JSF für die Kapselung von Benutzerinhalten und trennt damit die Oberfläche von der Fachlogik. Zu finden sind JSF Elementen in den Konfigurationsfiles und den xHTML-Dateien, wobei ersteres durch die Verwendung von JBoss Seam und Faceletts lediglich die View- und Seam-Handler enthält.

Als Beispiel für ein grafisches JSF-Element zeigt Listing 5.7 ein Formular mit einer Ausgabe und einem Button. Der Sourcecode hierfür wurde aus der Datei 'login.xhtml' entnommen und zur besseren Darstellung gekürzt. Der Namespace für die JSF-Komponenten wird innerhalb des 'div-Tags' definiert.

```
<div xmlns="http://www.w3.org/1999/xhtml" xmlns:s="http://
   jboss.com/products/seam/taglib" xmlns:ui="http://java.
   sun.com/jsf/facelets" xmlns:f="http://java.sun.com/jsf/
   core" xmlns:h="http://java.sun.com/jsf/html" class="menu
   ">
    <h:form rendered="#{identity.loggedIn}">
        <h:outputText value="#{currentUser.firstName}"/>
        <h:commandButton action="#{identity.logout}" value
           ="ausloggen"/>
    </h:form>
</div>
```

Listing 5.7: Gekürzter Ausschnitt aus der login.xhtml

Da JBoss Seam viele vordefinierte Elemente zur direkten Einbindung bietet, stützt sich die Referenzapplikation sich nur im geringen Maß auf die direkte Verwendung von JSF-Elementen. Webinhalte werden vielmehr über sogenannte Facelets eingebunden, welche in Abschnitt 5.4.8 behandelt werden.

Nähere Informationen zur Verwendung von JSF innerhalb der JBoss Seam Umgebung finden sie auf der JBoss Seam Homepage [JBo07f].

### 5.4.7. Java Server Pages

Java Server Pages wurden von Sun Microsystems entwickelt und erlauben die Einbettung von dynamischen Java-Inhalten in statische Webdatein. Dies geschieht über eine Vielzahl von vordefinierten XML-Tags, die in den Anzeigedatein verwendet werden und Zugriff auf serverseitige Informationen, wie Beans und Kontextvariablen haben. Listing 5.8 zeigt einen kurzen Einblick in die Syntax von JSP.

```
<html>
<head><title>First Example</title></head>
<body>
    <h1>Benutzerinformationen</h1>
    Ihre IP-Adresse ist: <%= request.getRemoteAddr() %>
```

```
</body>
</html>
```

Listing 5.8: Syntax der Java Server Pages anhand eines Beispiels

Aufgrund der Tatsache, dass der DVD-Store JSF und Faceletts für die Darstellung verwendet, finden sich nahezu keinerlei JSP-Informationen in den Webseiten. Lediglich in den 'index.jsp'-Datein befindet sich JSP-Inhalt welcher dazu dient, die Home-Seam Seite laden (siehe Listing 5.9).

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="
   c" %>
<c:redirect url="/home.seam" />
```

Listing 5.9: Umleitung auf die Home-Seite durch JSP

Durch die geringe Verwendung von JSP-Inhalten innerhalb des DVD-Stores wird an dieser Stelle nicht näher auf Java Server Pages eingegangen. Weiterführende Informationen finden sich auf [JSP07] und [Sun07b].

### 5.4.8. Facelets

Im letzten Abschnitt wurde die Verwendung der Java Server Pages grob erklärt. Ein entscheidender Nachteil der JSP wurde jedoch noch nicht genannt: das Rendering der Darstellung. Es kommt bei der Erzeugung der Oberfläche durch den Komponentenbaum zu Ungereimtheiten in der Komponentenerzeugung, was sich dadurch auswirkt, dass Elemente nicht auf Elemente zugreifen können, die noch erzeugt werden müssen. Zwar existieren Möglichkeiten, dieses Problem zu umgehen, dennoch kann es zu Anzeigeschwierigkeiten kommen sobald HTML Code neben JSP-Inhalten verwendet wird.

Zur Lösung dieses Problems wurde die Facelets-Technologie [Sun07c] entwickelt. Während JSP mithilfe der Tag-Library den Komponentenbaum und als Nebenprodukt die Komponenten erzeugt, trennt Facelets diese Schritte und generiert zuerst den kompletten Komponentenbaum. Dies Geschieht über ein XML-Template. Durch diese Auslagerung des Baumes in eine XML-Datei wird die Trennung noch deutlicher. Listing 5.10 zeigt das auf die Facelet-Aspekte reduzierte Template der Referenzapplikation. Das Beispiel zeigt die Referenz der Navigationelemente mithilfe des 'insert'-Tags.

Ein wesentliches Kriterium für die Verwendung von Facelets ist das Wegfallen des Anpassens der Taglib-Library, da Facelets direkt auf die JSF-Komponenten zugreifen kann. Hierfür werden eine Vielzahl von vordefinierten Befehlen angeboten, welche an dieser Stelle den Rahmen sprengen würde. Weiterführende Informationen zu Facelets finden sich auf der Projektseite [Sun07c].

### Facelets der Referenzapplikation

Der DVD-Store bedient sich einer Vielzahl von Facelets-Elementen. Als Beispiel hierfür wird das Einbinden der Navigation herangezogen, das in den Listings 5.10 und 5.11 gezeigt wird.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<head>
    <meta http-equiv="Content-Type" content="text/html;
       charset=iso-8859-1" />
    <title>Facelets Beispiel</title>
</head>

<body>
    <ui:insert name="navigation">
        Einfügen der Navigationselemente
    </ui:insert>
</body>
```

Listing 5.10: Teil des Facelet Templates des DVD-Stores

Die Einbindung der Navigation auf einer Seite des DVD-Stores ist in Listing 5.11 dargestellt. Es zeigt wie das Template-File definiert und danach das Facelets-Element mit dem Namen 'navigation' eingefügt wird.

```
<body>
<ui:composition template="/template.xhtml">

    <ui:define name="navigation">
        <ui:include src="/navigation.xhtml" />
    </ui:define>
```

```
</body>
```

Listing 5.11: Einbindung der Navigation

Das JBoss Seam Framework bietet auch in der Verwendung von Facelets weitere Verbesserungen, indem es das direkte Einbinden von Parametern in Facelet-Methodenaufrufen erlaubt. Hierfür muss jedoch ein spezieller View-Handler, der SeamFaceletViewHandler, in das JSF-Konfigurationsfile eingebunden werden. Die Referenzapplikation bedient sich dieser Methode nicht, da die dafür möglichen Bereiche durch jBPM abgebildet werden, wodurch diese Art von Aufrufen innerhalb der Benutzeroberfläche wegfällt.

# 6. Der Backend-Generator

Dieses Kapitel widmet sich dem Generator des Backend-Bereichs. Als Backend bezeichnet man jene systemnahen Komponenten einer Applikation, die dem Benutzer durch die Interaktion mit der Benutzeroberfläche verborgen bleibt. Innerhalb dieses Systembereichs erfolgen beispielsweise die Datenbankzugriffe, das Session-Handling und die Workflow-Abläufe.

Zuerst wird die Architektur der Anwendung näher beschrieben. Dies ist wichtig damit redundante Elemente gefunden und innerhalb des Generators wiederverwendet werden können. Weiters zeigt sie die Funktionsweise der Applikation und wie Zugriffe auf das System erfolgen.

Als nächsten Schritt werden die Planung der Generatorentwicklung und das gewünschte Ergebnis behandelt. Hierbei werden die einzelnen Bereiche der Applikation unterteilt und nach deren Automatisierbarkeit gegliedert.

Der darauf folgende Abschnitt widmet sich der Modellierung und den Profilen für die Erweiterung des Metamodells. Zuguterletzt wird der Generator selbst erklärt und das Ergebnis analysiert, wobei mögliche Verbesserungspotentiale das Ende des Kapitels darstellen.

## 6.1. Systemarchitektur

Wie bereits in Kapitel 5 gezeigt wurde, baut die Applikation auf dem JBoss Seam-Framework auf und bedient sich dabei den in Abbildung 5.5 gezeigten Technologien. Dabei stützt sich das Backend auf die Verwendung der Enterprise Java Bean- und der JBoss jBPM-Files. Weiters gibt es noch eine Vielzahl von Konfigurationsdateien, die die Abläufe steuern und Framework-spezifische Informationen beinhalten.

Das Java-Backend besteht aus 32 Java-Klassen, die die benötigten Objekte

(beispielsweise Produkte, Benutzer, Bestellungen uvm.) darstellen und Bearbeitungsmöglichkeiten für diese bereitstellen. Die genaue Aufteilung der Beans in deren Funktion wurden bereits im Kapitel 5 gezeigt.

Neben den Java-Files werden noch 5 jBPM-Files eingesetzt, deren Funktionalität bereits in Kapitel 5 erwähnt wurden. Die Erstellung dieser Prozessdatein ist mithilfe des JBoss Process Designers [JBo07e] möglich, der sich voll in die Eclipse-Umgebung integrieren lässt und die Modellierung zulässt.

Die meisten der Konfigurationsfiles sind Standarddatein der verschiedenen Framework-Distributionen, die während der Installation automatisch erstellt werden. Folgende Konfigurationsdatein werden in der Referenzapplikation verwendet:

- **components.properties:** Dieses File beinhaltet Informationen über die verschiedenen Komponenten der Applikation.

- **jbpm.cfg.xml:** In dieser Datei sind Informationen über den jBPM-Services gespeichert.

- **hibernate.cfg.xml:** Diese Datei beinhaltet die Definitionen und allgemeine Einstellungen für die Verwendung von JBoss Hibernate.

- **dvd-ds.xml:** Hier wird die Schnittstelle zur Datenbank definiert.

- **build.xml:** Die Build.xml Datei stellt Informationen über das Deployment und allgemeine Informationen über die Applikation bereit

Neben den einzelnen Datein werden noch zwei Konfigurationsordner benötigt, die Konfigurationen über den Meta- und Web-Bereich der Anwendung behinhalten.

- **META-INF-Ordner:** Die fünf Konfigurationsfiles dieses Ordners beinhalten Metainformationen über die Applikation und definieren Mappings zu Java-Bibliotheken.

- **WEB-INF-Ordner:** Dieser Ordner definiert über seine fünf XML-Dateien die Web-Informationen der Applikation. Neben der Navigation befinden sich auch Einstellungen über JSF oder Page-Mappings in den Dateien.

Der Benutzer der Anwendung greift über die Benutzeroberfläche auf die Applikation zu. Diese wird mittels Facelets und JSF-Objekte, die in die JSP-Seiten eingebunden sind, dargestellt und kann über den Seam-Kontext oder über die Zu-

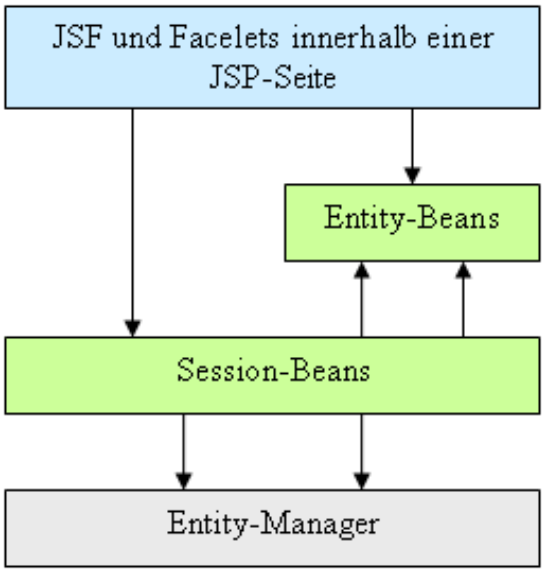


Abbildung 6.1.: Zugriffsschema der Applikation

griffsbeans die dahinter liegenden Entitäten und Methoden benutzen. Abbildung 6.1 zeigt diesen Zugriff der JSP-Seiten auf die Backend-Informationen.

Nachdem die einzelnen Backend-Elemente behandelt wurden, ist es für die Entwicklung des Generators notwendig, dessen Implementierung zu planen und Ziele für die Transformationen zu abzuleiten.

## 6.2. Planung und Zielsetzung der Entwicklung

Für die Erstellung des Generators ist es von entscheidender Bedeutung, eine bestehende Referenzapplikation hinsichtlich ihrer Eigenschaften zu analysieren. Hierbei werden nicht nur wiederkehrende Bereiche innerhalb des Source-Codes, sondern auch Zugriffskonzepte und architekturelle Aspekte gesucht, die sich mithilfe des Generators automatisieren lassen.

Die Entwicklung des Generators orientiert sich an der bereits in Kapitel 2 vorgestellten zweigleisigen, iterativen Vorgehensweise. Eine Vielzahl von Iterationen ist nötig, um den Generator auf einen akzeptablen Automatisationsgrad zu bringen. Nach jeder der Iteration wird eine kurze Testphase eingeleitet, die den Fortschritt der Entwicklung mit der Referenz vergleicht und künftige Änderungen am Generator aufzeigt.

Neben der Analyse der Referenzimplementierung ist es ebenfalls notwendig,

sich Gedanken über die Modellierung des Referenzmodells zu machen. Hierbei entstehen die benötigten Sprachelemente, die für die Entwicklung der Generatoren notwendig sind.

### 6.2.1. Referenzmodell

Das Referenzmodell ist nach [Tho07] ein wichtiges Instrument bei der Erstellung des Generators. Es stellt eine Instanz der DSL dar und zeigt, wie man die Sprachelemente dieser beispielhaft anwendet. Bei der Modellierung der Referenz ist darauf zu achten, dass eine Implementierung mit dem Generator möglich und die Syntax und die Semantik der DSL eingehalten wird.

Für die Modellierung des Referenzmodells ist es sinnvoll, Informationen aus der Analyse der Referenzapplikation heranzuziehen. Danach werden Klassenmodelle erstellt, die sowohl der DSL als auch den Requirements der Applikation entsprechen. Notwendige Anpassung der UML-Sprachelemente werden innerhalb des Profils definiert.

### 6.2.2. Analyse der Referenzimplementierung

Die Analyse der Referenzimplementierung dient in erster Linie der Überprüfung von Anforderungen an die Referenzapplikation. Bei einer zu komplexen und nicht modular aufgebauten Applikation wäre die Automatisierbarkeit nicht in einem akzeptablen Rahmen gewesen. Beispielsweise wäre eine Applikation mit nur zwei völlig überladenen Klassen für die Generatorentwicklung nicht praktikabel.

Durch die Verwendung von EJB innerhalb des JBoss Seam-DVD Stores ist eine gute Basis für Automatisierung gegeben. Sie eignen sich durch ihre klare Struktur und ihren gleichmäßigen Aufbau besonders gut für die Verwendung von Templates und lassen sich mit einfachen Mitteln modellieren.

Das Ergebnis der Analyse ist eine konkrete Vorgehensweise in der Entwicklung der Templates und des Generators:

- **Java Beans:** Für die unterschiedlichen Arten von Beans werden Stereotypen erzeugt. Dies hat den Vorteil der einfachen Trennung der Beans und erlaubt eine rasche Implementierung der Unterschiede. Weiters können die Getter- und Setter-Methoden der Entitäten automatisch erzeugt werden

und müssen nicht extra in den Modellen aufgeführt sein. Dies soll Entwicklungszeit sparen und die Modelle übersichtlicher halten.

- **Abhängigkeiten:** Die Zusammenarbeit der einzelnen Beans findet über die manuell zu implementierenden Bereiche der Applikation statt. Weiters werden die Entitäten nicht direkt in den Session Beans verwaltet, sondern über den Entitymanager geladen. Somit ist die Modellierung der Zusammengehörigkeiten der verschiedenen Java Beans im Modell überflüssig.

- **Konfigurationsfiles:** Die meisten Informationen der Datein sind manuell zu erstellen. Desweiteren lassen sich diese Informationen nicht in Modelle bringen oder müssten zwanghaft eingefügt werden, was das Verständnis und die Übersicht gefährden würde. Somit wurde entschlossen, diese Dateien nicht mit dem Generator zu erstellen, sondern diese den Installationen der unterschiedlichen Frameworks zu entnehmen und je nach Bedarf anzupassen.

Die Implementierung ist sehr einfach mit EJB aufgebaut und die Session Beans sind übersichtlich und einfach gehalten. Die Benennung der Beans ist lässt auf die Funktionalität schließen und jedes Session Bean verfügt über ein Interface, das sämtliche Public-Methoden zugreifen lässt. Somit lässt sich dieses Konzept auch einfach in den Generator einbauen.

### 6.2.3. Automatisationsgrad

Neben der Planung der Entwicklung ist es ebenfalls nötig, Ziele und Messgrößen und Messungen zu definieren. Dabei wurde bei der Entwicklung des Backend-Generators ein Automatisationsgrad von 75 Prozent angestrebt. Dieser Wert ist nach [Tho07] in einem durchaus üblichen Bereich der modellgetriebenen Entwicklung in objektorientierten Sprachen.

Die Messungen des Automatisationsgrades werden innerhalb dieser Arbeit mithilfe der Anzahl der Codezeilen möglich. Dazu werden die automatisch generierten Zeilen den manuell implementierten Zeilen gegenübergestellt, und deren prozentualen Anteil an der gesamten Applikation berechnet.

Der gesamte Entwicklungsaufwand wird zur Beurteilung der Entwicklung für Messungen nicht herangezogen. Der Wert des Entwicklungsaufwands ist aufgrund der schlechten Mess- und Bewertbarkeit der Generatorentwicklung ohnehin sehr fragwürdig und beruht auf Aufwandsschätzungen der Transformationen ge-

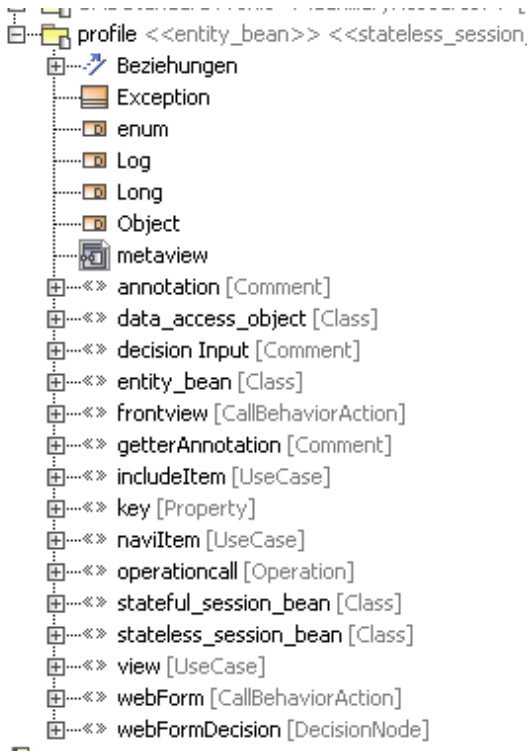


Abbildung 6.2.: Erzeugtes UML 2.0 Profile

genüber der Referenzapplikation. Auf dessen Ermittlung wurde aus Gründen der Objektivität bewusst verzichtet. Sie stellen somit kein qualitatives Merkmal der Generatorentwicklung dar.

## 6.3. UML 2.0 Profil

Dieser Abschnitt beschäftigt sich mit dem erstellten UML Profil. Dieses erweitert das UML 2.0 Metamodell und bietet die Möglichkeit, auf einfache Art und Weise auf die verschiedenen Elemente der Diagramme im Generator zuzugreifen. In Abbildung 6.2 sind die Elemente des Profiles aufgelistet. Im Folgenden werden die Elemente des Backend-Generators und deren Verwendung erläutert.

<<**entity_bean**>> Dieses Stereotyp lässt sich auf jedes UML::Class Objekt anwenden und dient zur Kennzeichnung der Entitäten in den Klassendiagrammen. Diese repräsentieren die Objekte der Datenhaltungsschicht und wurden bereits in Kapitel 5 näher besprochen.

Entity Beans können mit anderen Beans in Verbindung stehen, jedoch wird durch den Generator lediglich die Generalisierungsbeziehung miteinbezo-

gen. Auch abstrakte Klassen lassen sich durch die Transformationen erzeugen.

Zusätzlich zu den Eigenschaften wird durch den Generator für jedes nicht finale Attribut eine Getter- und Setter-Methode erstellt. Dies reduziert den Modellierungsaufwand und erlaubt es, auf die Visualisierung dieser Methoden zu verzichten.

<<**stateful_session_bean**>> **und** <<**stateless_session_bean**>> Elemente mit diesem Stereotyp sind Session Beans, welche bereits im Kapitel 5 vorgestellt wurden. Sie greifen innerhalb des JBoss Seam Kontexts auf die Entitäten zu.

Diese Beans werden im Generator unterschiedlich behandelt und weisen im Gegensatz zu den Entitäten Interfaces auf, die die Zugriffsmethoden anbieten. Interfaces die an eine Session Bean gekoppelt sind, bieten der Einfachheit halber sämtliche Public-Methoden des Beans zur Verwendung an. Sollte kein Interface über das Modell erstellt werden, wird dieses der Namensgebung (siehe Kapitel 5) entsprechend erzeugt.

<<**key**>> Dieses Stereotyp wurder der Vollständigkeit halber angelegt, um das Key-Attribut einer Entität zu bestimmen. Alternativ lässt sich dieses auch über eine Annotation im Modell definieren. Innerhalb des Generators wurde dieses Stereotyp in diesem Referenzmodell nicht behandelt, lässt sich jedoch für Erweiterungen einsetzen.

<<**data_access_object**>> Mit diesem Stereotypen gekennzeichnete Objekte sind Hilfsklassen in Java, die den Zugriff auf Entitäten, Session Beans oder Elementen der JBoss jBPM haben. Diese Klassen sind über den Stereotypen von den anderen Komponenten getrennt, da diese über kein Interface und über keine Getter- und Setter-Methoden verfügen.

<<**annotations**>> Dieses Stereotyp leitet sich von der Klasse UML::Comment ab und erlaubt die Definition von Java Annotationen. Diese werden durch den Generator an die erforderliche Stelle angefügt. Ein Beispiel hierfür sind Hibernate-Validierungs-Annotationen, die die Eingaben des Benutzers überprüfen und gegebenenfalls Fehlermeldungen ausgeben.

<<**getter_annotation**>> Zusätzlich zu den bereits gezeigten Annotationen wurde dieses Stereotyp definiert. Dieser lässt Annotationen an Positionen de-

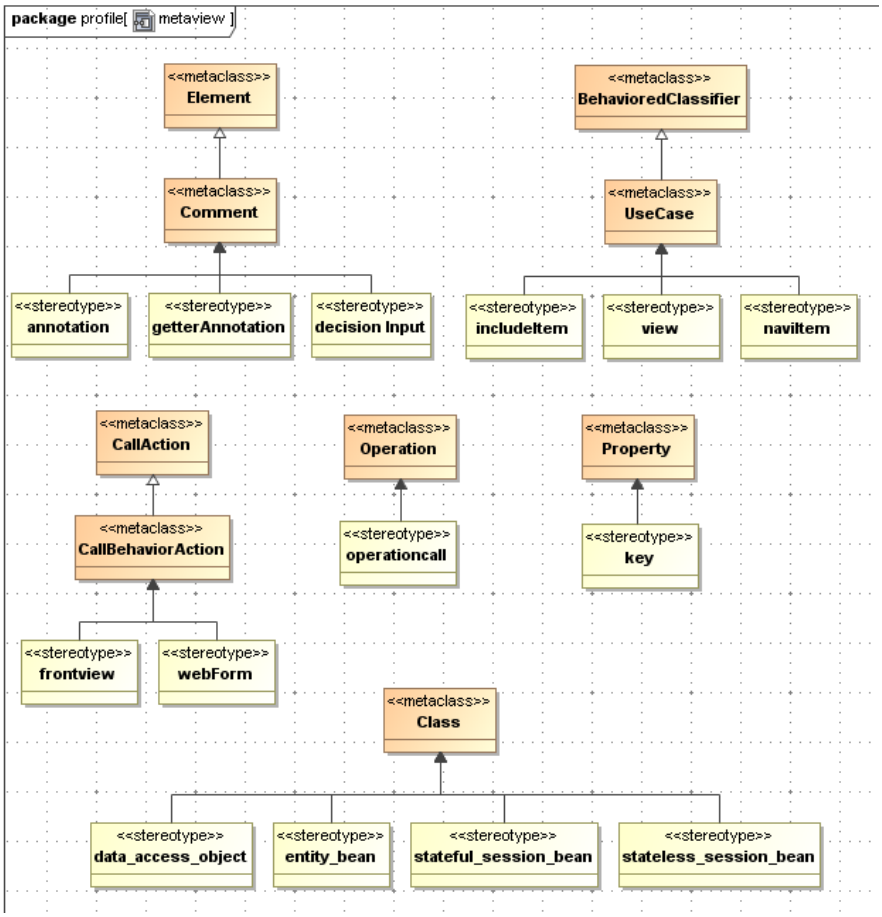


Abbildung 6.3.: Metadarstellung der Stereotypen des Profiles

finieren, die im vereinfachten Klassenmodell nicht darstellbar sind, da Getter- und Setter-Methoden aufgrund der Übersicht nicht modelliert werden müssen. Annotationen mit diesem Stereotypen werden nicht an der definierten Stelle, sondern bei der Getter-Methode des Attributes angefügt.

**Hinzugefügte Basic Datatypes** Neben Stereotypen werden einfache Datentypen verwendet, die dem Profile hinzugefügt wurden. Diese lassen sich dadurch in den Modellen verwenden. Wichtig ist jedoch, dass diese Datentypen vom Generator mit Java-Imports versehen werden, die die entsprechende Bibliothek angeben.

Das UML-Profil wurde mithilfe des Modellierungswerkzeugs angelegt und in eine XML-Datei exportiert. Abbildung 6.3 zeigt die Meta-Klassen und die abgeleiteten Stereotypen innerhalb des Profils, die für den Front- und Backend-Bereich definiert wurden.

## 6.4. Modellierung

Zur Modellierung der Diagramme wird MagicDraw [Mag07] in der Version 12.5 verwendet, da dieses neben dem Exportieren im XML Format auch das Importieren von XML-Modellen erlaubt. Der Export der Modelle in das XML Format ist für das Einlesen der Modelle in den Generator wichtig, als Alternative bietet openArchitectureWare jedoch auch das direkte einlesen der MagicDraw-Datein über einen speziellen Input-Reader. Die Modelle werden im Zuge der Arbeit jedoch als XML-Files exportiert, da diese dadurch flexibler anzuwenden sind und dadurch auch für andere MDSD-Generatoren-Frameworks anwendbar wären. Für die Modellierung der Aspekte des Backend-Generators werden folgende Diagramme erstellt:

- **Diagramm 'Entity':** Dieses Diagramm bildet die Entitäten der Applikation ab und realisiert die Schnittstelle zur Serialisierung.

- **Diagramm 'Beans':** Hier sind die Session Beans zu finden, die mithilfe des des Stereotypen unterschieden werden. Weiters werden die Interfaces abgebildet, die die Public-Methoden des Beans anbieten.

- **Diagramm 'others':** Dieses Diagramm beinhaltet die Hilfsklassen für das Prozessmanagement und andere Java-Klassen, die der Benutzer nicht direkt anspricht und somit keine Interface-Realisierung benötigen.

Wie bereits erwähnt sind die Entitäten und Hilfsklassen in drei unterschiedliche Diagramme aufgeteilt, um die Übersicht über das Backend zu erhöhen. Abbildung 6.4 zeigt einen Ausschnitt aus dem Klassendiagramm der Entitäten. Neben den Entitäten *OrderLines* und der *Produkte* zeigt es auch die Generalisierungsbeziehung der Benutzer an, die sich in Käufer und den Administrator aufteilen. Aus diesem Grund ist auch der *User* als abstrakte Klasse dargestellt. In Abbildung 6.5 sieht man einen Ausschnitt aus dem Diagramm der Session Beans. Hierbei werden die bereits erwähnten Stereotypen auf die Klassen angewandt und mit Interfaces verbunden. Diese sind vereinfacht modelliert und erben somit sämtliche Public-Methoden der Oberklasse. Gut zu erkennen ist die Verbindung mit der abstrakten Interface-Klasse *Serializable*, die es der Klasse erlaubt, serialisiert werden zu können und die entsprechenden Sourcecode-Änderungen automatisch vornimmt.

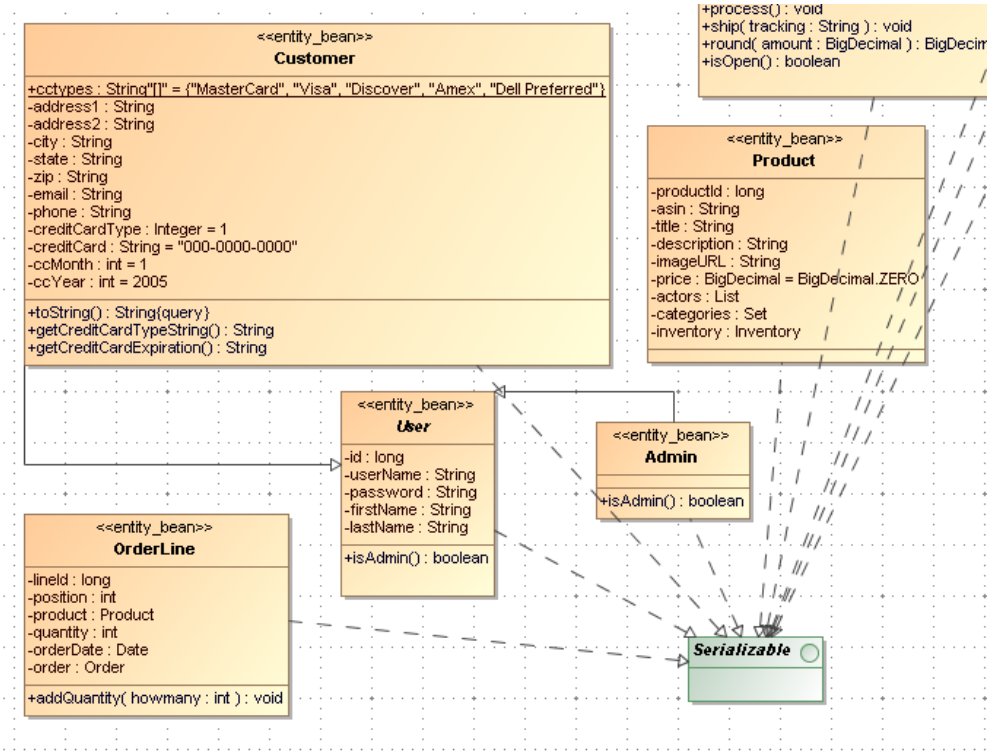


Abbildung 6.4.: Ausschnitt aus dem Entitäten-Klassenmodell

Die Modellierungsoberfläche von MagicDraw erlaubt nach der Erstellung und Visualisierung der einzelnen Klassen auch eine Bearbeitung der Eigenschaften. Abbildung 6.6 zeigt die Möglichkeiten der Anpassung der Klasse *User*. Neben der Definition von abstrakten Klassen und Schnittstellen lassen sich auch in dieser Benutzermaske Kommentare oder Annotationen einfügen. Gut zu erkennen ist beispielsweise die realisierte Schnittstelle auf die Klasse *Serializable*.

Auf eine vollständige Darstellung der Klassendiagramme wurde an dieser Stelle bewusst verzichtet, da diese aufgrund der Größe und Komplexität leicht unübersichtlich werden würden. Die Diagramme sind jedoch im Anhand der Arbeit auf dem beigefügtem Eclipse-Workspace zu finden und lassen sich mit Eclipse öffnen und bearbeiten. Durch die Verwendung von XML lassen sie sich auch in gängigen Modellierungswerkzeugen anzeigen

## 6.5. Beschreibung und Ablauf des Generators

Dieser Abschnitt befasst sich mit den Details und dem Ablauf des Generators und zeigt die wichtigsten Aspekte des Codes und der Templates.

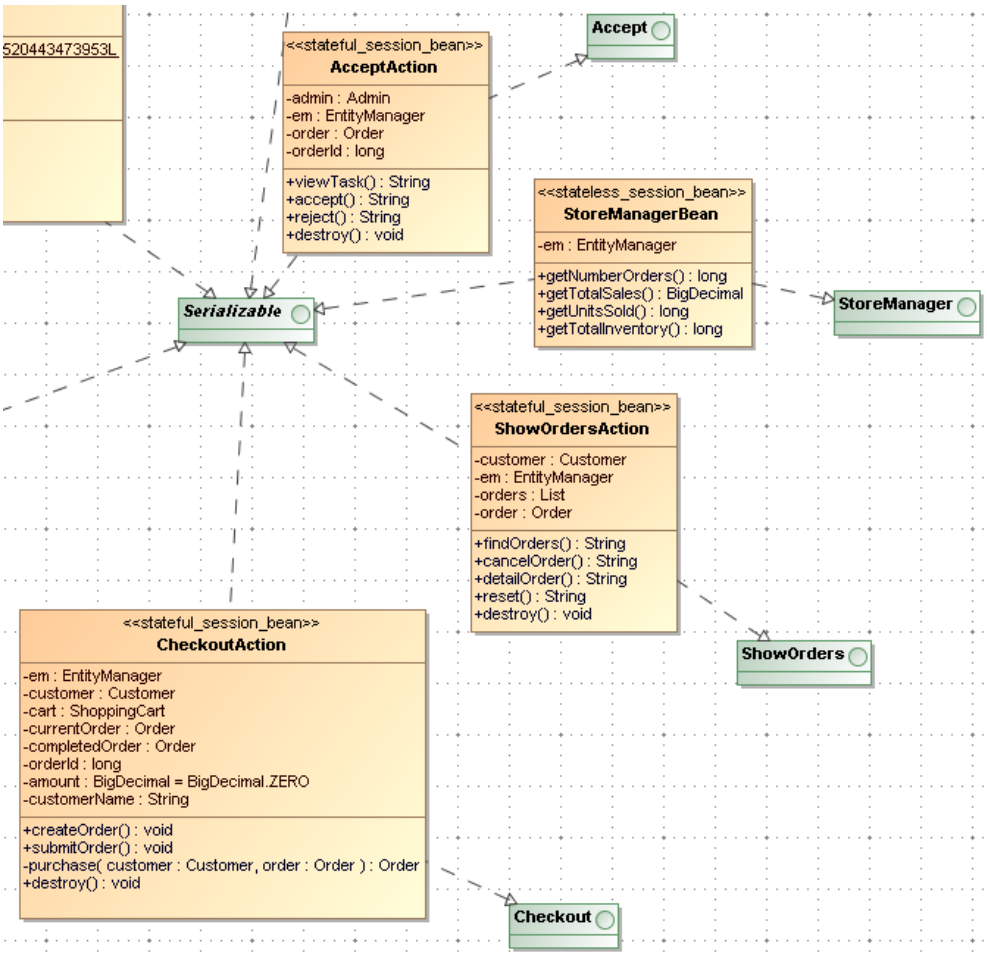


Abbildung 6.5.: Ausschnitt aus dem Klassenmodell der Session-Beans

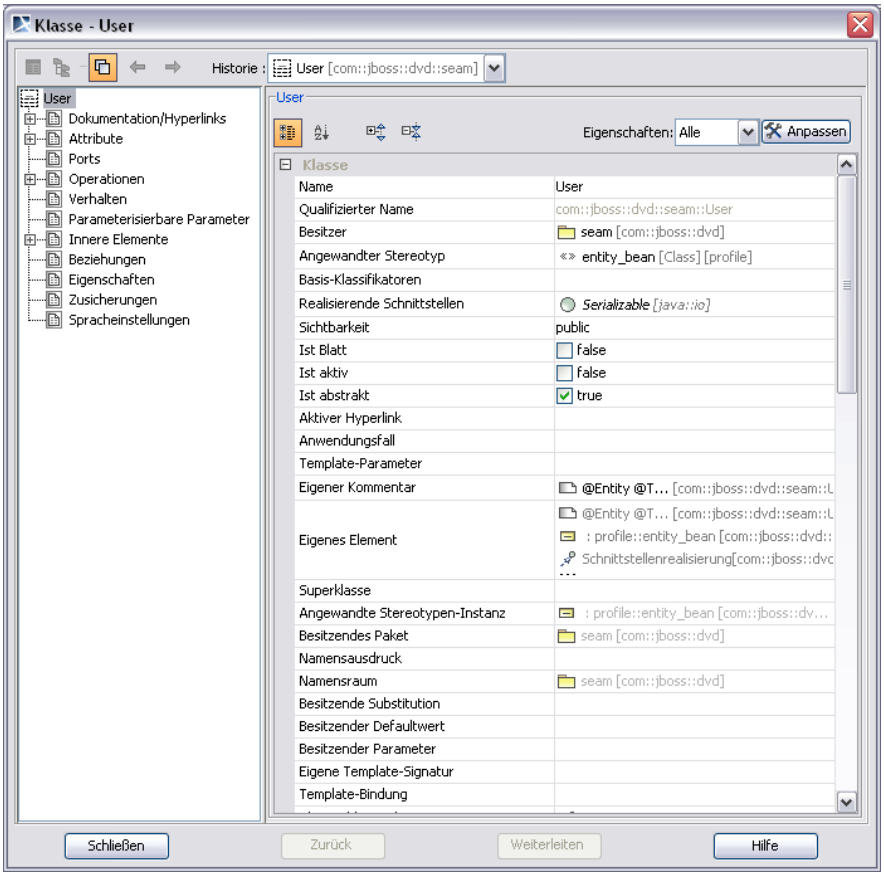


Abbildung 6.6.: Detailansicht der Klasse User

Für die Erstellung des Generators wird eine Ordnerstruktur gewählt, die übersichtlich und kompakt ist. Die Aufteilung in unterschiedliche Ordner vereinfacht die separate Erstellung von GUI und Backend. Abbildung 6.7 zeigt diese Struktur. Der Generator ist in folgende Ordner aufgeteilt:

- **src:** Hier befinden sich die Sourcefiles des Generators. Diese werden erneut aufgeteilt in Modelldatein, Templates und Java-Datein. Im Root des Ordners befindet sich die Workflow-Datei, die die Abläufe des Generators steuert. Die Templates teilen sich in zwei Ordner auf, was der Trennung von Back- und Frontend entspricht.

- **src-gen:** Dieser Ordner beinhaltet die durch den Backen-Generator erzeugten Javaklassen. Neben den Klassen lässt sich optional auch eine Textdatei erstellen, die sämtliche Informationen über die Generierung in Kurzform beinhaltet. Dadurch wird die Fehlersuche erleichtert, da sämtliche Informationen zum Generator nicht in den Javaklassen sondern in einer einzigen Datei zu finden sind.

- **src-web-gen:** In diesem Ordner befinden sich die generierten Webinhalte, die vom Frontend-Generator erzeugt werden. Auch hier kann optional eine Textdatei erstellt werden, die das Resultat der Generierung vereinfacht darstellt.

Im Folgenden wird auf die einzelnen Bereiche des Generators eingegangen.

### 6.5.1. Definition der Ablaufregeln

Die Steuerung des Generators erfolgt über das sogenannte Workflow-File, welches bereits in Kapitel 3 vorgestellt wurde. Dabei dienen 'Workflow-Components' als Bedienungselemente des Generators und verknüpfen die Transformationen, Validatoren und andere Generator-Elemente miteinander.

Das Workflowfile des JBoss-Seam-Generators ist in zwei separate Workflow-Files aufgeteilt, damit nicht bei jedem Generatordurchlauf der Back- und Frontend-Sourcecode erzeugt wird. Weiters lassen sich für den Fall, dass nur die Benutzerschnittstelle erzeugt werden soll, auch die GUI-Elemente unabhängig der EJBs erzeugen.

Das erste Workflow-File, *ejb_workflow.oaw*, definiert die Erstellung der

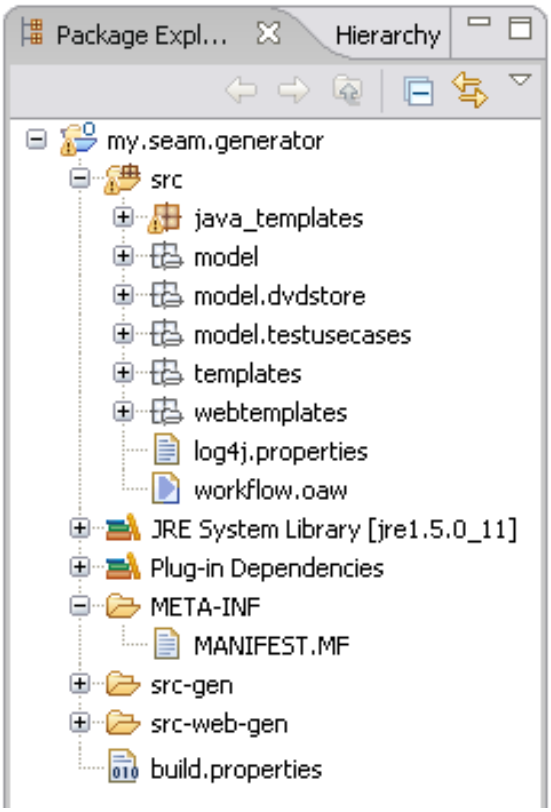


Abbildung 6.7.: Ansicht auf das Applikationspackage

Backend-Umgebung - also der Javaklassen für die Persistenz und die Geschäftslogik. Das Skript *web_workflow.oaw* erstellt die Benutzeroberfläche. Das Workflowskript für die Backend-Generierung besteht aus zwei Komponenten:

- **XMI-Reader:** Dieser Reader wird dazu benutzt, um die Modelle einzulesen. Zuerst muss der korrekte Pfad angegeben werden, der als Quelle für weitere Handlungen dient. Die XML-Datei wird durch den Reader eingelesen und eine Modellinstanz erstellt, welche durch das Element *outputSlot* der Workflowkomponente definiert wird. Im Falle des Beispiels können die modellierten Diagramme über die Variable *model* angesprochen werden. In Listing 6.1 ist die XMI-Komponente des Workflows zu sehen.

```
<component class="oaw.emf.XmiReader">
  <modelFile value="model/testusecases/dvdstore.uml"/>
  <outputSlot value="model"/>
</component>
```

Listing 6.1: XMI-Reader-Komponente des Workflows des Backend-Generators

- **Xpand-Generator:** In diesem Teil des Skripts werden zunächst die verwendeten Metamodelle definiert. Im Beispiel wird das Ecore- und das UML2-

Metamodell geladen. Als nächster Schritt werden die Metamodelle um das erstellte Profile ergänzt, welches die benötigten zusätzlichen Sprachelemente an den Generator bindet.

Nachdem die Metamodelle geladen sind werden Pfad-Informationen definiert. Als letzten Schritt werden die Templates aufgerufen, die das Modell bearbeiten. Als Postprozessor wird bei dem Backend-Generator der 'Java-Beautifier' gewählt, da der generierte Source-Code formatiert wird. Listing 6.2 zeigt den Aufruf des Generators innerhalb des Workflowfiles.

```
<component id="generator" class="oaw.xpand2.Generator"
   skipOnErrors="true">
  <metaModel class="oaw.type.emf.EmfMetaModel"
             metaModelPackage="org.eclipse.emf.ecore.
                EcorePackage"/>
  <metaModel class="oaw.uml2.UML2MetaModel"/>

    <metaModel id="profile" class="oaw.uml2.profile.
       ProfileMetaModel">
      <profile value="model/dvdstoreV16/profile.profile.uml
         "/>
    </metaModel>

    <prSrcPaths value="src-gen"/>
    <prDefaultExcludes value="false"/>

  <expand value="templates::Root::Root FOR model"/>
    <outlet path="src-gen">
       <postprocessor class="oaw.xpand2.output.
          JavaBeautifier"/>
    </outlet>
</component>
```

Listing 6.2: Generatoren-Komponente des Workflows des Backend-Generators

Dieses Workflow-Steuerungsskript befindet sich im *src*-Ordner dieser Arbeit und koordiniert die Abläufe des Generators.

### 6.5.2. Einlesen der Metamodelle und Profile

Das Einlesen der Metamodelle bindet die Generatoren an die in der Modellierung verwendeten Sprachelemente. Nachdem dieses wie bereits erwähnt über den Workflow mit dem Generator verbunden wurde, können die Stereotypen und Elemente direkt in den Templates verwendet werden.

Das Anpassen der Templates des Generators an die Metamodelle ist in Listing 6.3 zu sehen. Hier wird ein Modell beispielhaft in drei unterschiedliche Template-Definitionen aufgeteilt, wobei jeweils nach dem entsprechenden Stereotypen unterschieden wird.

```
«EXPAND Dummy_one FOREACH eAllContents.typeSelect(profile::
   stateless_session_bean)»
«EXPAND Dummy_two FOREACH eAllContents.typeSelect(profile::
   entity_bean)»
«EXPAND Dummy_three FOREACH eAllContents.typeSelect(uml::
   Class)»
```

Listing 6.3: Beispiel der Verwendung der Meta-Elemente

Neben UML 2.0 wird in dieser Arbeit kein weiteres Metamodell verwendet. Lediglich die Anpassung über das Profile wird genutzt, um zusätzliche Sprachelemente zur Verfügung zu stellen. Obwohl das oAW-Framework Ecore und andere Metamodelle erlauben würde, wird aus Gründen der Übersicht bewusst auf diese verzichtet, da die Modellierung mit mehreren Metamodellen als Basis eine Verwebung der Metamodelle erfordert würde. Dies wäre aufgrund der Komplexität der Aufgabe und der Applikation zu aufwendig.

### 6.5.3. Check Modellvalidierung

Die Verwendung von Validierungsmechanismen hilft dabei, den Generator schon im Vorfeld vor fehlerhaften Eingabemodellen zu schützen. Diese Validierung kann über das Anlegen von *Check*-Datein geschehen, welche die von openArchitectureWare vorgestellte Sprache XTend verwenden. Neben der Validierung über den Generator bieten die meisten Modellierungsumgebungen bereits vollständige Überprüfungen der Modelle an. Aus Gründen der Vollständigkeit wurden dennoch einige Checks erstellt, die wichtige Aspekte abdecken und sich teilweise nicht über das Modellierungswerkzeug MagicDraw überprüfen lassen. Listing 6.4

zeigt die erstellten Checks, welche die Beans innerhalb der Modelle auf deren Vollständigkeit überprüft.

```
import org::eclipse::uml2::uml;
import uml;
import profile;

//Entitäten haben einen Namen
context profile::entity_bean ERROR "An entity must have a
   name!":
this.name != null;

//Entitäten sollten Attribute besitzen!
context profile::entity_bean WARNING "Entity " + name +
   "... There are no field values defined!" :
this.ownedElement.typeSelect(uml::Property).size > 0;

//Operationen sollten Return-Werte besitzen!
context uml::Operation WARNING "Operation " + this.class.
   name + "." + name + ": An operation must have a return
   type!" :
this.type != null;
```

Listing 6.4: Check-Files des Backend-Generators

Die ersten zwei Regeln betreffen die Entitäten und überprüfen, ob diese Namen und Attribute besitzen. Das Resultat der Validierung der Attribute führt lediglich zu einer Warnung in der Systemconsole, die Namensüberprüfung hingegen liefert eine Fehlermeldung und stoppt den Generator. Der dritte Check überprüft, ob die Operationen einen Rückgabewert definiert haben. Dies resultiert ebenfalls in einer möglichen Warnung.

### 6.5.4. Templatestruktur

Die Struktur der Templates ist wie ein Baum aufgebaut. Die einzelnen DEFINE-Blöcke stellen die Blätter dar, die Wurzel wird mithilfe des Workflow-Parameters übergeben. Listing 6.2 zeigt die Übergabe des Modelles an die Tempaltes mithilfe des Aufrufes 'templates::Root::Root FOR model'. Dies bewirkt, dass der zuständige DEFINE-Block des Templates angesprochen wird, und die nächsten Schritte iterativ abarbeitet. Die direkte Steuerung über das Workflow-File erlaubt

es, mehrere Bäume für die einzelenen Bereiche der Applikation zu verwenden. Der erstellte Generator verfügt über zwei separate Bäume. Der erste Baum steuert die Generierung der Backend-Aspekte der Anwendung. Zum Aufrufen des entsprechenden Templates wird folgender Parameter innerhalb des Workflow-Files ausgeführt: 'templates::Root::Root FOR model'. Der zweite Baum kümmert sich um die Erstellung der Frontend-Aspekte.

Neben der Trennung der Eingabedaten ist der Generator auch in unterschiedliche Template-Files möglichst modular aufgeteilt. Innerhalb des Seam-Generators sind folgende Template-Datein für die Erstellung des Backends zuständig:

- **Annotation.xpt:** Dieses Template kümmert sich um die Erstellung der Annotationen. Dabei wird je nach Stereotyp des Kommentars das entsprechende Annotations-Element ausgegeben.

- **Attribut.xpt:** Das Attribut-Template kümmert sich um die Instanzierung und die Erstellung der Getter- und Setter der Attribute des Diagrammes.

- **Class.xpt:** Dieses zentrale Template kümmert sich um die Erstellung der Klassen und greift dabei auf andere Templates zurück, sobald ein entsprechender Datentyp gefunden wird. Neben der Generierung der Klasse werden auch die verschiedenen Klassen-Bodys erstellt, die je nach Stereotyp unterschiedliche Informationen beinhalten.

- **Interface.xpt:** Hier werden die Interfaces erstellt, die für die Session Beans benötigt werden.

- **Operation.xpt:** Dieses Template erstellt die Methoden der Klasse. Dabei wird zwischen Methoden für Konstruktoren, leere Methoden und Methoden für Interfaces unterschieden, da diese spezielle Eigenschaften erfüllen müssen.

- **Root.xpt:** Das Root-Template dient als Wurzel der Templates und führt die ersten Schritte bei der Generierung des Backends durch. Neben der Verteilung der Klassen werden auch Textdatein erstellt, die das Resultat der Generierung mitprotokolliert und somit einen besseren Überblick liefert.

Abbildung 6.8 zeigt diese Struktur anhand des Eclipse Workspaces.

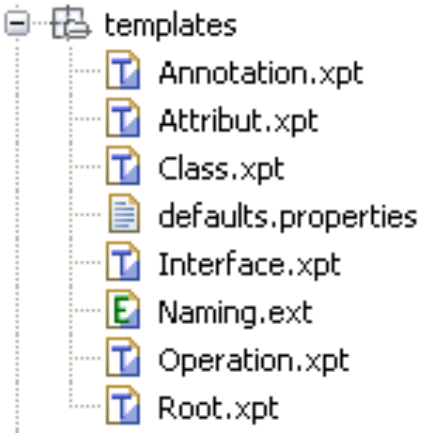


Abbildung 6.8.: Ansicht der Templatestruktur des Backend-Generators

### 6.5.5. Transformationen am Beispiel der Entitäten-Generierung

Dieser Abschnitt betrachtet einige der wichtigsten Transformationsregeln des Backend-Generators. Dabei werden nur einige Beispiele gezeigt, da eine komplette Darstellung der Templates nicht sinnvoll wäre und den Rahmen dieser Arbeit sprengen würde.

Die Templates und deren Zusammenspiel werden am Beispiel einer Entität erläutert. Dabei werden grob folgende Schritte durchlaufen:

- Aufruf des Root-Templates
- Erstellung der Klassenstruktur innerhalb des Class-Templates
- Importierung der notwendigen Java-Bibliotheken
- Instanzierung der Variablen über das Attribut-Template
- Erstellung der Methoden mittels Operation-Templates
- Zugriff auf die Templates der Annotationen
- Speicherung der Kurzinformationen über die Result-Textdatei

Aus Gründen der Übersicht werden die einzelnen Templates separat betrachtet, da diese sehr stark miteinander verwoben sind und eine iterative Erklärung an manchen Stellen für Verwirrung sorgen könnte.

### Root-Template

Nach der Übergabe und der Instanzierung des Modelles im Speicher wird das Root-Template aufgerufen. Dieses Template ist in Listing 6.5 dargestellt.

```
«IMPORT org::eclipse::uml2::uml»
«IMPORT uml»
«IMPORT profile»
«EXTENSION java_templates::Timer»
«EXTENSION java_templates::Datatype»

«DEFINE Root FOR Model»

  «FILE "!INFO_Result.txt"»
    TimeStamp: «getTimestamp()»

    Entity-Beans done - Details see file entity_beans.txt
    «FILE "!INFO_entity_bean.txt"»
      «EXPAND Result FOREACH eAllContents.typeSelect(
         profile::entity_bean)»
    «ENDFILE»
  «ENDFILE»

    «EXPAND Class::JavaClass FOREACH eAllContents.
       typeSelect(uml::Class).select(e|!e.
       getAppliedStereotypes().isEmpty)»

  «EXPAND PackageRoot FOREACH (List[uml::Package])
     ownedElement»
«ENDDEFINE»
```

Listing 6.5: Root-Template des Backend-Generators

Um die Sprachelemente ohne deren Präfix und Verzeichnisinformationen nutzen zu können, werden die notwendigen 'IMPORTS' vorgenommen. Weiters werden die für die Templates notwendigen Extensions definiert, die den Zugriff auf statische Methoden erlauben.

Das instanzierte Modell wird dem Root-DEFINE übergeben, welches daraufhin sofort eine Informationsdatei namens *!INFO_Result.txt* erstellt. Auf die Erzeugung und die Templates für diese Informationsdatei wird an dieser Stelle nicht

näher eingegangen. Die Erzeugung dieser Datei wurde auf die Informationen über die Generierung der 'entity_beans' reduziert und lässt sich mit einfachen Mitteln auf sämtliche Beans des Backend-Generators erweitern, sodass für jeden Stereotyp eine eigene Datei erzeugt werden kann.

Nach der Erzeugung der Textdatein werden sämtliche Elemente die den Stereotypen 'uml::Class' aufweisen an das Class-Template übergeben, welches im nächsten Abschnitt bahandelt wird.

### Class-Template

Das Class-Template ist in Listing 6.6 zu sehen und dient als Wurzel zur Generierung von Javaklassen.

```
«DEFINE JavaClass FOR uml::Class»
  «FILE name+".java"»

    «fileComment(getTimestamp(), this)»
    «packageDefinition(this.package)»;
    «EXPAND ClassImports FOR this»
    «EXPAND Annotation::AnnotationElement FOREACH
       ownedComment»

    «visibility.toString()» «IF isAbstract»abstract«ENDIF»
       «getType(this)» «name»

    «FOREACH superClass AS e»
      «IF e==superClass.first()»
        extends «e.name»
      «ENDIF»
    «ENDFOREACH»

    «IF (!getImplementedInterfaces().isEmpty)»
      implements
    «ENDIF»

    «FOREACH getAllImplementedInterfaces() AS e SEPARATOR
       ", "»
      «e.name»
    «ENDFOREACH»
```

```
    {
      «EXPAND JavaClassBody FOR this»
    }
  «ENDFILE»
  «EXPAND Interface::Interface FOREACH
     getAllImplementedInterfaces()»
«ENDDEFINE»
```

Listing 6.6: Ausschnitt aus dem Class-Template des Backend-Generators

Die Generierung der Javaklassen im Template beginnt mit der Erstellung der .java-Datei, welche den Namen der Entität aus dem Diagramm übernimmt. Als nächstes wird ein Kommentar der Klasse erzeugt, welche mit dem Datum und Zusatzinformationen aus der Entität versehen ist. Dies hilft dabei, die Klassen zu versionieren und Standard-Informationen einzubetten. Nach dem Kommentar wird das Package definiert und die Imports der Klasse erzeugt. Die zu importierenden Bibliotheken werden ebenfalls dem Modell entnommen, wobei das entsprechende Template auf die dem Diagramm hinzugefügten Elemente zugreift und jede Bibliothek importiert, die in der entsprechenden Entität genutzt wird. Wird beispielsweise der Datentyp *java.util.Hashmap* im Diagramm verwendet, wird die dazu nötige Bibliothek importiert.

Nachdem die Imports erzeugt wurden, werden Annotationen hinzugefügt und die Klasse unter Berücksichtigung von implementierten Schnittstellen und Super-Klassen 'initialisiert'. Der Inhalt der Klasse, der sogenannte Klassen-Body wird über ein weiteres Template erzeugt, welches je nach Stereotyp der Klasse unterschiedlichen Informationen benötigt. Listing 6.7 zeigt das entsprechende Template für die Generierung der Body-Inhalte der der Entitäten.

```
«DEFINE JavaClassBody FOR profile::entity_bean»
  «REM» Variablen definieren  «ENDREM»
    «EXPAND Attribut::Attribut_Definition FOREACH
       getAllAttributes().select(e|e.getOtherEnd()==null)»
  «REM» Leerer Konstruktor  «ENDREM»
    «EXPAND Operation::Konstruktor_leer FOR this»
  «REM» Konstruktor mit allen Variablen der Klasse, wenn
     Variablen vorhanden! «ENDREM»
    «IF getAllAttributes().isEmpty==false»
      «EXPAND Operation::Konstruktor FOR this»
    «ENDIF»
  «REM» Getter und Setter erstellen «ENDREM»
```

```
    «EXPAND Attribut::Attribute_getter_setter FOREACH
       getAllAttributes().select(e|e.getOtherEnd()==null)»
  «REM» Operationen erstellen «ENDREM»
    «EXPAND Operation::Operation FOREACH getOperations()»
«ENDDEFINE»
```

Listing 6.7: Classbody-Template des Backend-Generators

Nach der Erzeugung der Body-Informationen werden etwaige Interfaces über das entsprechende Template der Interfaces behandelt. Zuvor wird die Bearbeitung der aktuellen Javaklasse abgeschlossen. Dies geschieht über den Aufruf 'ENDFILE', der den Output-Stream in die Datei beendet.

### Attribut-Template

Das Attribut-Template kümmert sich um alle für die korrekte Funktion der Eigenschaften der Klassen notwendigen Schritte. Dies betrifft unter Anderem die Instanzierung der Attribute und die automatische Generierung von Gettern und Settern.

Im Falle einer Entität wird das Attribut-Template für sämtliche definierten Inhalte verwendet, wobei als Beispiel die Erstellung der Getter in Listing 6.8 dient. Dabei werden zuerst die Annotationen über das zuständige Template erzeugt. Als nächstes wird die Methode selbst implementiert, wobei das Präfix 'get' vor den Namen der neuen Methode vorangestellt wird. Zuletzt wird noch die verlangte Variable mittels 'return'-Sstatement zurückgegeben, was auch schon das semantische Ende des Templates darstellt.

```
«DEFINE Attribut_getter FOR uml::Property»
«REM» Annotations setzen  «ENDREM»
  «EXPAND Annotation::AnnotationElement FOREACH
     eAllContents.typeSelect(profile::getterAnnotation)»
  public
  «getType(this)» get«name.toFirstUpper()»()
  {
    return «name»;
  }
«ENDDEFINE»
```

Listing 6.8: Ausschnitt aus dem Attribut-Template des Backend-Generators

Nach dem kurzen Einblick in die Arbeitsweise des Attribut-Templates wird im folgenden Abschnitt auf die Operationendefinition eingegangen.

### Operation-Template

Im Operation-Template werden Methoden erzeugt und Konstruktoren erstellt. Dieses Template wird zumeist vom Classbody-Template aufgerufen und erstellt beispielsweise den Konstruktor der Entität, wie er in Listing 6.9 zu sehen ist. Hierbei handelt es sich um jene Konstruktorenerzeugung, die sämtliche Attribute der Klasse miteinbezieht und die Variablen der Klasse setzt.

```
«REM» Default Konstruktor (Alle Attribute)  «ENDREM»
«DEFINE Konstruktor FOR uml::Class»
  «visibility.toString()» «name»(
  «FOREACH getAllAttributes().select(e|(e.getOtherEnd()==
     null&&(e.type.name.contains("enum")==false))) AS e
     SEPARATOR ", "»
    «getType(e)» «e.getLabel()»
  «ENDFOREACH»
  ) {
  «FOREACH getAllAttributes().select(e|(e.getOtherEnd()==
     null&&(e.type.name.contains("enum")==false))) AS e»«IF
      e.isStatic==false»      this.«e.getLabel()» = «e.
     getLabel()»;«ENDIF»
  «ENDFOREACH»
  }
«ENDDEFINE»
```

Listing 6.9: Ausschnitt aus dem Operations-Template des Backend-Generators

Die Ausführung des Templates generiert den Methodenaufruf und fügt sämtliche nicht statische Attribute in die Parameterdefinition ein. Danach werden diese mit 'this.NAME = NAME' abgearbeitet, wobei 'NAME' für den Namen des Attributes aus dem Diagramm steht.

### Interface-Template

Im Interface-Template werden die Interfaces zu den Session-Beans erzeugt. Hierbei wird ähnlich der normalen Klassenerstellung vorgegangen, wobei das bereits

vorgestellte ClassBody-Template 6.7 nicht ausgeführt wird. Stattdessen wird für jede öffentliche Methode eine leere Zugriffsmethode erzeugt, die den Zugriff auf die Operationen des Session Beans zulässt. Der Sourcecode des Interface-Templates ist in 6.10 zu sehen.

```
«REM» Interfaces erzeugen «ENDREM»
«DEFINE Interface FOR uml::Interface»
  «IF !isAbstract»
    «FILE name+".java"»

      «fileComment(getTimestamp(), this)»

      «packageDefinition(this.package)»;

      «interfaceImports()»

      «EXPAND InterfaceImports FOR this»

      «EXPAND Annotation::AnnotationElement FOREACH
         ownedComment»

      «visibility.toString()» interface «name»
      {
        «EXPAND Operation::Interface_Methode_leer FOREACH
           ((uml::Class)getTargetDirectedRelationships().
           source.first()).getAllOperations()»
      }
    «ENDFILE»
  «ENDIF»
«ENDDEFINE»
```

Listing 6.10: Ausschnitt aus dem Interface-Template des Backend-Generators

Nachdem nun die wichtigsten Templates für die Erstellung der Beispielentität gezeigt wurden, wird im folgenden Abschnitt das Ergebnis der Generierung anhand eines einfachen Beispiels gezeigt.

## 6.6. Ergebnis der Generierung

Die Generierung der Backend-Datein des Generators beschränkt sich auf die Erstellung von Java-Klassen. Als Beispiel in der Templatebeschreibung wurde eine Entität gewählt, da diese sehr klare Strukturen haben und stets identisch aufgebaut sind. Nachdem eine Klasse generiert wurde, wird diese zuletzt durch den sogenannten *Java-Beautifier* strukturiert und für einen Benutzer übersichtlich dargestellt.

Das Resultat eines Generierungsvorganges zeigt Listing 6.11, welches die Entität des *Users* als Javaklasse zeigt. Der Sourcecode wurde gekürzt und einige Attribute und Methoden entfernt, um die Darstellung kompakter zu gestalten. Die Klasse im Modell lässt sich auf Abbildung 6.4 erkennen und ist ein zentrales Element, welches als Superklasse für den Kunden und den Administrator dient. Weiters ist sie im Modell als 'abstract' gekennzeichnet, was sich im Sourcecode widerspiegelt.

```
/*Mon Oct 08 23:38:41 CEST 2007
 * Generated Entity-File: User.java
 * Generated by Thomas Hiebler
 * Generated with my.seam.generator */
package com.jboss.dvd.seam;

+ import org.hibernate.annotations.Cache;

@Entity
@Table(name = "USERS")
public abstract class User implements Serializable {
    private long id;
    private String userName;

    public User() {
    }
    public User(long id, String userName) {
        this.id = id;
        this.userName = userName;
    }
    @Column(name = "USERNAME", unique = true, nullable =
       false, length = 50)
    @NotNull
```

```
    @Length(min = 4, max = 16)
    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
    @Transient
    public boolean isAdmin() {
        /*PROTECTED REGION ID(User_isAdmin) ENABLED START*/
        // Insert Code here!

        // Default Return-Statement (if available):
        return false;

        /*PROTECTED REGION END*/
    }
}
```

Listing 6.11: Vereinfachte User-Klasse

Der Javacode lässt leicht die einzelnen Teile des Generators erkennen und lässt sich mithilfe der im vorherigen Kapitel beschriebenen Templates fast gänzlich ergründen. Zuerst startet die Klasse mit dem Generator-Kommentar und den Importieren der Java-Bibliotheken. Als nächster Schritt werden folgende Elemente in den *Class-Body* übertragen:

- Initialisierung der Attribute
- Konstruktoren
- Getter- und Setter-Methoden
- Manuelle Implementierungen

Während die ersten drei Inhalte des *Class-Bodys* bereits behandelt wurden, ist auf die Definition der manuellen, händisch zu implementierenden Methoden noch nicht eingegangen worden. Diese befinden im obrigen Beispiel in einem geschützten Bereich, der 'PROTECTED REGION'. Diese Möglichkeit der manuellen Erweiterungen der Javaklassen wird unter Anderem im nächsten Abschnitt näher betrachtet.

## 6.7. Manuelle Anpassungen

Die Generierung von Sourcecode basierend auf UML Modellen erlaubt es, eine Vielzahl von Aspekten der finalen Applikation mit den zur Verfügung stehenden Diagrammen abzudecken. Jedoch ist es in den seltensten Fällen möglich, sämtliche Bereiche automatisch zu generieren. Der Grund dafür ist zumeist systemspezifische Fachlogik-Elemente, die nicht über Diagramme dargestellt werden können.

Um den über Modelle entwickelten Sourcecode mit manuellen Informationen anzureichern, gibt es nach [Rol06] verschiedene Strategien die den manuellen vom automatisch erstellten Code zu trennen.

- **Separate Dateien:** Diese Methode ist je nach Technologie unterschiedlich verwendbar. In Bezug auf die Erstellung einer Java-Applikation lassen sich manuelle und generierte Codeartefakte in zwei oder mehrere Datein aufspalten.

  Zum Beispiel befindet sich der generierte Teil einer Entität innerhalb einer abstrakten Java-Klasse. Dieser soll und darf zum Schutz der Generierungsergebnisse nicht manuell bearbeitet werden und bleibt zu 100 Prozent generiert. Neben dieser Basisklasse wird eine zusätzliche Klasse erstellt, die die Basisklasse um die manuellen Konzepte erweitert. Die Erweiterungsklasse darf durch den Generator nicht verändert werden. Diese Vererbung erlaubt die Trennung der manuellen von den generierten Bereichen.

- **Geschützte Bereiche:** Diese Strategie erlaubt die einfachste Möglichkeit der Trennung der generierten von den manuellen Implementierungen. Dies geschieht über sogenannte *Protected Areas* im generierten Sourcecode, der vom Generator nicht erkannt und nicht überschrieben wird. Innerhalb dieses Bereichs, der mit einer eindeutigen ID gekennzeichnet ist, lassen sich die manuellen Implementierungen in die Applikation integrieren. In Listing 6.11 lässt sich diese Verwendung der geschützten Bereiche erkennen.

  So einfach diese Methode auch ist birgt sie auch einen entscheiden Nachteil. Der manuelle Code ist innerhalb der generierten Datei eingebettet und nicht klar getrennt in mehreren Datein. Somit kann es Probleme beim Versionsmanagement oder der Entwicklung kommen, da sich der Entwickler innerhalb des generierten Codes bewegen und diesen auch verstehen muss.

- **Anreicherung der Modelle:** Eine weitere Möglichkeit der Trennung ist über

die Modellierung von Implementationsdetails in die Modelle. Dies lässt sich beispielsweise über Kommentare oder andere sogenannte 'Tagged Values' lösen, die das Modell mit Implementierungskonzepten verunreinigen. Das Hauptproblem dieser Strategie ist die Verschmelzung der Verantwortlichkeiten, da Experten der Fachlogik innerhalb des Sprachbereichs der Modellierer arbeiten, was zwangsläufig zu Fehler in den Modellen führt.

Im Zuge dieser Arbeit wurde die Trennung der Implementierungsvarianten mittels geschützter Bereiche gelöst, was folgende Gründe hat:

- Der Java-Beautifier von openArchitectureWare zerstört die geschützten Bereiche im Bereich der Imports der Java-Bibliotheken. Eine Kombination zweier Varianten ist somit nicht denkbar, da der Verwaltungsaufwand bei einer Verwendung rapide ansteigen würde.
- Die Trennung in mehrerer Datein reduziert die Übersicht bei sehr vielen Javaklassen.
- Die Verunreinigung der Modelle mit Implementationsdetails ist nicht sinnvoll für größere Applikationen.
- JBoss Seam erzeugte Fehlermeldungen bei komplexeren Vererbungsbeziehungen und hatte Probleme bei der Verarbeitung von abstrakten Methoden.

Neben der Verwendung der geschützten Bereiche gibt es im Seam-Generator eine zusätzliche Möglichkeit, die Implementierung außerhalb der Modelle zu beeinflussen. Dies hat den Hintergrund, dass die Modellierungsumgebung von MagicDraw einige Kombinationen von Sprachelementen nicht vorsieht und somit nicht erzeugen lässt. Als Beispiel dafür dient folgende Instruktion, die in Listing 6.12 dargestellt ist

```
private Order cartOrder = new Order();
```

Listing 6.12: Modellierungsproblem in MagicDraw

Das Attribut lässt sich zwar in den Diagrammen darstellen, die Initialisierung mit einem Standardwert funktioniert aber bei komplexeren Datentypen, wie es in diesem Beispiel der Fall ist, nicht. Um diese Implementierung dennoch zu erzeugen, wurde eine Textdatei erstellt die während der Generierung eingelesen wird und an der entsprechenden Stelle für den korrekten Ausdruck sorgt. Mit-

hilfe dieser Implementierung ist es möglich, Bugs oder Fehler im verwendeten Modellierungswerkzeug zu umgehen. Der Ausdruck innerhalb der Textdatei ist in Listing 6.13 zu sehen.

```
Data::com::jboss::dvd::seam::ShoppingCartBean::cartOrder
   private Order cartOrder = new Order();
```

Listing 6.13: Manuelle Behebung des Modellierungsproblems

Im Folgenden Abschnitt wird auf mögliche Erweiterungen des Generators eingegangen.

## 6.8. Erweiterungsmöglichkeiten

Obwohl der Generator bereits gute Ergebnisse liefert, gibt es eine Vielzahl von Erweiterungsmöglichkeiten die den Generator verbessern oder die Performance erhöhen würden. Folgende Aufzählung liefert einen kurzen Überblick über mögliche Erweiterungspotentiale:

- **Operationen zusammenfassen:** Derzeit sind die Operationen stark getrennt und recht spezifisch aufgebaut. Eine Vereinheitlichung und die Reduzierung auf ein Standard-Operationen-Template das von anderen mit Parametern aufgerufen wird, könnte eine bessere Übersicht bieten und weitere Adaptionen vereinfachen. Weiters wäre es möglich neue Funktionalitäten an nur einer Stelle, dem Operations-Template, zu verbessern und muss nicht auf redundante Templatebereiche achten.

- **Erweiterung der Typenerfassung:** Die Typendefinition wurde beim Seam-Generator mittels einer Java-Methode gelöst, welche den Typen aufschlüsselt und ihn in ein für Java verwendbaren Typen-Syntax verarbeitet. Zusätzlich werden Standardausgaben und sogenannte Default-Werte auch über diese Java-Methode geregelt. Als Verbesserungsmöglichkeit würde sich eine Integration in die Modelle oder eine einzulesende zusätzliche XML-Datei anbieten.

- **Testinhalte für SQL Imports:** Für das Füllen der Datenbank mit ersten Daten würde sich über eine Wörterbuch-Datei oder über Standardinhalte realisieren lassen. Dadurch wäre ein erster Test der Datenbank möglich, ohne sich über die Inhalte Gedanken zu machen.

- **Klassensystem zusammenfassen:** Ähnlich der Templates der Operationen würde sich das Klassensystem zusätzlich erweitern lassen. Dadurch ließe sich der Generator noch kompakter implementieren und flexibler in Bezug auf die Verwendung von externen Parametern steuern.

- **Importsystem auslagern:** Das dynamische Importsystem des Generators könnte man in ein eigenes Profile laden, damit jeder Modellierer es in seine Diagramme mit einfachen Mitteln importieren und die zusätzlichen Sprachmöglichkeiten nutzen könnte. Dies würde die Modellierung einfacher gestalten, aber das Hinzufügen von neuen Datentypen erschweren, da das entstandene Profile gewartet und verteilt werden muss.

Natürlich stellt diese Auflistung nur eine kleine Auswahl an möglichen Verbesserungen dar. Es wäre neben der oben genannten Punkte auch eine Steigerung der Performance und eine genauere Validierung der Modelle in den nächsten Generator-Versionen möglich.

# 7. Der Frontend-Generator

Dieses Kapitel widmet sich der Erstellung der Frontend-Aspekte und der Generierung einer möglichen Benutzerschnittstelle. Mithilfe dieser Schnittstelle ist es dem Benutzer möglich, auf die Applikation zuzugreifen und die Funktionalitäten über eine graphische Oberfläche zu nutzen. Diese besteht aus vielen einzelnen Komponenten und Technologien, welche bereits in Kapitel 5 vorgestellt wurden.

Bei der Erstellung der Benutzeroberfläche mithilfe des Seam-Generators wurde auf die Verwendung von HTML-Tags bewusst verzichtet, da diese primär für die Optik der Applikation zuständig und somit sehr individuell anzupassen sind. Diese visuellen Eigenschaften sind schwer zu automatisieren, da diese stark im Zusammenhang mit Design und Usability der Webseite stehen, und von Seite zu Seite unterschiedlich definiert werden.

In den folgenden Abschnitten werden die Architektur und die Diagramme näher betrachtet. Zudem gibt es einen Überblick über die Templates, die für die Generierung der Oberfläche zuständig sind. Zuletzt wird noch auf mögliche Erweiterungen des Generators eingegangen.

## 7.1. Systemarchitektur

Wie bereits in Kapitel der Referenzapplikation gezeigt wurde (siehe dazu 5.5), setzt sich die Benutzeroberfläche aus verschiedenen Komponenten zusammen. Neben JSF und Facelet-Elementen werden noch JSP und HTML-Tags verwendet, wobei letztere ausschließlich zur Definition der trennbaren Bereiche mithilfe der 'Div-Tags' dienen.

Das Frontend der Referenzapplikation besteht nach einem Aufspalten der einzelnen Komponenten aus 20 xHTML-Datein, 3 JSP-Datein und einer Vielzahl von Konfigurationsfiles. Die Trennung der JSF-Komponenten lässt eine modula-

re Verwendung zu und unterstützt die Generierung.

Abbildung 7.1 zeigt die Ordnerstruktur des Frontend-Bereichs. In einem geschützten Ordner befinden sich die für den Administrator verwendeten Dateien. Zugriff auf diese ist nur im eingeloggten Zustand möglich. Dieser Sachverhalt ist in einer der Konfigurationsdateien festgelegt.

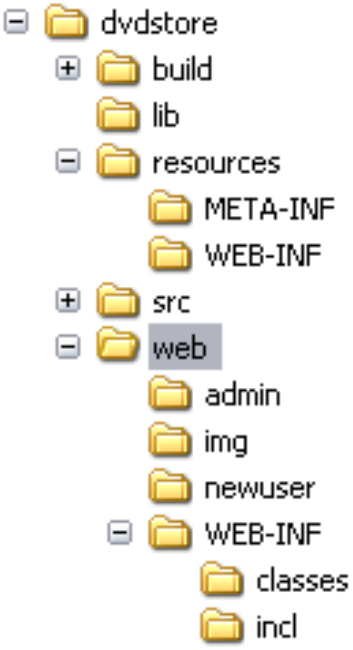


Abbildung 7.1.: Ordnerstruktur des Frontends

Neben dem geschützten Admin-Ordner gibt es noch Ordner, die für den Zugriff mittels jBPM-Prozessen vom Rest der Applikationsdateien getrennt sind. Der *newuser*-Ordner behinhaltet beispielsweise die notwendigen xHTML-Dateien für das Anlegen eines neuen Benutzers, dessen Arbeitsweise wie bereits erwähnt über einen Prozess gesteuert wird. Im Ordner *img* befinden sich Bilder, die von der Benutzeroberfläche verwendet werden.

## 7.2. Planung und Zielsetzung der Entwicklung

Für die Planung des Frontend-Generators wurde die Referenzapplikation analysiert und hinsichtlich der Generierbarkeit näher betrachtet. Für die Transformationsregeln war es notwendig, die xHMLT-Seiten möglichst modular zu gestalten. Dies wird mit 'include'-Statements verwirklicht, die von einer anderen Seite, eine Art Wurzel eines Seiten-Aufbau-Baumes, eingebunden werden.

Das Ziel des Frontend-Generators ist die Erstellung einer Art 'Skelett'-Benutzeroberfläche, die die grundlegenden Elemente der Seite beinhaltet, jedoch keine Informationen über die Darstellung definiert. Dies ist ein logischer Schritt, da die Applikationsoberfläche ohnehin meist von Designern bearbeitet wird. Das Resultat des Generators ist demnach ein funktionierendes Zusammenspiel der Elemente, beispielsweise Textboxen, Buttons oder Textstücke, deren Verknüpfungen bereits funktionieren, sodass das Testen der Fachlogik möglich ist. Lediglich die Darstellung muss nach der Generierung noch angepasst werden, wobei dies vorzugsweise in einem eigenen Designerwerkzeug durchzuführen ist.

Ein weiterer Grund für den Fokus auf eine möglichst einfache, klar strukturierte Benutzeroberfläche ist die Tatsache, dass es innerhalb der HTML oder xHTML-Datein kein funktionierendes Werkzeug gibt, das die manuellen von dem automatisch erzeugten Inhalten trennt. Es gibt im Gegensatz zum Backend-Generator (siehe 6) keine sogenannten 'Protected-Areas', die für eine Trennung zuständig sind. Daher soll das Ergebnis des Generators möglichst einfach und klar verständlich sein, da dieses ohnehin noch weiter verfeinert werden muss.

Das Hauptproblem bei der Entwicklung des Frontend-Generators ist das Fehlen eines passenden Diagramms in UML. [Dou07] bringt diese Tatsache mit folgendem Satz auf den Punkt:

> *There is still no standard notation in UML for defining user interfaces or screen flows.*

Aus diesem Grund wurde nach einer Möglichkeit in der UML Modellierung gesucht, die Seiteninhalte darzustellen und möglichst gut an das Backend und die Entitäten zu knüpfen. Das angewandte Konzept wird später im Abschnitt 7.4 näher betrachtet.

## 7.3. UML 2.0 Profil

Ähnlich dem Backend-Generator in Kapitel 6 wird UML für die Arbeit mit dem Frontend-Generator um Sprachelemente erweitert. Die folgenden Modellierungselemente dienen der Modellierung der Benutzeroberfläche und sind im Profil zu finden:

<<**decision Input**>> Ein 'Decision-Input' ist ein Kommentar, welches einen

Entscheidungsknoten im Diagramm näher erklärt. Diese genauere Definition der Entscheidung soll bei der Implementierung helfen und gibt an, unter welchen Bedingungen der Aktivitätsfluß fortgesetzt wird.

<<**includeItem**>> Dieses Stereotyp beschreibt einen Anwendungsfall im dazugehörigen Diagramm näher und steht für eine JSF-Komponente, die von der darunterliegenden Seite eingebunden, also inkludiert, wird.

<<**naviItem**>> Ein 'naviItem' stellt ein Platzhalter für die Navigationsleiste dar, die permanent sichtbar ist. Ausgehend von der Startseite wird jeder Anwendungsfall dieses Stereotyps in diese Leiste aufgenommen und ist somit von jeder Unterseite aus erreichbar.

<<**webForm**>> Eine 'webForm' repräsentiert ein Formular innerhalb einer xHTML-Seite. An einer solchen Form können Action-Pins für die Ein- und Ausgabe von Daten angebracht werden. Eingangs-Pins werden direkt für die Ausgabe auf der Webseite angelegt und geben dem Benutzer Informationen über die Seite. Ausgang-Pins definieren Objekte oder Datenfelder, die sich auf Entitäten aus dem Klassendiagramm beziehen, und somit die Felder der Webseite definieren. Beispielsweise lässt ein Attribut mit dem Datentyp String eine Textbox die Bestätigung der Seite entstehen.

<<**webFormDecision**>> Dieses Stereotyp ist eine Spezialisierung von 'UML:DecisionNode' und dient der Trennung von unterschiedlichen Abläufen im Aktivitätsdiagramm. Jeder aus dem Knoten gehende Aktivitätsfluß wird separat abgehandelt und in verschiedene Formulare aufgeteilt, wobei deren Sichtbarkeit auf die definierten Ausgänge gesetzt werden. Beispielsweise wird ein generiertes Webformular nur gezeigt, wenn der Benutzer eingeloggt ist. Dies würde durch ein 'identity.loggedIn' abgefragt werden.

Das UML Profil ist in Abbildung 6.2 dargestellt.

## 7.4. Modellierung

Da die UML kein Standarddiagramm für die Definition von Weboberflächen oder Seitenflüssen bietet, wurde eine alternative Möglichkeit der Darstellung erarbeitet. Hierbei handelt es sich um eine Kombination aus drei verschiedenen Dia-

grammen:

- Aktivitätsdiagramm
- Klassendiagramm
- Anwendungsfalldiagramm

Das Konzept für den Frontend-Generator bedient sich der Informationen aus der Verhaltensmodellierung. Diese zeigen die Interaktion mit dem Benutzer mit dem System, welche sich auf die Verwendung der Webseiten der Applikation mappen lässt.

Abbildung 7.2 zeigt einen Ausschnitt aus dem Anwendungsfalldiagramm der Referenzimplementierung. In diesem Modell erkennt man die Verknüpfung der Anwendungsfälle mit dem Kunden, der als Akteur dargestellt ist. Die Anwendungsfälle sind den entsprechenden Stereotypen des Profils zugeordnet, welche bereits in Abschnitt 7.3 vorgestellt wurden. Der Benutzer verfügt beispielsweise über die Möglichkeit, sich am System anzumelden und sich einzuloggen. Diese sind über die Anwendungsfälle *login* und *register* implementiert.

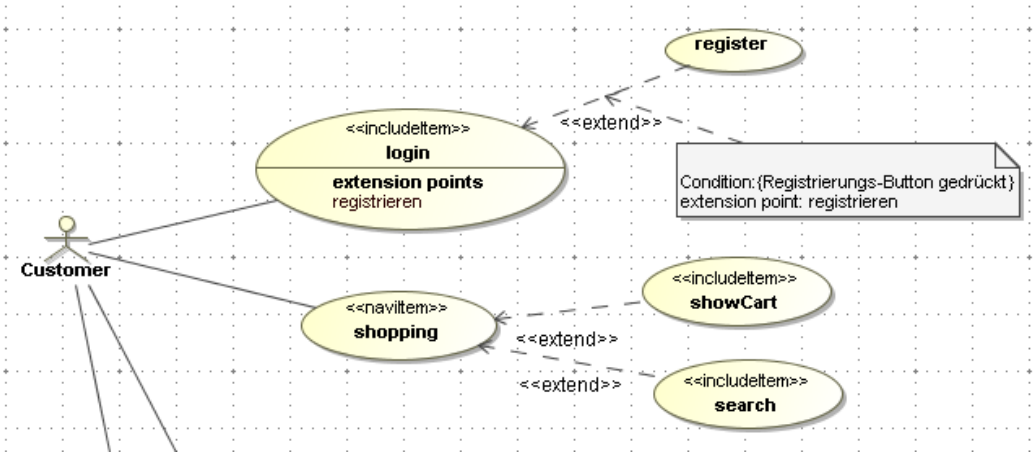


Abbildung 7.2.: Ausschnitt aus einem Frontend-Anwendungsfalldiagramm

Bei jedem Anwendungsfall im Diagramm lässt sich ein dazugehöriges Aktivitätsdiagramm hinterlegen, das die wichtigsten Aspekte der Interaktion in verfeinerter Darstellung modellieren lässt. Am Beispiel der Anmeldung am System, dem 'login', zeigt Abbildung 7.3 einen Ausschnitt aus einem solchen Diagramm. Zu erkennen ist neben dem Laden des Formulares auch eine Verzweigung, die darauf reagiert, ob der Benutzer bereits im System angemeldet ist. Sollte dies nicht der Fall sein, ist der Benutzer in der Lage, sich anzumelden. Sollte er be-

reits eingeloggt sein, wird ihm die Möglichkeit geboten, sich auszuloggen, was in der Abbildung dargestellt wird.

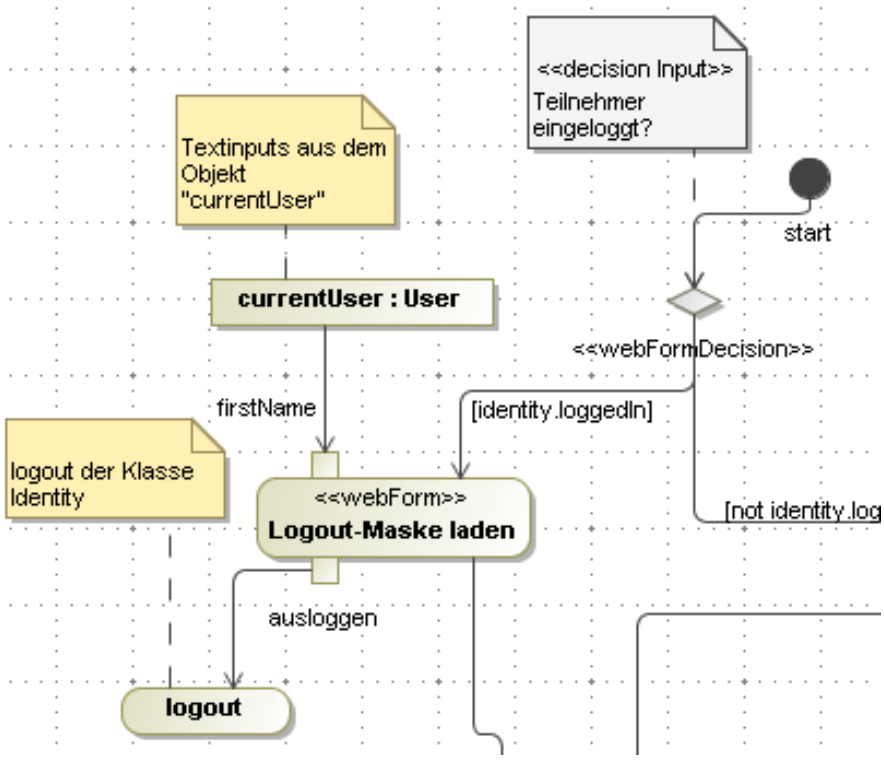


Abbildung 7.3.: Ausschnitt aus einem Frontend-Aktivitätsdiagramm

Durch die Verwendung der Aktion-Pins lässt sich das Formular an die Klassendiagramme binden und beispielsweise Methoden aufrufen. Dabei wird überprüft, ob die angesprochene Klasse über eine *Seam-spezifische ID* verfügt, welche durch die Annotation '@Name' definiert wird. Sollte dies der Fall sein, wird anstelle des Objekts im Diagramm der korrekte Wert der ID übernommen. Diese Art der Bindung an das Klassendiagramm bietet den Vorteil, dass die Benutzeroberfläche dynamisch auf das Backend zugreift und keine starren Verknüpfungen beinhaltet. Redundanzen werden somit vermieden.

## 7.5. Beschreibung und Ablauf des Generators

Dieser Abschnitt beschäftigt sich mit den Templates für die Generierung des Skelett-Frontends. Ähnlich der Templates für das Backend wurden die Scripts des Frontend-Generators in unterschiedliche xPand-Dateien aufgeteilt. Die folgende Auflistung erklärt die zwei Dateien und bietet einen kurzen Überblick über deren Funktionalitäten.

- **WebRoot.xpt:** Dieses Template ist die Wurzel für die Erzeugung der Webseiten. Hier befinden sich auch die DEFINE-Blöcke für die Generierung der Formulare und der JSF-Dateien.

- **WebStart.xpt:** Das WebStart-Template generiert die grundlegenden Dateien, die für die Verwendung der Benutzerschnittstelle wichtig sind. Als Beispiel hierfür dient das Anlegen der Template-Datei für die JSF-Komponenten und die Navigationselemente.

Wie bereits in 6 gezeigt wurde, ruft das Konfigurationsfile für den Generator den Frontend-Generator auf. Dies geschieht mittels dem Befehl 'templates::Root::WebRoot FOR model', welcher die Modelle in das entsprechende Template lädt. Nach der Erzeugung der grundlegenden Codeteile wie beispielsweise der Navigationselemente und die JSF-Konfigurationstemplates, wird eine Startseite erzeugt. Ein Ausschnitt aus dem Template dafür wird in Listing 7.1 gezeigt.

```
«DEFINE HomeDefinition FOR Model»
  => home.xhtml anlegen

  «FILE "home.xml"»
  <html xmlns="http://www.w3.org/1999/xhtml"
        xmlns:s="http://jboss.com/products/seam/taglib"
        xmlns:ui="http://java.sun.com/jsf/facelets"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:h="http://java.sun.com/jsf/html">

  <body>
    <ui:composition template="/template.xhtml">

          <ui:define name="navigation">
              <ui:include src="/navigation.xhtml"></ui:
                 include>
          </ui:define>

    «FOREACH eAllContents.typeSelect(uml::Actor) AS actor»
      «EXPAND HomeIncludes FOREACH actor.getRelationships()
         .relatedElement»
    «ENDFOREACH»
          <ui:define name="body">
              Home
          </ui:define>
```

```
    </ui:composition>
  </body>
</html>
  «ENDFILE»
«ENDDEFINE»
```

Listing 7.1: Erzeugung der Datei home.xhtml

Da ähnlich dem Backend-Generator ein Textfile mit dem Generierungsresultat erzeugt wird, protokolliert das Template das Anlegen der Startseite mittels 'home.xhtml anlegen'. Danach wird die eigentliche Seite angelegt, wobei nach der Definition der Namespaces der Inhalt der Datei implementiert wird. Darin befinden sich weitere Aufrufe von Templates und Facelet-Tags, die beispielsweise Definitionsblöcke anlegen und inkludierte Inhalte importieren.

Für die Erzeugung der einzelnen Formulare werden Ein- und Ausgabe-Pins für die Definition der Inhalte verwendet. Listing 7.2 zeigt die Templates für die Generierung der Felder für die Ausgabe-Pins.

```
«DEFINE WebOutputPin FOR uml::OutputPin»
  «log("Output found")»
  «EXPAND WebOperationCall FOREACH outgoing.target»

    «FOREACH incoming.source AS incNode»
            <h:inputText value="#{«incNode.name».«name»}"
               title="«name»" />
    «ENDFOREACH»
«ENDDEFINE»


«DEFINE WebOperationCall FOR uml::CallOperationAction»
    <h:commandButton action="#{«EXPAND ActionCallName FOR
       this»}" value="«incoming.source.first().name»" />
«ENDDEFINE»
```

Listing 7.2: Templates für die Ausgabe-Pins

Der Aufruf des Templates startet mit der Erzeugung einer Textausgabe, die mit der Funktion 'log()' durchgeführt wird. Als nächster Schritt wird jedes Ausgabe-Pin analysiert und für jeden ausgehenden Operationsaufruf ein Button für die Bestätigung erzeugt. Dies geschieht in einem weiteren 'DEFINE'-Block, der in

der unteren Hälfte der Listing 7.2 zu sehen ist. Nach der Generierung der Buttons wird für jede eingehende Verbindung Textfelder erzeugt, welche die späteren Ausgabedaten des Formulars beinhalten.

## 7.6. Ergebnis der Generierung

Wie bereits erwähnt, wird mithilfe des Generators eine Benutzeroberfläche erstellt, die die wichtigsten Elemente für die Bedienung beinhaltet. Listing 7.3 zeigt die einen kurzen Ausschnitt aus der Ausgabe des xHTML-Files 'login.xhtml'. Zu erkennen ist das Formular, das unter den Definierten Umständen gerendert, also dargestellt, wird. In diesem Fall ist die Bedingung 'identity.loggedIn', welche ermittelt, ob der Benutzer im System angemeldet ist oder nicht. Wird das Formular schließlich erstellt, werden zwei JSF-Komponenten erzeugt; die Ausgabe des Namens des eingeloggten Benutzers und ein Button, der den Benutzer das Ausloggen aus dem System ermöglicht.

```
<h:form rendered="#{identity.loggedIn}">
    <h:outputText value="#{currentUser.firstName}"/>
    <h:commandButton action="#{identity.logout}" value="
       ausloggen"/>
</h:form>
```

Listing 7.3: Code einer Skelett-xHTML-Datei

Da durch den Generator keinerlei visuellen Elemente vorgesehen sind und die typischen HTML-Tags wie Tabellen oder Stylesheet-Klassen fehlen, erscheint die generierte Benuteroberfläche ohne Farben und ohne eine spezielle Anordnung der Elemente. Dennoch beinhaltet sie sämtliche, für die Funktion der Applikation relevanten JSF-Komponenten, die gegebenenfalls über eine graphisches HTML-Werkzeug versetzt und angepasst werden können. Dadurch ist die Trennung zwischen Programmierer und Designer der Webseite gegeben - der Designer kann sich auf die Darstellung konzentrieren und muss nur die bereits erstellten JSF-Inhalte positionieren. Abbildung 7.4 zeigt die Benuteroberfläche nach der Generierung.

Durch das Hinzufügen von HTML-Tags und Klassenspezifikationen ist es möglich, mit einfachen Mitteln das Layout und die Darstellung der Webseite anzupassen. Wie ein bearbeitetes xHTML-Dokument aussehen kann, zeigt Listing 7.4. Hier wurden die hinzugefügten HTML-Elemente roter und die durch die Generierung entstandenen Komponenten in blauer Farbe markiert.


Abbildung 7.4.: Abbildung Ausschnitt aus der Skelett-Benutzteroberfläche

```
<h:form rendered="#{identity.loggedIn}">
    <dl>
        <dt class="menuHeader">Welcome, <h:outputText value
           ="#{currentUser.firstName}"/></dt>
        <dd class="menuForm">
            <dl>
                <dd>Thank you for choosing the DVD Store</
                   dd>
                <dd>
                    <h:commandButton action="#{identity.
                       logout}" value="Logout"
                                     styleClass="formButton
                                        " style="width: 166
                                        px;"/>
                </dd>
            </dl>
        </dd>
    </dl>
</h:form>
```

Listing 7.4: Sourcecode der fertigen login.xhtml-Datei

Was bei der Betrachtung des Sourcecode-Stückes auffällt, ist die Tatsache, dass die JSF-Komponenten übernommen und lediglich um darstellungsspezifische Informationen erweitert wurden. Hierzu zählen Style-Klassen für die Anpassung mittels CSS-Stylesheets oder die Größe des Buttons. Das fertige Resultat der Anpassung ist in Abbildung 7.5 zu sehen.

Das Ergebnis ist eine vollständig verwendbare Benutzeroberfläche, deren Basiselemente generiert und mittels einfacher Mittel erweitert wurde. Der nächste Abschnitt 7.7 beschäftigt sich mit möglichen Erweiterungspotentialen des Frontend-Generators.

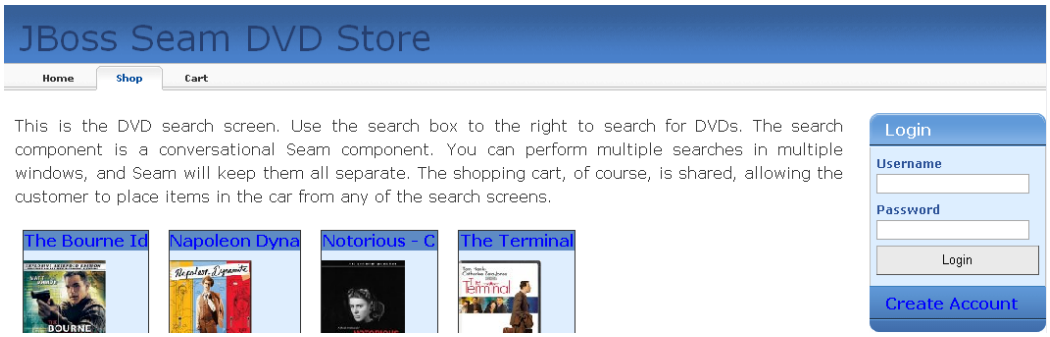


Abbildung 7.5.: Ausschnitt aus dem fertigen Frontend des DVD-Stores

## 7.7. Erweiterungsmöglichkeiten

Dass der Frontend-Generator bereits brauchbare Ergebnisse liefert wurde in Abschnitt 7.5 demonstriert. Da es sich hierbei jedoch um einen Prototypen zur Erstellung der Frontend-Aspekte handelt, gibt es noch viele Möglichkeiten das Ergebnis zu verbessern und um weitere Komponenten zu erweitern. Das Hauptproblem bei der Implementierung der Benutzeroberfläche ist jedoch das bereits in Abschnitt 7.4 bechriebene Problem, dass es derzeit kein standardisiertes UML-Diagramm für die Definition der Webseiten gibt und somit die Informationen aus der Verhaltensmodellierung herangezogen werden müssen. Eine eigene Diagrammart könnte fehlende Darstellungsinformationen beinhalten und Seitenflüsse besser definieren und somit die Arbeit des Frontend-Generators entscheidend verbessern.

Folgende Aufzählung gibt einen kurzen Überblick über mögliche Erweiterungspotentiale des Frontend-Generators:

- **Automatische Datenzugriffsseiten:** Derzeit wird die graphische Benutzeroberfläche ausschließlich auf Basis der eingegebenen Verhaltensmuster generiert. Um dem Entwickler Bearbeitung der Daten und Entitäten zu erleichtern, wäre es möglich, automatisch für jede Entität eine eigene Zugriffsseite anzulegen, die sämtliche Datenbankinhalte der Entität tabellarisch anzeigt und bearbeiten lässt. Somit wäre das Eingeben von Testdaten oder das Adaptieren in andere Bereiche erleichtert.

- **Bessere Prozessintegration:** Die Einbindung von Prozessen und Steuerungsmöglichkeiten eines Prozessmanagement-Systems ist derzeit noch nicht über den Seam-Frontend-Generator möglich. In der aktuellen Version muss der Modellierer über zukünftige Prozesse bescheid wissen und diese

Bereiche in der Verhaltensmodellierung auslassen, damit es nicht zu späteren Überschneidungen kommt. Eine Möglichkeit der Verbesserung wäre somit das Erfassen von Prozessen in Verbindung mit der Modellierung der Verhalten. Dies könnte das Abstimmen von Benutzeroberfläche und Prozesssteuerung erleichtern und den UML-Modellierer vom Prozessdenken lösen.

- **Standardlayouts:** Die Skelett-Oberfläche ist zwar losgelöst von HTML und Visualisierungselementen, könnte jedoch in Bezug auf die Erstellung von fertigen und benutzbaren Prototypen um Standard-Layouts erweitert werden. Neben Standard-Stylesheets könne auch ein Layout erzeugt werden, das den im Internet üblichem Schema entspricht und bereits einen besseren Überblick über die generierte Benutzeroberfläche geben. Dies würde die Möglichkeit bieten, schnelle und ansprechende Benutzeroberflächen mit dem Seam-Frontend-Generator zu erstellen, die nicht nur Funktional, sondern auch Optisch ansprechend sind.

- **Trennen der Templates:** Ähnlich dem in Kapitel 6 vorgestellten Backend-Generator befinden sich viele Templates noch zentral gelegen und sind nicht modular aufeinander abgestimmt. Durch eine Trennung der Web-Templates können effizientere JSF-Komponenten erzeugt werden, deren Einbindung in die xHTML-Seite weniger starr ist. Als weiterer Vorteil für diese Trennung wären die einfachere Wartung und die fehlenden Redundanzen in den Templates zu verzeichnen.

Da die Generierung von Benutzeroberflächen noch in den Kinderschuhen steckt, ist in naher Zukunft mit einer Fülle von neuen Möglichkeiten zu rechnen. In zukünftigen Versionen wären noch viele Verbesserungsmöglichkeiten denkbar, die an dieser Stelle den Rahmen sprengen würden. Eine grobe Auflistung einiger Erweiterungen wurde bereits geliefert, wobei ein neues UML-Diagramm für Web-Oberflächen die wohl beste Lösung darstellen würde.

# 8. Begriffe des Qualitätsmanagements

Das Testen von Software nimmt in der Entwicklung einen hohen Stellenwert ein. Gerade aus diesem Grund ist es wichtig, sich laufend mit Tests und möglichen Verbesserungen auseinanderzusetzen. Durch die agilen Entwicklungsmöglichkeiten der letzten Jahre ist es im Bereich der Qualitätssicherung zu einem Umdenken gekommen. Das Testen rückt dadurch immer mehr in den Vordergrund und begleitet die Software während ihrer Entwicklung, wodurch ein kontinuierlicher Verbesserungsprozess angestrebt wird. Die Testautomatisierung liefert ein recht dynamisches Hilfsmittel für die laufende Überprüfung des Systems. [Mat03] nennt zwei Hauptgründe, warum die Bereitstellung von Tests und Testcode für die Entwicklung wichtig ist:

- Sie unterstützen und validieren die laufende Entwicklung der Software.
- Sie bieten ein Framework für die langfristige Wartung und Evolution der Anwendung an.

Nach [Min04] haben agile Entwicklungprozesse jedoch mit dem wechselnden und unstabilen Bedingungen zu kämpfen, was die Tests und das Qualitätsmanagement stark beeinflusst. Daher wird sich dieser Teil der Arbeit auf die Verwendung des bereits in Kapitel 2.4 vorgestellten iterativen Entwicklungsprozesses anlehnen.

Die Definition von Tests lässt sich parallel zur eigentlichen Entwicklung durchführen, sodass der erzeugte Code stets über Testmethoden verfügt, die an die neuen Gegebenheiten angepasst sind. Abbildung 8.1 zeigt, wie die nebenläufige Generierung von Tests. [Lin07] zeigt, wie die herkömmliche, manuelle Erstellung von Tests mittels dem modellgetriebenen Ansatz automatisiert werden kann. Näheres zu diesem modellgetriebenen Konzept wird in Abschnitt 8.1 behandelt.

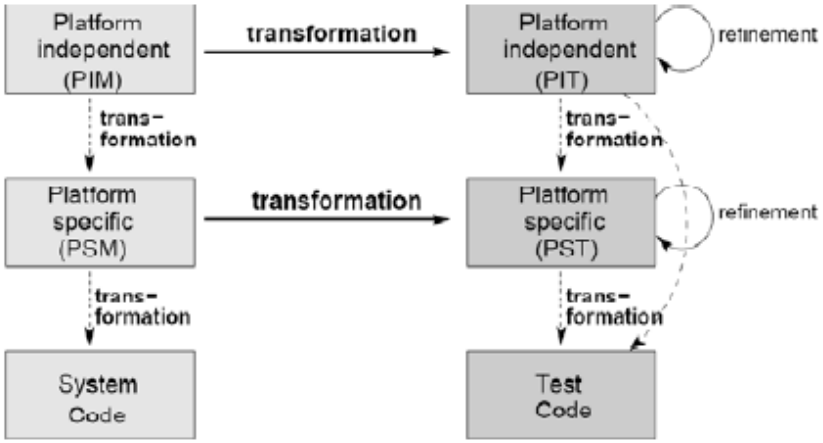


Abbildung 8.1.: Parallele Testentwicklung [Lin07]

Das Hauptaugenmerk dieses Abschnitts liegt in der kompakten Auflistung von Möglichkeiten zur modellgetriebenen Softwarequalitätssicherung. Dabei sollen Konzepte vorgestellt werden, die dabei helfen sollen, zukünftige Tests aus den Modellen abzuleiten. Als Referenz hierzu dient der bereits generierte Seam DVD-Store.

## 8.1. Modellgetriebenes Testen

Dieser Abschnitt beschäftigt sich mit der Möglichkeit, den Testablauf modellgetrieben zu gestalten. Das modellgetriebene Testen versucht Informationen aus den Modellen abzuleiten, die die Tests vorantreiben und bei deren Erzeugung helfen.

[Tho07] stellt für die Erstellung und Definition der Tests ein Modell zur Verfügung, welches in Abbildung 8.2 gezeigt wird. Hierbei werden die einzelnen Tests zu Gruppen zusammengefasst und auf dem Testling ausgeführt. Dieser wurde bereits mithilfe von Generatoren und manuellen Implementierungen erzeugt und stellt den aktuellen Stand der zu testenden Applikation dar. Die Ergebnisse der Tests wandern sowohl in die Entwicklung der Applikation, also in die Generatoren und die manuellen Bereiche, als auch in die Testartefakte.

Für die Definition der Testartefakte stellt openArchitectureWare ein Metamodell zur Verfügung, das die Tests erzeugen und über Java-Schnittstellen durch externe Programme ausführen lässt. Dieses Metamodell ist in Abbildung 8.3 dargestellt.

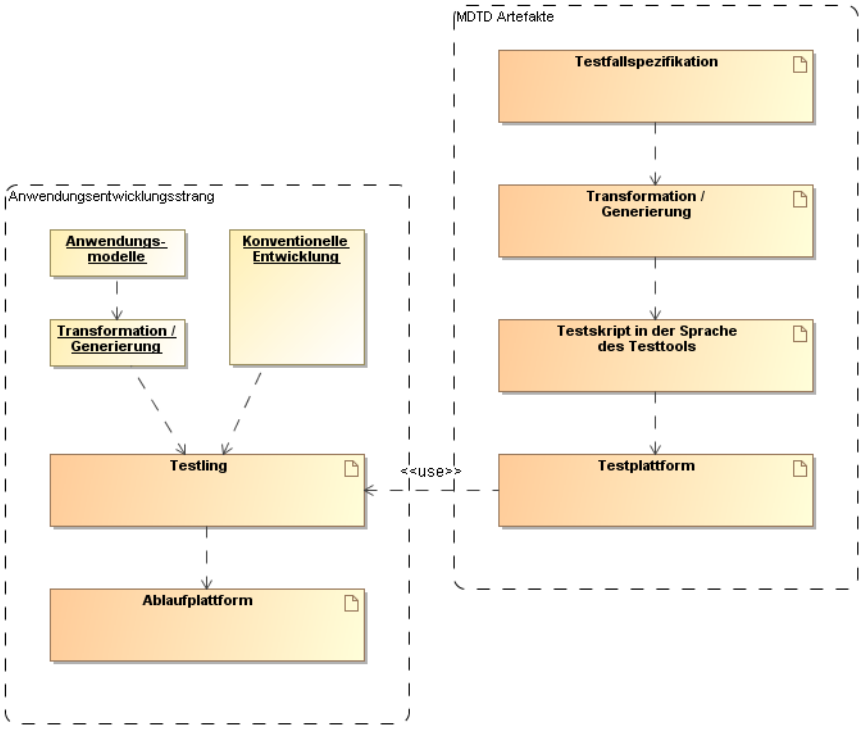


Abbildung 8.2.: Modellgetriebenes Test-Grundmodell

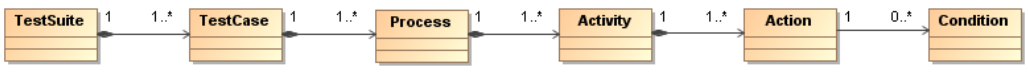


Abbildung 8.3.: Metamodell von openArchtectureWare-Tests

Die einzelnen Elemente eines oaw-Tests sind nach [Tho07]:

- **TestSuite:** Eine TestSuite definiert einen Ablauf von Testfällen, sich als geschlossene Gruppe ausführen lassen. Somit sind Handlungsstränge testbar, die komplexere Benutzerinteraktionen simulieren.

- **TestCase:** Ein Testfall ist die Definition von zu testenden Sachverhalten. Der Begriff wurde bereits in Abschnitt 8.2 behandelt.

- **Process:** Mittels Prozessen werden die einzelnen Tests ausgeführt. Dies erlaubt eine Vielzahl von Einstellungsmöglichkeiten wie beispielsweise die Anzahl von parallelen Prozessen oder Wartezeiten.

- **Activity:** Eine Aktivität beinhaltet eine Menge von Aktionen, also einer Abfolge von Handlungsdefinitionen.

- **Action:** Die Aktion ist die Ausführung einer Testdefinition und startet somit den Test. Sie definiert welche Informationen an den Testling gesendet werden.

- **Condition:** Der Soll-Zustand nach der Ausführung des Tests wird über die Kondition definiert. Ein Vergleich von Soll- und IstWert liefert die Ergebnisse über das Testverhalten.

Nähere Informationen zu diesem Thema und Beispiele in der Anwendung finden sich in der Literatur [Lin07] und [Tho07].

## 8.2. Tests aus den Klassendiagrammen

Durch die Generierung der Backend-Bereiche aus den UML Diagrammen liegt es nahe, dass diese auch Modelle für Tests der Entitäten und der Datenerhaltungsschicht herangezogen werden können. Für das Testen der Klassen und Beans haben sich sogenannte Unit-Tests bewährt, die Testfälle durchlaufen und schnell Resultate liefern. Nach [Wol01] ist hierfür Analysemodelle und Informationen über die Datenstrukturen notwendig, wodurch sich die Anlehnung an die Klassendiagramme ergibt.

*'Ein Testfall ist ein Stellvertreter einer realen Betriebssituation' [Pet98]*

Für die Java Umgebung hat sich ein kleines Hilfsframework namens JUnit etabliert. Dabei besteht ein Test aus der Testspezifikation, dem Aufrufen der zu testenden Klasse und dem Vergleich der Soll- und Istwerte. Ein Vorteil nach [Rei03] ist die einfache Erstellung von Testsuiten, die mehrere Tests zusammenfassen und sich auf wiederkehrende Problemstellungen anwenden lassen. Somit lässt sich eine Testsuite sehr vielseitig verwenden.

Aus den Klassendiagrammen der Java Beans lassen sich die für die Tests benötigten Daten gewinnen. Am Beispiel der Entität 'User' 4.1 lässt sich ein Testfall erstellen, wie er in Listing 8.1 dargestellt ist. Als Sprache für die Definition der Testfälle eignet sich XText, da diese ein Teil der openArchitectureWare ist und eine sehr einfachen Syntax aufweist. Nähere Informationen zu XText sind in der Literatur [OAW07] zu finden.

```
Testpassword(
    User {
        id:= 1
        username:= testuser
        password:= test
        lastname:= mustermann
        firstname:= markus
    }) ergibt test
```

Listing 8.1: Beispiel eines Testfalls

Das Beispiel testet den Wert des eingegebenen Passwortes. Neben einfachen Abfragen sind natürlich auch komplexere Testfälle möglich, die Methoden oder Verknüpfungen überprüfen. Eine Automatisation ist mit einfachen Mitteln möglich und würde sich in eine modellgetriebene Entwicklung integrieren lassen.

Die Testdaten für die automatisch generierten Testfälle würde man aus einer Datenbank oder separat abgelegten Datenfile beziehen. Dies trennt die Daten von den Implementierungen der Testfälle und erlaubt es, die Daten zentral zu verwalten.

Unter Verwendung einer externen Datenquelle in Form einer Datenbank und der exakten Abbildung der Testfälle mittels der Modelle, würden sich sämtliche Attribute der Klassen automatisch überprüfen lassen. Ein Automatisationsgrad von etwa 90 Prozent wären denkbar, wobei die restlichen Tests für Sonderfälle und spezielle Methoden manuell implementiert werden müssten.

## 8.3. JBoss Seam Tests aus Modellen

Das Framework JBoss Seam verfügt über eine eigene Definitionssprache für Tests. Listing 8.2 zeigt eine Testumgebung, die überprüft, ob sich ein Produkt ohne Attribute in das System eingeben lässt. Aufgrund der von der Klasse geforderten Definitionen sollte dies nicht möglich sein.

```
public class ProductUnitTest extends SeamTest
{
    EntityManager em() {
        EntityManagerFactory emf = Persistence.
           createEntityManagerFactory("dvdDatabase");
        EntityManager        em  = emf.createEntityManager
           ();
        assertNotNull("entity manager", em);
        assertTrue("entity manager open", em.isOpen());
        return em;
    }

    @Test
    public void testRequiredAttributes()
        throws Exception
    {
        Product p = new Product();

        EntityManager em = em();
        try {
            em.persist(p);
            fail("Leeres Produkt wurde persistiert!");
        } catch (PersistenceException e) {
            // Test bestanden
        } finally {
            em.close();
        }
    }
}
```

Listing 8.2: JBoss Seam Test

Als eigentlicher Test für die Datenschicht wird ein neues Produkt-Objekt erzeugt und versucht, dieses über den 'Entitymanager' zu persistieren. Die

Ausführung dieser Speicherung muss korrekt abgefangen werden, was im Beispiel mit dem Abfangen der 'PersistenceException' erfolgt.

Diese Möglichkeit der Erzeugung von Testabläufen würde sich ebenfalls aus den Klassendiagrammen ableiten lassen. Hierzu müsste man für jede Entität ein Test generiert werden, dessen Ausführung darüber Aufschluss gibt, ob die Entität korrekt vom 'Entitymanager' behandelt wird.

Neben der Datenspeicherung lassen sich auch komplexere Systemtests anlegen, die Methoden oder ganze Prozesse überprüfen. Diese würden sich nur schwer aus den Klassendiagrammen herauslesen lassen, da diese keinerlei Informationen über die Verwendung des Benutzers geben. Durch die Verwendung der Verhaltensmodellierung würden sich komplexere Szenarios erstellen lassen, da diese genügend Informationen über die Prozessabläufe bieten.

# 9. Zusammenfassung und Ausblick

Dieses Kapitel beschäftigt sich mit der Zusammenfassung der wichtigsten Aspekte der Arbeit. Zunächst wird die eigentliche Generatorenentwicklung und der Generationsgrad des Generators näher betrachtet. Zuletzt wird noch ein Ausblick auf mögliche Themen weitere Untersuchungen gegeben.

## 9.1. Zusammenfassung

Im Zuge dieser Arbeit wurde mithilfe der modellgetriebenen Entwicklung ein Generator entwickelt, der die Implementierung einer Webapplikation auf Basis von JBoss Seam [JBo07f] vereinfacht. Der entstandene Generator versucht die Vorteile des MDSD mit dem JBoss Seam Framework bestmöglich zu kombinieren.

Zu Beginn wurden die Anforderungen an die Referenzimplementierung anhand einer simplen HelloWorld-Implementierung abgeleitet. Mithilfe des Sourcecodes dieser einfachen Testanwendung wurden die ersten Templates für den Generator entwickelt. Nach dieser kurzen Einlernphase folgte die eigentliche Entstehung des Generators.

Die finale Entwicklung des Generators wurde mithilfe des zweigleisigen, iterativen Entwicklungsprozesses (siehe dazu 2.4) durchgeführt, der die Implementierung der Infrastruktur von der Anwendungsentwicklung trennt, sowie den Generator und die Templates iterativ verbessert. Die Arbeit mit diesem Prozess hatte den Vorteil, den Generator exakt an die Anforderungen der Referenz anzupassen und die Modelle unabhängig vom Generator zu modellieren. Als Referenzimplementierung für diese Arbeit diente ein DVD-Store, der für die Generatorentwicklung und das Referenzmodell herangezogen wurde. Während der Entwicklung wurden die Modelle und die Generatoren ständig verbessert, was sich auf die Qualität und den Automatisierungsgrad des *JBoss-Seam-Generators* positiv auswirkte.

Für die Entwicklung des Generators wurden die Backend- und Frontend-Bereiche voneinander getrennt. Dadurch ist es möglich, diese separat zu generieren und getrennt voneinander einzusetzen. Somit lassen sich beide Bereiche je nach Bedarf generieren, warten und weiterentwickeln.

Der Automatisierungsgrad des Generators gibt an, wieviele Zeilen Sourcecode im Verhältnis zur manuellen Implementierung automatisch generiert wurden. Um diesen Grad zu ermitteln, wurden zunächst die Generierungsergebnisse analysiert. Hierzu wurde das Zählprogramm *LOCC* [LOC04] eingesetzt, das den Java-Code durchsucht und Informationen zur Anzahl der Zeilen liefert. Einen Ausschnitt der Ergebnise liefert Abbildung 9.1. Sie zeigt die Kurzzusammenfassung der Ergebnisse, wobei diese sich aus den ausgewerteten Resultaten der Zählung der Sourcecodezeilen der Klassen und Methoden zusammensetzen. Das vollständige Ergebnis befindet sich im Anhang auf der dieser Arbeit beigelegten CD. Als nächster Schritt wurde das Generat um die manuelle Fachlogik erweitert, woraufhin eine erneute Zählung der Codezeilen folgte. Dieses Ergebniss ist ebenfalls auf der CD zu finden. Nachdem beide Werte verfügbar waren, ließ sich ein Automatisierungsgrad von 81,6 Prozent für den Backend-Generator ermitteln.

| | A | B |
|---|---|---|
| 1 | Summary Information | LOC/File |
| 2 | Totals | 1375 |
| 3 | Average | 42,96875 |
| 4 | Max | 142 |
| 5 | Min | 8 |
| 6 | | |

Abbildung 9.1.: Ausschnitt aus der Analyse der generierten Java-Klassen

Für den Frontend-Generator wurde dieser Grad der Automatisierung nicht ermittelt, da sich Design und Sourcecode nicht völlig voneinander lösen lassen. Es wurde durch den Generator ein Grundgerüst erstellt, welches als Blackbox-Prototyp dient oder zur Weiterbearbeitung mit einem Designwerkzeug genutzt werden kann. Eine Anzahl der Sourcecodezeilen stellt keine sinnvolle Meßgröße in diesem Bereich dar und wurde daher nicht ermittelt.

## 9.2. Ausblick

Obwohl es viele Werkzeuge und Standards in der modellgetriebenen Entwicklung gibt, ist es nicht einfach, eine passende Kombination von Werkzeugen zu finden. Der Markt wird regelrecht überschwemmt von verschiedenen Modellie-

rungssprachen, Transformationssprachen und Entwicklungsparadigmen, die oft nicht halten was versprochen wird. In dieser Arbeit wurde UML verwendet, da sie viele Möglichkeiten bietet und die Diagramme eine breite Anwendung seitens der Industrie finden, was sich durch eine gute Werkzeugunterstützung zeigt. Jedoch ist diese Wahl für diese Problemstellung nicht optimal, da sich mit dieser nicht sämtliche Aspekte der Webanwendung modellieren lassen.

In dieser Arbeit konnten nicht sämtliche Themen im Dunstkreis von MDA behandelt werden, da dies sonst den Rahmen der Arbeit gesprengt hätte. Im Laufe der Entwicklung und Implementierung sind viele weitere Aspekte aufgetreten, die Ausgangspunkte für weitere Untersuchungen sein könnten:

- **Entwicklungsprozess:** Obwohl der zweigleisige, iterative Entwicklungsprozess sehr gute Ergebnisse in der Generatorentwicklung liefert, können leicht Schwierigkeiten in der iterativen Verwendung der Feedback-Informationen auftreten. Durch die klare Trennung von Generator und Anwendung kann es dazu kommen, dass beide Bereiche unterschiedliche Verbesserungsinformationen aus dem Feedback gewinnen und diese in der aktuellen Iteration anwenden. Somit kommt es zur Erweiterung beider Problembereiche und es kann dadurch zu zusätzlichen Problemen führen. Ein Vergleich von unterschiedlichen Entwicklungsprozessen für eine modellgetriebene Entwicklung wäre durchaus interessant und würde in Zukunft vielleich eine bessere Steuerung der Generatorenentwicklung zulassen.

- **Benutzeroberfläche:** UML bietet derzeit kein Diagramm um die Benutzeroberfläche zu modellieren. Eine Evaluierung der Zusammenarbeit von MDA-Werkzeugen mit Webdesign-Tools wäre der nächste logische Schritt in der modellgetriebenen Entwicklung von Benutzeroberflächen.

- **Trennung der Source-Webinhalte:** MDA sieht eine iterative Verbesserung des erzeugten Sourcecodes vor und die gängigsten Programmiersprachen bieten dazu eine Vielzahl von Möglichkeiten, generierten von manuellen Code zu trennen und separat zu verbessern. Dies ist in HTML-Code nur schwer möglich. In möglichen Untersuchungen gilt es herauszufinden, wie man den Sourcecode der Benutzeroberfläche für die modellgetriebene, ständige Weiterentwicklung modular trennt. Dies würde eine kontinuierliche Verbesserung zulassen, ohne die manuellen Fragmente innerhalb der HTML oder JSP-Datein zu beeinträchtigen.

- **Cartridge-Entwicklung:** Derzeit gibt es keinen Standard in der Definition der

Templates der Generatoren. Somit ist es jedem selbst überlassen, für eine sinnvolle Trennung der Templates in verschiedene Bereiche zu sorgen. Ein Standard oder ein Konzept für die Implementierung der Templates würde dabei helfen, die Struktur von externen Templates besser zu verstehen und in eigene Arbeiten zu integrieren. Statt einzelner Insellösungen sollte es in Anlehnung an Web-Services [W3C07] möglich sein, andere Templates dynamisch einzubinden. Hierzu wäre auch eine Standardbeschreibung der Templates oder der Cartridge ähnlich der WSDL bei Web-Services [W3C01] sinnvoll. Untersuchungen bezüglich einer besseren Vernetzung von modellgetriebenen Entwicklungen könnten den Austausch und die Anbindung von Cardridges erleichtern.

- **Prozessdefinition:** Die Definition von Prozessen mittels UML ist zwar möglich, wurde innerhalb dieser Arbeit jedoch nicht behandelt. Für eine bessere Prozessunterstützung mittels JBoss jBPM [JBo07d] wäre es jedoch sinnvoll, diese Diagramme vom *JBoss jBPM Graphical Process Designer* [JBo07e] zu lösen und in die bereits bestehenden UML Diagramme zu integrieren. Dadurch könnten die Prozessinformationen und möglicherweise die Konfiguration der Prozesse ebenfalls generiert werden. Dazu wären jedoch weitere Untersuchungen nötig.

# Listings

# Abbildungsverzeichnis

# Literaturverzeichnis


[Ald01] Aldo Bongio, Stefano Ceri, Piero Fraternali, Andrea Maurino. Modeling Data Entry and Operations in WebML. In *Selected papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases*, pages 201–214, London, UK, 2001. Springer-Verlag.

[Apa07] Apache Ant. http://ant.apache.org/, [Online; accessed 07-Dez-2007], 2007.

[Chr05] Chris Schalk. Introduction to Javaserver Faces. http://www.oracle.com/technology/tech/java/newsletter/articles/introjsf/index.html, April 2005.

[Dou07] Doug Rosenberg, Matt Stephens. *Use Case Driven Object Modeling with UML: Theory and Practice.* Apress, 2007.

[Ecl07] Eclipse Foundation. http://www.eclipse.org/, [Online; accessed 04-Dez-2007], 2007.

[Ger04a] Gerd Beneken, Florian Deißenböck. Inside JavaServer Faces. *JavaSPEKTRUM, Ausgabe 1*, April 2004.

[Ger04b] Gerhard Wohlgenannt. Auswirkungen des Einsatzes von Application Frameworks in der Entwicklung von Web-Software mit PHP. Master's thesis, Wirtschaftsuniversität Wien, 2004.

[InO07] Interactive Objects - ArcStyler. http://www.arcstyler.com/, [Online; accessed 04-Dez-2007], 2007.

[JBo07a] JBoss - a division of Red Hat. http://www.jboss.com/, [Online; accessed 04-Dez-2007], 2007.

[JBo07b] JBoss Application Server. http://www.jboss.org/products/jbossas, [Online; accessed 07-Dez-2007], 2007.

[JBo07c] JBoss Hibernate. http://www.hibernate.org/, [Online; accessed 04-Dez-2007], 2007.

[JBo07d] JBoss jBPM. http://www.jboss.com/products/jbpm/, [Online; accessed 04-Dez-2007], 2007.

[JBo07e] JBoss jBPM Graphical Process Designer. http://docs.jboss.com/jbpm/v3/gpd/, [Online; accessed 04-Dez-2007], 2007.

[JBo07f] JBoss Seam. http://www.jboss.com/products/seam/, [Online; accessed 04-Dez-2007], 2007.

[Jia03] Jianguo Lu, John Mylopoulos. Automated EJB Client Code Generation sing Database Query rewriting. In *Proceedings of the Seventh International Database Engineering and Applications Symposium*, Hong Kong, SAR, April 2003.

[JSP07] JSP - Tutorial. http://www.JSPtutorial.org, [Online; accessed 04-Dez-2007], 2007.

[Lin07] Linard Moll. Testing-Möglichkeiten für MDD/MDA. http://seal.ifi.uzh.ch/fileadmin/User_Filemount/Vorlesungs_Folien/Seminar_SE/SS07/SemSE07-Linard_Moll.pdf, 2007.

[LOC04] LOCC User's Guide. http://csdl.ics.hawaii.edu/Tools/LOCC/dist/doc/userguide/userguide.html, [Online; accessed 04-Dez-2007], 2004.

[Mag07] No Magic MagicDraw. http://www.magicdraw.com/, [Online; accessed 04-Dez-2007], 2007.

[Mar01] Mark Wutka. *Special Edition Using Java 2 Enterprise Edition (J2EE): With JSP, Servlets, EJB 2.0, JNDI, JMS, JDBC, CORBA, XML and RMI.* QUE, 2001.

[Mar05a] Marco Kuhrmann, Dirk Niebuhr. Das V-Modell XT in der Praxis - IT-WiBe. In *Proceedings of the 12. Workshop der Fachgruppe WI-VM der Gesellschaft für Informatik e.V. (GI) zum Thema: Entscheidungsfall Vorgehensmodelle.* Shaker Verlag, 2005.

[Mar05b] Martin Hitz, Gerti Kappel, Elisabeth Kapsammer, Werner Retschitzegger. *UML @ Work.* Dpunkt Verlag, 2005.

[Mar06] Mario Prinz. Modellgetriebene Entwicklung ubiquitärer Web-Anwendungen - Evaluierung aktueller Ansätze und Werkzeuge. Master's thesis, Technische Universität Wien, 2006.

[Mat03] Matthew J. Rutherford, Alexander L. Wolf. A case for test-code generation in model-driven systems. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 377–396, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[Mic04] Michael Plöd. Winterschläfer Objektrelationales Mapping mit Hibernate. http://entwickler.de/zonen/portale/psecom,id,101,online,596,.html, Juli 2004.

[Min04] Ming Huo, June Verner, Liming Zhu, Muhammad Ali Babar. Software Quality and Agile Methods. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04) pp. 520-525*, Hong Kong, September 2004.

[MSD07] Distributed Component Object Model. http://msdn2.microsoft.com/en-us/library/ms809340.aspx, [Online; accessed 04-Dez-2007], 2007.

[Nor01] Nora Koch, Andreas Kraus, Rolf Hennicker. The Authoring Process of the UML-based Web Engineering Approach. In *Proceedings of the 1st International Workshop on Web-Oriented Software Technology*, Valencia, Spain, 2001.

[Nor02] Nora Koch, Andreas Kraus. The expressive Power of UML-based Web Engineering. In *Proceedings of the 2nd International Workshop on Web Oriented Software Technology, CYTED, pp. 105-119*, Málaga, Spain, 2002.

[OAW06] Workflow Engine Reference. http://www.eclipse.org/gmt/oaw/doc/4.1/r05_workflowReference.pdf, [Online; accessed 04-Dez-2007], 2006.

[OAW07] openArchitectureWare. http://www.openarchitectureware.org, [Online; accessed 04-Dez-2007], 2007.

[OMG07a] MetaObject Facility. http://www.omg.org/mof/, [Online; accessed

04-Dez-2007], 2007.

[OMG07b] The Object Management Group (OMG). http://www.omg.org/, [Online; accessed 04-Dez-2007], 2007.

[OMG07c] UML® 2.0 OCL Specification. http://www.omg.org/docs/ptc/03-10-14.pdf, [Online; accessed 04-Dez-2007], 2007.

[OMG07d] UML® 2.1.1 Specification. http://www.omg.org/technology/documents/formal/uml.htm, [Online; accessed 04-Dez-2007], 2007.

[Pet98] Peter Liggesmeyer, Martin Rothfelder, Michael Rettelbach, Thomas Ackermann. Qualitätssicherung Software-basierter technischer Systeme - Problembereiche und Lösungsansätze. *Informatik-Spektrum, Band 21, Heft 5, S. 249 - 258*, Oktober 1998.

[Pet03] Peter Roßbach, Thomas Stahl, Wolfgang Neuhaus. Model Driven Architecture - Grundlegende Konzepte und Einordnung der Model Driven Architecture (MDA). *Java Magazin, Ausgabe 9/2003*, April 2003.

[Rei03] Reiko Heckel, Marc Lohmann. Towards Model-Driven Testing. *Electronic Notes in Theoretical Computer Science 82 No. 6 (2003)*, Oktober 2003.

[Ric06] Richard Monson-Haefel, Bill Burke. *Enterprise JavaBeans 3.0.* O'Reilly, 2006.

[Rol06] Roland Petrasch, Oliver Meimberg. *Model-Driven Architecture. Eine praxisorientierte Einführung in die MDA.* Dpunkt Verlag, 2006.

[RUP07] Rational Unified Process. http://www-306.ibm.com/software/awdtools/rup/, [Online; accessed 04-Dez-2007], 2007.

[Ste02] Stephanie Bodoff, Dale Green, Kim Haase, Eric Jendrock, Monica Pawlan, Beth Stearns. *The J2EE Tutorial.* Prentice Hall, 2002.

[Sun05] JSF KickStart: A Simple JavaServer Faces Application. http://www.exadel.com/tutorial/jsf/jsftutorial-kickstart.html, [Online; accessed 04-Dez-2007], 2005.

[Sun07a] Sun Developer Network. http://java.sun.com/javaee/, [Online; accessed 04-Dez-2007], 2007.

[Sun07b] Sun Microsystem JSP. http://java.sun.com/products/jsp/, [Online; accessed 04-Dez-2007], 2007.

[Sun07c] Sun Microsystems JSF Projektseite. https://facelets.dev.java.net/, [Online; accessed 04-Dez-2007], 2007.

[Tho01] Thomas Neumann, Ulf Schreier, Martin Fabini. Auf dem Weg von EJB zu Fachkomponenten. *Objekt Spektrum, Ausgabe 1*, Jänner 2001.

[Tho07] Thomas Stahl, Markus Völter, Sven Efftinge. *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management.* Dpunkt Verlag, 2007.

[Vol06] Volker Gruhn, Daniel Pieper, Carsten Röttgers. *MDA. Effektives Softwareengineering mit UML2 und Eclipse: Effektives Software-Engineering mit UML2 und Eclipse.* Springer, Berlin, 2006.

[W3C01] Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl/, [Online; accessed 04-Dez-2007], 2001.

[W3C07] W3C - Web Services Activity. http://www.w3.org/2002/ws/, [Online; accessed 04-Dez-2007], 2007.

[Web07] The Web Modeling Language. http://www.webml.org/, [Online; accessed 04-Dez-2007], 2007.

[Wol01] Wolfgang Zuser, Stefan Biffl, Thomas Grechenig, Monika Köhle. *Software Engineering mit UML und dem Unified Process.* Pearson Studium, 2001.

# A. Benutzerdokumentation

Dieses Kapitel beschäftigt sich mit dem beiliegenen Datenträger. Es wird gezeigt, wie die benötigte Infrastruktur für die Ausführung des *JBoss Seam Generators* geschaffen wird. Abbildung A.1 zeigt den Inhalt der beiliegenden CD-Rom.

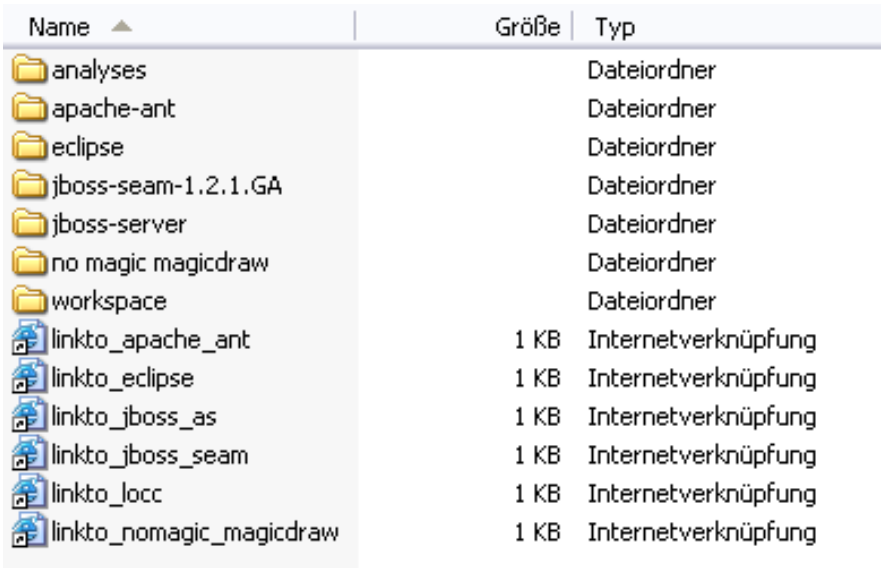


Abbildung A.1.: Inhalt der beiliegenden CD-Rom

Im Root-Verzeichnis der CD sind Internet-Verknüpfungen zu den wichtigsten Informationsseiten zu finden. Auf diesen Seiten können Informationen über die verschiedenen Distributionen, deren Verwendung und Installationshinweise gefunden werden. Desweiteren befinden sich die verwendeten Distributionen in den entsprechenden Ordnern. Je nach Bedarf können die aktuellsten Distributionen von den Internetseiten geladen oder die beiliegenden Versionen verwendet werden. Jedoch wurden die aktuellsten Distributionen nicht auf deren Kompatibiliät mit dem JBoss Seam Generator getestet.

Im folgenden Abschnitt wird die Installation der einzelnen Komponenten gezeigt und auf notwendige Einstellungen eingegangen.

## A.1. Apache Ant

Apache Ant [Apa07] wird dazu benötigt, das *Build-File* der JBoss Seam Distribution auszuführen, den Sourcecode automatisch zu kompilieren und die entstandenen Datein auf den Server zu deployen.

Um Apache Ant zu nutzen, sind innerhalb einer Windows-Umgebung folgende Schritte notwendig:

- Die Ant Distribution in einen beliebigen Ordner auf der Festplatte kopieren (z.B.: c:\ant)
- Die Umgebungsvariable *ANT_HOME* setzen
- Das *bin-Verzeichnis* in den *Pfad* aufnehmen

Je nach verwendetem Betriebssystem sind möglicherweise unterschiedliche Installationsmaßnahmen durchzuführen, wobei diese auf der Internetseite der Distribution zu finden sind.

## A.2. JBoss Application Server

Der JBoss Application Server [JBo07b] lässt sich über den beiliegenden *jems-installer-1.2.0.GA.jar* installieren. Dieser führt durch die wichtigsten Schritte der Installation. Wichtig ist, dass die Installation die Verwendung von *ejb* oder *ejb-clustered* unterstützt, um die JBoss Seam Anwendungen ausführen zu können. Abbildung A.2 zeigt das Installationsfenster der Routine, die durch die Installation des Servers leitet.

Alternativ können auch andere Versionen des Servers installiert werden, auf deren Installation an dieser Stelle nicht eingegangen wird.

## A.3. JBoss Seam

JBoss Seam [JBo07f] liegt in der Version 1.2.1 der CD bei. Für eine Installation muss die Distribution zunächst auf die Festplatte kopiert und die Datei *build.properties* auf die lokalen Einstellungen angepasst werden. Abbildung A.3

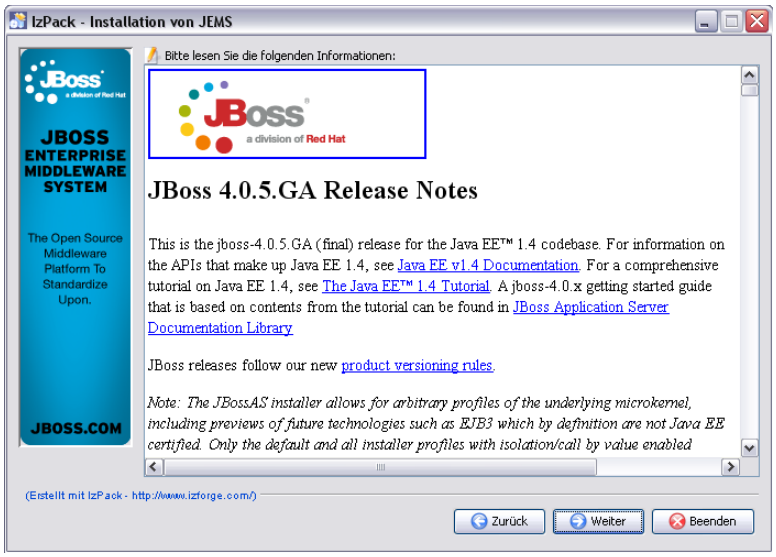


Abbildung A.2.: Installationsroutine des JBoss Application Servers

zeigt die zwei zu konfigurierenden Einträge.

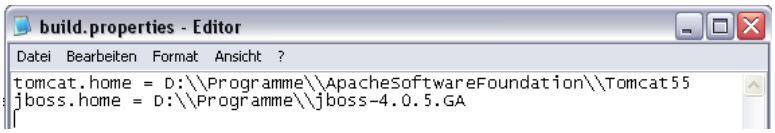


Abbildung A.3.: Konfiguration von JBoss Seam

Nach diesen Einstellungen muss die Datei *build.xml* ausgeführt werden, wobei Apache Ant dazu herangezogen wird. Mittels des Befehls *ant* innerhalb des entsprechenden Ordners wird diese Datei ausgeführt, und JBoss Seam auf dem System installiert.

## A.4. Eclipse

Eine für das Beispiel ausreichend konfigurierte Eclipse-Distribution ist in der Version 3.3 auf der CD zu finden. Die nötigen Plugins und Einstellungen für die Verwendung von openArchitectureWare sind bereits installiert und können sofort eingesetzt werden. Weitere Hinweise für die Installation oder andere Versionen von Eclipse finden sich auf der entsprechenden Internetseite [Ecl07].

## A.5. No Magic MagicDraw

Für die Modellierung der UML 2.0 Diagramme und die Verwendung des Profiles eignet sich die Verwendung eines Modellierungswerkzeuges. Obwohl eine Anpassung der Modelle ohne Probleme mit Eclipse möglich ist, wird ein externes Werkzeug wie MagicDraw [Mag07] empfohlen, welches für die graphische Erstellung der Diagramme und dem Export im XML-Format zulässt.

Eine Distribution in der Version 12.5 ist der beiliegenden CD beigefügt. Sie bedarf keiner Installation sondern lässt sich über die Datei *mduml.exe* im *bin*-Verzeichnis starten. Für eine Ausführung wird jedoch ein Evaluierungsschlüssel benötigt, der auf der Internetseite der Distribution erhältlich ist.

## A.6. Beispielausführung

Für die Ausführung des JBoss Seam Generators wurde der *Eclipse-Workspace* der CD beigefügt. Darin befinden alle benötigten Datein um den Generator auszuführen. Für einen Test des Generators kann das Backend der Referenzanwendung generiert werden, wobei folgende Schritte durchzuführen sind:

- Starten der Eclipse-Umgebung. Hierbei muss der Workspace der CD eingestellt werden, der zuvor auf eine lokale Festplatte kopiert werden muss, da ansonsten keine Daten generiert werden können.
- Projekt des Generators öffnen
- Starten der Generator-Workflowdatei *generate_backend.oaw* als oAW-Workflow (siehe Abbildung A.4)
- Kopieren der erstellten Java-Klassen, die sich nach der Ausführung des Generators im Ordner *src-gen* befinden, in den entsprechenden *src-Ordner* der Referenzanwendung innerhalb des JBoss Seam Ordners (siehe Abbildung A.5). Bereits darin befindliche Datein können überschrieben werden.
- Starten des JBoss Application Servers.
- Apache Ant innerhalb des Root-Verzeichnis ausführen, damit das entsprechende *build.xml* ausgeführt wird, welches für das Kompilieren und Deployen der Anwendung zuständig ist (siehe Abbildung A.6).

- Starten des Internetbrowsers und Eingabe der URL: http://localhost:8080/seam-dvd/

Nachdem die URL in den Webbrowser eingegeben wurde, wird die Anwendung gestartet.

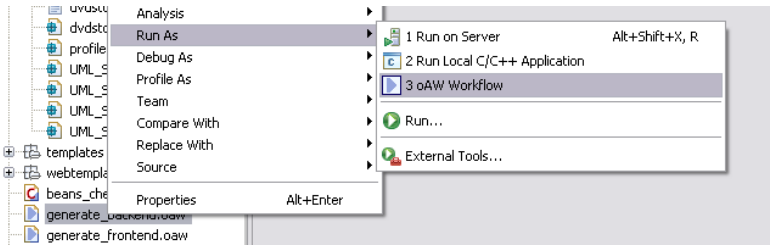


Abbildung A.4.: Starten des Generators

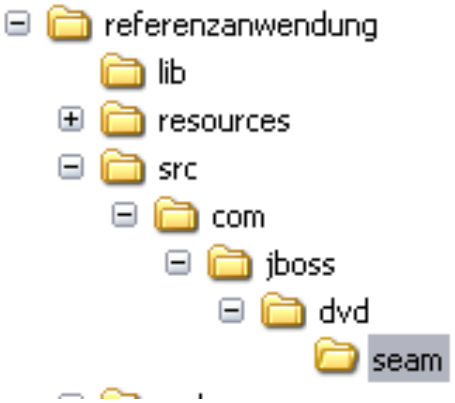


Abbildung A.5.: Ordner der Referenzanwendung

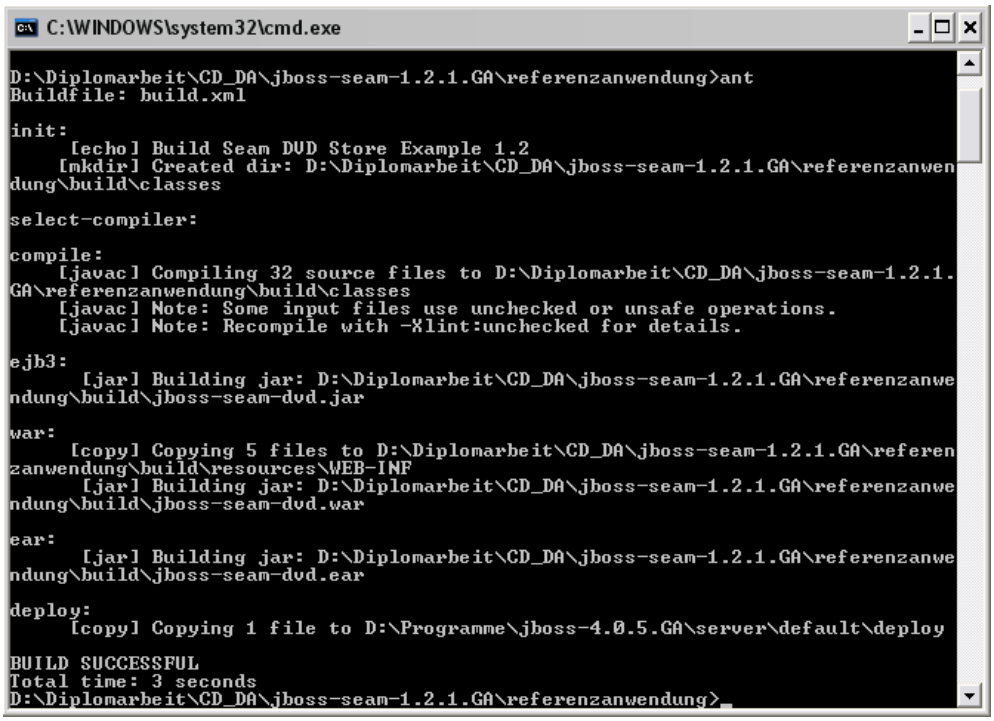


Abbildung A.6.: Kompilierung und Deployment der Referenzanwendung