

2 Einführung in MDSD

Dieses Kapitel erklärt in groben Zügen, was MDSD ist, und legt damit das Fundament für die folgenden Kapitel. Der erste Abschnitt definiert MDSD und diskutiert, welche Ziele mit einem entsprechenden Vorgehen erreicht werden können. Im letzten Abschnitt dieses Kapitels zeigen wir ein Beispiel, in dem es darum geht, mit Modellen und Generatoren eine dynamische Webanwendung zu erstellen. In dieser ersten Fallstudie werden wir dabei hauptsächlich die Sicht des Entwicklers und Modellierers einnehmen.

Die konkrete Definition bzw. Wiederverwendung von Modellierungssprachen sowie die Implementierung eines Generators ist Thema des zweiten Teils dieses Buches.

2.1 Was ist MDSD

Diesem Buch liegt die folgende Definition zu Grunde:

Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen.

Die Definition besteht aus drei Teilen, die wir im Folgenden näher betrachten.

2.1.1 Formale Modelle

Modelle tauchen in vielen Projekten in unterschiedlichen Erscheinungsformen auf – Architekturskizzen an Whiteboards, UML-Diagramme, die bestimmte Designaspekte dokumentieren etc. Diese unterschiedlichen Arten von Modellen haben ihre Berechtigung, aber nicht jedes von ihnen ist für MDSD geeignet – nicht jedes Modell ist *formal*.

»Formal« bedeutet hier, dass das Modell irgendeinen Aspekt der Software vollständig beschreibt. Das heißt nicht, dass das Modell alles beschreibt – das wäre illusorisch –, aber es muss klare Regeln geben, worüber das Modell Aussa-

gen macht und worüber nicht. Im Beispiel am Ende dieses Kapitels werden z.B. Entitäten und Komponenten mit entsprechenden Modellen beschrieben werden.

Formale Modelle sind übrigens nicht notwendigerweise UML-Modelle. Klassendiagramme sind weit verbreitet und allgemein bekannt, und deshalb verwenden wir sie speziell für die einführenden Beispiele. Viele Dinge lassen sich aber mit textuellen Modellen deutlich besser beschreiben, und im Laufe des Buches werden auch Beispiele für solche textuellen Modelle vorkommen. Hier am Anfang soll der Hinweis genügen, dass es auch Nicht-UML-Modelle gibt und man die konkrete Notation für ein Modell im Einzelfall selbst bestimmen und den spezifischen Bedürfnissen anpassen kann.

2.1.2 Lauffähige Software erzeugen

Das Endprodukt von MDSD ist lauffähige Software. Wenn ein formales Modell zur Dokumentation verwendet wird oder als Spezifikation für eine manuelle Implementierung dient, dann kann das Modell nützlich sein, das Ganze ist aber keine modellgetriebene Softwareentwicklung.

Es gibt zwei grundlegende Ansätze, aus Modellen ausführbare Software zu erzeugen: Generatoren und Interpreter. Der Begriff MDSD, wie wir ihn in diesem Buch verwenden, umfasst dabei beide Möglichkeiten. Vieles von dem, was wir beschreiben, gilt dabei für beide Ansätze. Anderes gilt ausschließlich für das Generieren, und wir haben uns bemüht, dies jeweils zu kennzeichnen.

Ein Generator ist ein Stück Software, das aus einem Modell Quelltext einer anderen Programmiersprache, z.B. C# oder Java, erzeugt. Es handelt sich also im Prinzip um einen Übersetzer. Die vielfältigen Möglichkeiten, einen Generator zu implementieren, werden in Kapitel 8 näher beschrieben. Generatoren sind – wie auch Compiler – Bestandteil des Build-Prozesses.

Ein Interpreter ist ein Stück Software, das ein Modell zur Laufzeit einliest und abhängig von seinem Inhalt verschiedene Aktionen ausführt. Kapitel 9 behandelt Interpreter im Detail.

Die entscheidende Gemeinsamkeit von beiden ist, dass sie dafür sorgen, dass Modelle vom Rechner ausgeführt werden können, also eine lauffähige Software entsteht.

An dieser Stelle merkt man in der Praxis noch einmal genau, ob die Bedeutung der Modelle wirklich klar definiert ist. Wenn man nämlich versucht, ein Modell »auszuführen«, dann stößt man in ganz natürlicher Weise auf Lücken und Mehrdeutigkeiten. Und wenn ein bestimmtes Feature im Modell noch nicht vorgesehen ist, dann stellt man das z.B. fest, wenn man einen Generator für dieses Feature schreiben will.

2.1.3 Automatisch

Der Schritt vom Modell zu ausführbarer Software soll außerdem automatisch erfolgen. Das bedeutet zunächst, dass das Modell nicht einfach nur als Spezifikation für manuelle Implementierung verwendet wird. Allgemeiner heißt es aber auch, dass man einen automatischen Build-Prozess im Sinne von *make* oder *ant* aufsetzen kann, in den die Modelle als Input einfließen und an dessen Ende ein lauffähiges Programm steht.

Man generiert also nicht im Stil eines IDE-Wizards einmalig aus einem Modell Quelltext, den man dann anschließend von Hand weiterentwickelt. So nützlich solche Wizards sind, sie führen dazu, dass man spätere Änderungen immer direkt am Quelltext durchführen muss – das Modell spielt keine Rolle mehr.

Der Kerngedanke ist dabei, dass die Modelle die Rolle der Quelltexte einnehmen und generierte Artefakte nur eine Art temporäre Build-Zwischenprodukte sind. Änderungen werden an den Modellen vorgenommen, so dass der generierte Code immer einheitlich ist und die Modelle automatisch den aktuellen Stand widerspiegeln.

Es sei an dieser Stelle darauf hingewiesen, dass Modelle natürlich im Allgemeinen nicht das gesamte System beschreiben. Das Gesamtsystem enthält sowohl generierte als auch manuell implementierte Anteile.

2.2 Gründe für MDSD

Es geht bei MDSD zunächst einmal um die Verbesserung von Softwarequalität und Wiederverwendbarkeit sowie die Steigerung der Entwicklungseffizienz. Dies bedeutet insbesondere, den Softwareentwickler von lästiger und fehleranfälliger Routinearbeit zu befreien.

Entwickler sind heutzutage sehr komplexen Software-Infrastrukturen ausgesetzt: Application-Server, Datenbanken, Open-Source-Frameworks, Protokollen, Schnittstellentechnologien etc., die alle in geeigneter Weise in Zusammenhang gebracht werden müssen, so dass letztlich eine performante, robuste und wartbare Software entsteht.

2.2.1 Abstraktion

Der wohl wichtigste Grund für MDSD ist die Möglichkeit, auf einer höheren Abstraktionsebene zu programmieren. Die Modelle beschreiben das System mit größeren Bausteinen, als sie die zugrunde liegende Programmiersprache kennt (z.B. »Entität« oder »Komponente« statt »Klasse«).

Dadurch wird die Gesamtkomplexität aufgeteilt in die Abbildung vom Modell auf die Zielsprache, also die Generierung bzw. Interpretation, und die

»eigentliche« Komplexität, die im Modell übrigbleibt. Man kann beide Teile separat angehen und verstehen, und insbesondere muss man zum Modellieren nicht die technischen Details der Implementierung kennen und berücksichtigen.

Dabei müssen die Abstraktionen der Modelle durchaus nicht immer technisch sein. Sie können auch fachliche Aspekte wie einen Versicherungsvertrag oder die Rechenregeln für ein Basel-II-Rating beschreiben. Solche fachlichen Abstraktionen werden dann typischerweise interpretiert, weil sie oft zur Laufzeit eines Systems zur Verfügung gestellt werden. Die Techniken, mit denen solche Abstraktionen geschaffen werden, sind aber generell die gleichen.

2.2.2 Einheitliche Architektur

Wenn man ein formales Modell automatisiert in lauffähige Software verwandelt, dann passiert das per Definition einheitlich: Wenn ein Modell Entitäten oder Komponenten beschreibt, dann werden alle Entitäten bzw. Komponenten auf dieselbe Weise übersetzt. Es ist also nicht möglich, dass eine Architektur »verwässert«. Es gibt eben nur einige in der Modellierungssprache explizit definierte Punkte, in denen sich beispielsweise Entitäten unterscheiden können.

Das ist sehr nützlich, denn man kann an einer zentralen Stelle einen Rahmen festlegen, der universell gilt. Das vereinfacht die Dokumentation der Architektur – und es stellt sicher, dass das System mit dieser Architektur tatsächlich übereinstimmt!

Das bedeutet übrigens nicht, dass die Architektur ein für alle Mal festgelegt ist und sich nie mehr ändern kann oder darf. Vielmehr werden Modellierungssprachen und Generatoren ähnlich wie Frameworks iterativ weiterentwickelt und somit den sich veändernden Anforderungen angepasst. Eine durchgängige Änderung – z.B. Umstellung des synthetischen Datenbank-Primärschlüssels von 32- auf 64-Bit-Integer – ist mit MDSD deutlich einfacher, weil man die Änderung an einer zentralen Stelle vornehmen kann.

2.2.3 Entwicklungsgeschwindigkeit

Teilweise wird eine verbesserte Entwicklungsgeschwindigkeit als Argument für MDSD angeführt. Eine solche Zeitersparnis könnte an verschiedenen Stellen erfolgen.

Zunächst einmal könnte die Ersparnis darin bestehen, dass man die Zeit für das Eintippen von Quelltexten spart, die ja nun generiert werden. Das spielt aber in der Praxis eine vernachlässigbare Rolle – eine einfache Überschlagsrechnung zeigt, dass der Zeitanteil für das eigentliche Eintippen von Quelltext bei nicht-trivialen Projekten verschwindend gering ist.

Eine andere mögliche Quelle von Zeitersparnis könnten fertige Generatoren sein, mit denen man eine fertige Architektur einsetzt. Wiederverwendung kann

natürlich nützlich sein, aber ein fertiger Generator »von der Stange« muss oft im Laufe der Zeit so weitgehend angepasst werden, dass die anfängliche Ersparnis wieder aufgebraucht wird.

Ein wirklicher Produktivitätsgewinn ergibt sich mittelfristig, wenn das System gewachsen ist und einige Änderungen hinter sich hat. In dieser Situation zahlt es sich aus, dass das System trotz seiner bewegten Vergangenheit konform zu seiner Architektur ist und es automatisch aktuelle Designdokumentation gibt – das erleichtert Änderungen und hilft beim Ändern Fehler zu vermeiden.

Ein weiterer wesentlicher Vorteil bzgl. der Entwicklungsperformance ist außerdem, dass das Expertenwissen (z.B. eines Softwarearchitekten) in die Automation einfließt und damit für alle Projektmitarbeiter wiederverwendbar wird.

2.2.4 Wiederverwendung

Architekturen, Modellierungssprachen und Generatoren können im Sinne einer Produktlinie zur Herstellung mehrerer Softwaresysteme verwendet werden. Das ist eingeschränkt für technische Domänen wie Spring, Hibernate etc. möglich, Wiederverwendung im großen Rahmen erfordert aber den Zuschnitt der Produktionsstraße auf die konkreten Anforderungen der jeweiligen Produktfamilie.

Wenn man eine Reihe von Systemen baut, die erhebliche Ähnlichkeiten haben, dann ist diese Art der Wiederverwendung ein großer Vorteil – nicht zuletzt, weil die Architektur einheitlich und dokumentiert ist. Näheres hierzu steht in Abschnitt 11.5.

2.2.5 Interoperabilität und Plattformunabhängigkeit

Die OMG hat bei der Definition von MDA (s. Anhang A) die Ziele der Interoperabilität (Herstellerunabhängigkeit durch Standardisierung) und der Plattformunabhängigkeit verfolgt. Es war dabei Teil der Vision, aus denselben Modellen Code für unterschiedliche Zielumgebungen zu generieren, z.B. für Java EE und für .NET, so dass ein Plattformwechsel nicht mehr Arbeit erfordert, als »einen Schalter umzulegen«.

Diese Art von Plattformunabhängigkeit sieht in Hochglanz-Präsentationen gut aus, in der Praxis erfordert eine Umstellung aber auch mit Modellen erhebliche Arbeit. Ein Hauptgrund dafür ist, dass die Zielplattform sich in der einen oder anderen Weise in den Modellen niederschlägt. Ein anderer Grund ist, dass man ja in der Regel nicht 100 % eines Systems generiert und der von Hand geschriebene Code mit dem generierten Code zusammenspielen muss und sich auch dort Abhängigkeiten von der Zielplattform finden.

Man kann die Zielplattform recht weitgehend kapseln (zumindest solange man innerhalb derselben Programmiersprache bleibt), aber das erfordert erheblichen Mehraufwand. Der lohnt sich aber oft nicht – oder wann haben Sie zuletzt

tatsächlich ein fertiges System von Java auf .NET portiert? Trotzdem ist eine Migration bei einem modellgetriebenen Projekt natürlich wesentlich leichter durchzuführen als bei einem konventionellen Projekt.

2.2.6 Softwarequalität

MDSD kann die Qualität der entwickelten Software verbessern, aber eine solche Aussage ist mit Vorsicht zu genießen, denn sie ist beliebig vage. Der primäre Qualitätsgewinn liegt darin, dass Systeme eine einheitliche, getestete und dokumentierte Architektur umsetzen.

2.3 Erstes Fallbeispiel: Generator für einfache Webanwendungen

Bevor wir tiefer in die Konzepte und die Terminologie einsteigen, sehen wir uns eine einfache Webanwendung als typisches Beispiel an. Das soll das Vorgehen und die möglichen Vorteile modellgetriebener Entwicklung illustrieren.

Dieses Beispiel betrachtet MDSD aus der Sichtweise eines Anwenders, der einen Generator benutzt. Er muss nicht bis ins Detail verstehen, was hinter den Kulissen geschieht, und seine Perspektive illustriert, wie MDSD sich auf ein Projektteam auswirkt.

Dieses Beispiel ist nicht als »Krönung der Generierungskunst« gedacht. Reale Systeme stellen oft viel spezifischere Anforderungen an die Softwarearchitektur, die mit modellgetriebenem Vorgehen komfortabel in den Griff zu bekommen sind. Wir beschreiben hier außerdem nicht alle Lösungen bis ins letzte Detail. Das Beispiel zeigt aber die Grundideen modellgetriebenen Arbeitens, und es hat den Vorteil, einfach zu sein.

2.3.1 Die technische Basis

Als technische Basis kommen Java Server Faces [JSF] für die Präsentation, Spring [Spring] für die Komponenteninfrastruktur und Hibernate [Hibernate] für die Persistenz in einer relationalen Datenbank zum Einsatz.

Die Entscheidung für konkrete Bibliotheken ist aus Sicht eines Programmierers, der den Generator einfach verwendet, weitgehend transparent – und das ist ja fürs Erste unser Blickwinkel. Der Generator legt fest, welche Artefakte von Hand erstellt werden müssen und wie diese Dinge intern auf die Basistechnologie abgebildet werden.

Das ist einer der wichtigen Vorteile modellgetriebener Entwicklung: Es genügt, wenn einige Entwickler die verwendete Plattform gut kennen, weil sie ihr Wissen in den Generator einfließen lassen. Dadurch profitiert das ganze System davon, und es ist architektonisch einheitlich.

Die verwendeten Bibliotheken werden trotzdem schon hier am Anfang erwähnt, und zwar erstens um die Neugier der Leser zu befriedigen. Zweitens beeinflussen sie an einigen Stellen doch mehr oder weniger direkt das Vorgehen, so dass es nützlich ist, dieses Wissen im Hinterkopf zu haben.

Dennoch ist dieses Beispiel auch ohne Kenntnis dieser Bibliotheken problemlos zu verstehen – nicht zuletzt deshalb, weil der Generator ihre Details kapselt.

2.3.2 Entitäten, Komponenten und Webseiten

Das Hauptanliegen, unsere Webanwendung modellgetrieben zu entwickeln, besteht darin, technischen Infrastruktur-Code zu generieren. Das betrifft in unserem Fall drei Bereiche: Persistenz, Komponenten und Masken.

Alle drei Bereiche modellieren wir mit UML-Diagrammen, wobei wir Stereotypen verwenden, um verschiedene Arten von Klassen zu unterscheiden. Als Beispiel wird ein Ausschnitt aus einer Software für eine Autovermietung verwendet.

Die folgenden Abschnitte beschreiben jeweils die Einzelheiten.

Entitäten

Für die Entitäten, also die Datentypen, die in der Datenbank gespeichert werden sollen, ist ein Klassendiagramm gut geeignet (s. Abb. 2–1). Dabei kennzeichnet der Stereotyp `<<Entity>>` die Datentypen, um sie von anderen Arten von UML-Klassen zu unterscheiden, z.B. Komponenten.

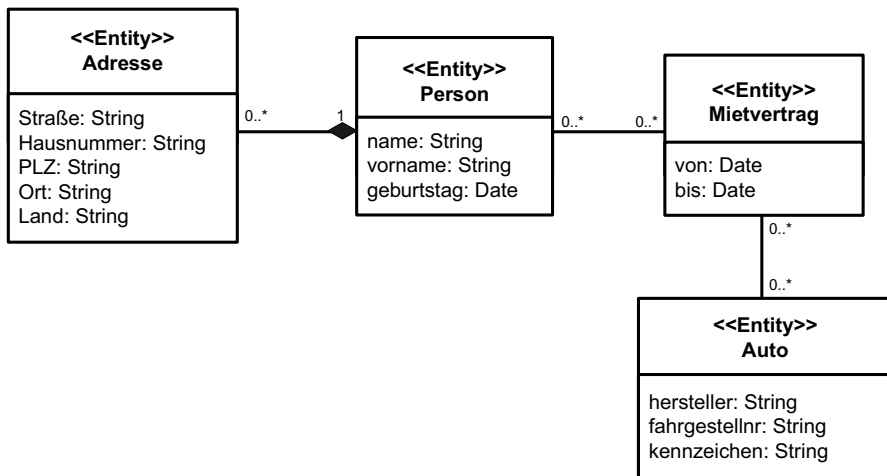


Abb. 2–1 Klassendiagramm mit Entitäten – Person, Adresse, Mietvertrag, Auto

Neben der Liste der Attribute sowie den Beziehungen zwischen den Entitäten enthält das Modell auch Informationen über Kompositionsbeziehungen: Wenn eine Person gelöscht wird, sollen auch alle dazugehörigen Adressen gelöscht werden.

Außerdem kann man ausdrücken, dass eine Beziehung bidirektional ist: Wenn eine Person einen Mietvertrag schließt, erscheint der Vertrag in der Liste aller Verträge der Person, und gleichzeitig »kennt« der Vertrag die Person.

Komponenten

Die Funktionalität der Anwendung steckt in Komponenten, von denen es zwei Arten gibt: *BusinessComponents* und *DataAccessObject-Komponenten* (DAOs). Erstere enthalten Logik im engeren Sinne, Letzere dienen zum Zugriff auf die Datenbank (s. Abb. 2–2).

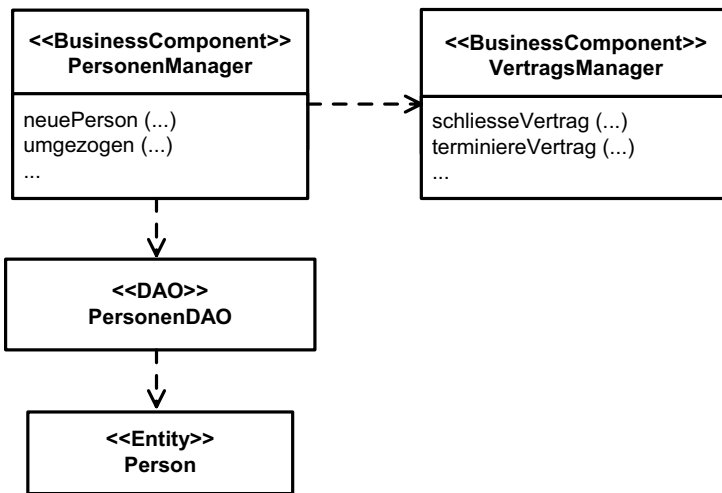


Abb. 2–2 Klassendiagramm mit Komponenten

Komponenten haben natürlich Operationen, die von Programmierern mit dem Anwendungscode gefüllt werden. Außerdem enthält das Modell Abhängigkeiten zwischen Komponenten – der generierte Code kümmert sich mit Hilfe dieser Informationen darum, dass die Komponenten in der richtigen Weise miteinander »verdrahtet« sind.

Außerdem gehört zu jeder DAO-Komponente genau eine Entity, für deren Persistenz sie zuständig ist.

Webseiten

Der letzte Teil der Webanwendung sind die Seiten, die im Browser angezeigt werden.

Die Navigation zwischen den Seiten wird als Aktivitätsdiagramm modelliert, wobei jede Activity eine Seite und jeder Übergang eine Navigation repräsentiert – der Name der auslösenden JSF-Action steht als »Bedingung« in eckigen Klammern am Übergang (s. Abb. 2–3).

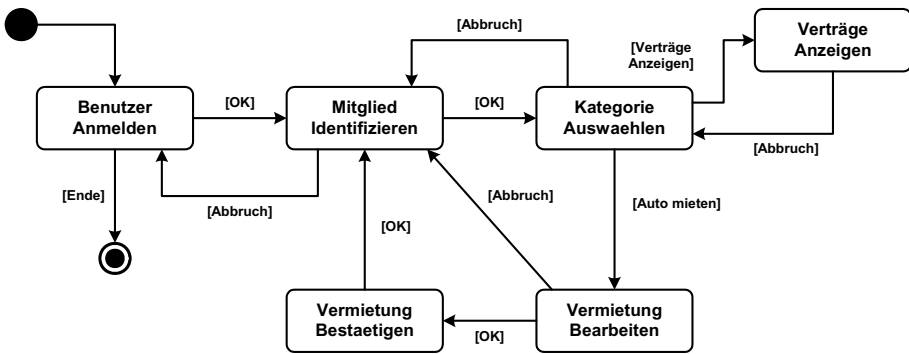


Abb. 2-3 Aktivitätsdiagramm mit Masken

2.3.3 Anwendungscode

Nach diesem Überblick darüber, was das Modell enthält, ist der nächste Schritt, dass wir uns ansehen, was noch von Hand implementiert werden muss, damit eine fertige Anwendung entsteht.

Entitäten

Für die Entitäten werden Java-Klassen sowie die Mapping-Dateien für die Persistenz generiert, wobei der generierte Code sich sogar um Besonderheiten wie Komposition oder bidirektionale Beziehungen kümmert. Hier ist keine manuelle Implementierungsarbeit nötig.

Komponenten

Für jede Komponente muss eine Java-Klasse implementiert werden, die die Implementierungen der fachlichen Methoden bereitstellt. Diese Implementierung erbt von einer generierten abstrakten Superklasse, die sich um das »Zusammenstecken« der Komponenten kümmert und alle benötigten anderen Komponenten über get-Methoden bereitstellt:

```

package my.demo.pkg;

/**
 * Diese Klasse berechnet den Rabatt, der einer Person auf Grund ihrer
 * Eigenschaften zusteht. Sie enthält die von Hand erstellte Implementierung der
 * Geschäftslogik.
 */
public class RebateCalcImpl extends AbstractRebateCalc {
    // die Klasse erbt von einer generierten Basisklasse
  
```

```
public int getRebatePercentage (long personId) {
    // die Methode getPersonDao() wird von der generierten
    // Basisklasse bereitgestellt und kapselt den Zugriff auf
    // eine andere Komponente.
    Person person = getPersonDao ().get (personId);

    int result = 0;
    if (person.getContracts().size() >= 10)
        result += 10; // Stammkundenrabatt

    if (System.currentTimeMillis() - person.getBirthday() >
        25 * 86400 * 1000L)
        result += 5; // Rabatt für Fahrer über 25

    return result;
}
}
```

Für jede Komponente gibt es außerdem ein generiertes Interface, das die »öffentliche« Funktionalität bereitstellt. Andere Komponenten bekommen nur dieses Interface zu Gesicht.

Webseiten

Die Implementierung der JSP-Seiten bleibt vollständig dem Anwendungsprogrammierer überlassen – man kann zwar auch GUI-Code generieren, das erfordert aber eine genaue Analyse der konkreten Anwendung, weshalb wir für unseren allgemeinen Webseitengenerator darauf verzichten.

Der Name der JSPs muss dem Namen im Aktivitätsdiagramm entsprechen, und für jede der JSPs muss es eine gleichnamige Managed Bean geben. Der generierte Code kümmert sich um die Konfiguration der Managed Beans sowie die Navigation zwischen den Seiten.

Dies ist übrigens ein Beispiel dafür, dass moderne Bibliotheken wie JSF oft schon sehr leistungsfähige Abstraktionen bieten – die Beschreibung der Navigation in der *faces-config.xml* entspricht praktisch 1:1 dem UML-Modell, so dass das Generieren hier keinen Abstraktionsgewinn bringt. Andere Vorteile wie Validierung zur Compilezeit oder eine grafische Übersicht über den Seitenfluss hat man aber natürlich trotzdem.

2.3.4 Generator

Dieser Abschnitt blickt »hinter die Kulissen« und geht mehr als die anderen auf die verwendeten Bibliotheken ein. Er beschreibt genauer, welche Dateien generiert werden und wie das Ganze zusammenspielt.

Hier passiert ein »Rollenwechsel«: Die bisher beschriebenen Inhalte muss jeder Entwickler kennen und verstehen. Was jetzt kommt, ist nur noch zum Wei-

terentwickeln des Generators erforderlich. Natürlich ist es hilfreich, wenn möglichst viele Projektmitglieder den Generator verstehen, und es spricht auch im Prinzip nichts dagegen, dass jeder Entwickler in XP-Manier den Generator weiterentwickelt. Trotzdem ist die Trennung der beiden Sichtweisen nützlich, weil es sich um Kapselung handelt.

Es geht hier nur darum, einen oberflächlichen Eindruck zu geben – der gesamte zweite Teil des Buches behandelt die hier angerissenen Themen ausführlicher.

Was wird generiert

Aus den verschiedenen UML-Modellen werden unterschiedliche Dateien generiert – auf diese Weise fließen die Modellinhalte in den Build-Prozess und letztendlich in die fertige Anwendung ein.

Abb. 2–4 gibt einen Überblick darüber, was wie generiert wird.

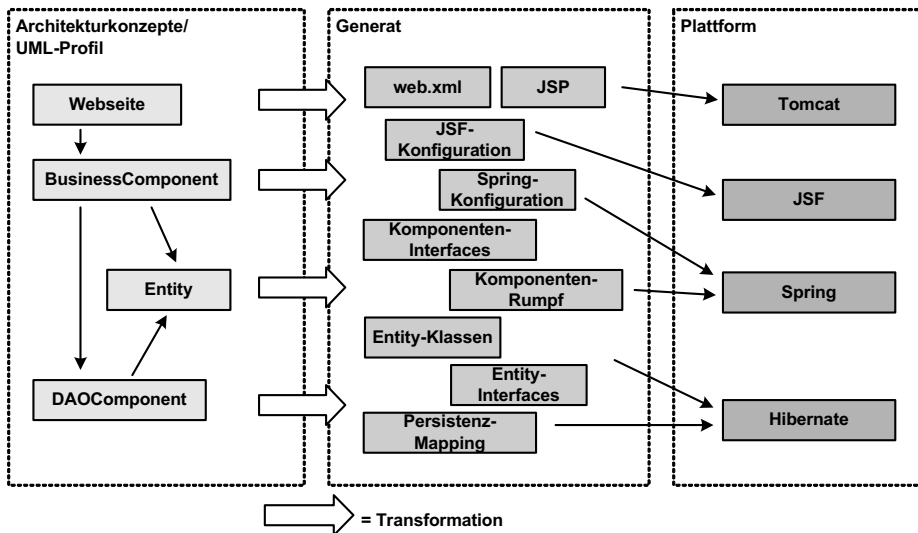


Abb. 2–4 Zusammenhang von Modellen und generierten Artefakten

Zur besseren Trennung gibt es im Verzeichnisbaum des Projektes zwei Source-Verzeichnisse: `src-gen` enthält alle generierten Dateien, und `src` enthält alle manuell erstellten Dateien.

Moderne IDEs unterstützen durchweg das Arbeiten mit mehreren Source-Verzeichnissen, so dass z.B. die Implementierung einer Komponente im Verzeichnisbaum unter `src` problemlos von einer abstrakten Basisklasse erben kann, die im selben Package unter `src-gen` liegt.

Der Build-Prozess erhält einen zusätzlichen Schritt, nämlich das Generieren. Dieser Schritt erfolgt vor dem Compilieren, und der Compiler muss so aufgerufen werden, dass er Quelltexte aus zwei Verzeichnisbäumen gemeinsam verarbeitet.

Templates

Der Blick hinter die Kulissen wäre nicht vollständig, ohne dass wir uns zumindest oberflächlich ansehen, wie der Generator weiß, was er generieren soll. Diese Information erhält er aus sogenannten Templates.

Das folgende Listing zeigt beispielhaft ein Template für den open-ArchitectureWare-Generator, das das Java-Interface für eine Geschäftslogik-Komponente generiert:

```
«DEFINE BIComponentInterface FOR BusinessComponent»
  «FILE fullyQualifiedName.replaceAll (".", "/") + ".java"»
  package «package.fullyQualifiedName»;

  import java.util.*;

  public interface «name» {
    «EXPAND method FOREACH operation»
  }
«ENDFILE»
«ENDEDEFINE»

«DEFINE method FOR Operation»
  «returnType.javaName» «name» (
    «FOREACH parameters AS param SEPARATOR ", "»
    «param.type.javaName» «param.name»
    «ENDFOREACH»
  );
«ENDEDEFINE»
```

Die französischen Anführungszeichen kennzeichnen dabei »Befehle« in der Template-Sprache (s. Abschnitt 8.4.3 für eine ausführlichere Beschreibung).

Dieses Template erzeugt z.B. das folgende Interface:

```
package my.demo.pkg;

import java.util.*;

public interface ContractManager {
  void confirmContract (Contract contract);
  void extendContract (Contract contract, Date new endDate);
}
```

2.3.5 Gesamtsicht

Damit haben wir uns der Reihe nach die technische Basis, die Systembestandteile, den manuell erstellten und den generierten Code sowie schließlich den Generator selbst angesehen.

Jetzt betrachten wir noch an einigen typischen Projektsituationen das Zusammenspiel dieser Einzelteile.

Build-Prozess

Als Erstes betrachten wir den Build-Prozess. Er beruht auf dem gleichen Ablauf wie der Build-Prozess ohne MDSD, es kommt allerdings gleich zu Beginn ein weiterer Schritt hinzu, nämlich die Generierung.

Im Anschluss daran übersetzt der Compiler jetzt sowohl manuell geschriebene als auch generierte Quelltexte. Sie referenzieren sich gegenseitig und sind deshalb auch nur gemeinsam übersetzbar.

Deshalb müssen als erster Schritt beim Bauen des Systems alle generierten Quelltexte neu aus den Modellen erzeugt werden. Dabei wird die alte Version der Quelltexte unmittelbar vor dem Generieren gelöscht, damit keine »Leichen« übrig bleiben.

Dieser Build-Ablauf macht auch noch einmal deutlich, dass die Modelle die Rolle von »Quelltexten« haben: Sie fließen unmittelbar in den Build-Prozess ein, an dessen Ende das compilierte Programm steht.

Änderungen am Modell

Die häufigste Tätigkeit im Projekt, die mit MDSD zu tun hat, ist das Ändern von Modellen. Deshalb ist es wichtig, dass es hierbei möglichst gute Werkzeugunterstützung gibt (s. Kapitel 6).

Nehmen wir also an, ein Entwickler benötigt das zusätzliche Attribut *postfach* in der Entität *Adresse*, das er in der neu zu erstellenden Komponente *AddressManager* verwenden will. Der erste Schritt besteht darin, das Attribut sowie die Komponente in das Modell aufzunehmen.

Als Nächstes soll die manuelle Implementierung der *AddressManager*-Komponente erfolgen. Doch die generierten Quelltexte auf dem Rechner des Entwicklers sind noch auf dem Stand von vor der Modelländerung – die Änderung im Modell spiegelt sich nicht automatisch sofort wieder.

Also gilt es, den Generator zu starten, damit die Quelltexte auf Basis der geänderten Modelle neu erzeugt werden. Jetzt gibt es die generierte Klasse *AbstractAddressManager*, und man kann die Klasse *AddressManagerImpl* schreiben, die von ihr erbt und die Geschäftslogik enthält.

Jetzt ist der Zeitpunkt gekommen, die automatischen Modultests laufen zu lassen, und wenn sie erfolgreich durchlaufen, dann kann man mit gutem Gewis-

sen die Änderungen – sowohl die geänderten Modelle als auch die von Hand implementierte Klasse – in die Versionsverwaltung einchecken.

Die generierten Klassen werden dabei aber natürlich nicht eingecheckt – sie sind ja Ergebnisse des Build-Prozesses, und man stellt ja auch die *.class*-Dateien nicht unter Versionsverwaltung.

Durch den Einsatz von MDSD umfasste die Änderung zwei zusätzliche Schritte, die Modelländerung und den Generatorlauf. Dafür musste man sich als Entwickler nicht um die Implementierungsdetails der Entitäten und ihrer Persistenz oder der Komponenten kümmern, was das Arbeiten vereinfacht.

Fehlendes Generator-Feature

Ein besonders spannender Fall beim modellgetriebenen Arbeiten tritt dann ein, wenn man feststellt, dass ein Generator-Feature fehlt. Dieser Fall ist idealerweise nicht allzu häufig, aber es wäre utopisch anzunehmen, dass es ihn nicht gibt.

Nehmen wir also z.B. an, dass bei der Entwicklung der Wunsch auftritt, dass Entitäten voneinander erben können – es gibt jetzt nicht mehr nur Autos, sondern Autos und Motorräder, die beide von einer Entität *Fahrzeug* erben sollen.

UML kennt selbstverständlich solche Vererbungsbeziehungen. Der Generator unterstützt sie aber bisher nicht, und deshalb ist es verboten, eine Vererbungshierarchie zu modellieren. Der Generator würde einen Fehler melden und das Modell als ungültig abweisen:

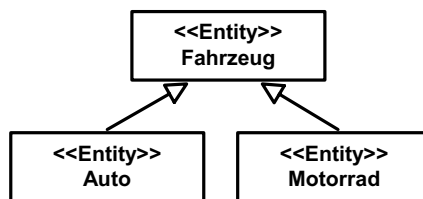


Abb. 2-5 UML-Modell mit Vererbungshierarchie von Entitäten

An dieser Stelle brauchen wir also eine Erweiterung des Generators selbst.

Eine solche Änderung ist automatisch auch eine Änderung der Architektur des Systems, die man nicht nebenbei machen sollte. Je nach Kultur des Projektes ist dies ein Punkt, an dem man z.B. eine kurze informelle Teambesprechung abhält oder die Änderung dem Architektur-Gremium vorlegt.

In diesem Fall haben wir Glück und bei der informellen Teambesprechung ergibt sich schnell ein Konsens, wie das Feature in die Architektur integriert werden sollte – Vererbungsbeziehungen zwischen Entitäten werden mit dem *<subclass>*-Feature von Hibernate ausgedrückt (die detaillierten Auswirkungen dieser Entscheidung übergehen wir hier einfach einmal).

Also setzt sich der Entwickler mit jemandem zusammen, der den entsprechenden Teil des Generators gut kennt, und erweitert ihn entsprechend. Die Änderungen sollten im Prinzip vollständig rückwärtskompatibel sein, und das Ausführen der automatisierten Test-Suite mit dem erweiterten Generator verifiziert das. Also werden die Änderungen am Generator eing_checked, und der Weg zur eigentlichen fachlichen Änderung ist frei.

Das Vorgehen bei Änderungen am Generator ist im Wesentlichen genauso wie bei Änderungen an bibliotheks- oder frameworkartigen Programmteilen. Bei kleinen, überschaubaren Projekten ist ein informelles Vorgehen wie das eben beschriebene möglich, bei größeren Projekten ist ein formalerer Prozess nötig (s. Abschnitt 11.3 für eine ausführlichere Behandlung dieses Themenkomplexes).

2.3.6 Fazit und Ausblick

Wie unterscheidet sich die modellgetriebene Entwicklung einer Webanwendung also jetzt vom »klassischen« Vorgehen? Oder anders gefragt: Was haben wir durch den zusätzlichen Aufwand gewonnen?

Es gibt eine Reihe von Vorteilen, die mehr oder weniger ausgeprägt sind und die wir der Reihe nach betrachten.

- *Einheitliche Architektur.* Ein großer Vorteil besteht darin, dass die Anwendung eine einheitliche Architektur erhält. Es gibt eine durchgängig einheitliche Art, wie Komponenten, Entitäten etc. implementiert werden.
- *Gewonnener Abstraktionsgrad.* Die Modelle sind – zumindest teilweise – näher an der fachlichen Domäne als die technischen Details der verwendeten Bibliotheken. Die Modelle bieten einen Überblick und eine Dokumentation, die per definitionem immer auf dem aktuellen Stand sind.
- *Gekapselte Plattform-Details.* Der Generator sorgt dafür, dass die Details der Verwendung von Spring, Hibernate und teilweise JSF gekapselt sind. Projektspezifische Konventionen wie z.B. die Vergabe von Primärschlüsseln oder das verwendete Transaktionsmodell bei Spring sind an einer zentralen Stelle festgelegt, und man muss sie beim »normalen« Programmieren nicht ständig berücksichtigen. Gleichzeitig kann man diese Festlegungen aber nach Bedarf ändern und erweitern, man ist also nicht in der engen Welt eines CASE-Werkzeuges gefangen.

Dieses Beispiel hat zur Einstimmung auf die detailliertere Behandlung illustriert, wie MDSD in der Praxis aussehen kann und welche Auswirkungen es auf ein Projekt hat.