Modellbasierte Schichtenarchitektur zur Komposition von Geschäftsanwendungen

Peter Hänsgen[1], Heiko Kern[2]

[1]Intershop Communications AG
Intershop Tower, 07745 Jena
p.haensgen@intershop.de

[2]Betriebliche Informationssysteme, Universität Leipzig
Johannisgasse 26, 04103 Leipzig
kern@informatik.uni-leipzig.de

Abstract: Durch den Einsatz modellgetriebener Entwicklungstechniken wie Codegenerierung können viele Vorteile bei der Softwareentwicklung erzielt werden. Im praktischen Einsatz gibt es allerdings auch eine Reihe von Nachteilen. In diesem Beitrag wird eine modellbasierte Schichtenarchitektur vorgeschlagen, welche eine flexible Komposition von Anwendungen aus einzelnen generischen wieder verwendbaren Schichten erlaubt und die eine Verbesserung bzgl. der im Beitrag beschriebenen Probleme bringen soll.

1 Einleitung

Modellgetriebene Softwareentwicklung (*Model-Driven Software Development*, MDSD) [SV06] basiert auf den Konzepten Modellierung, Modellvalidierung, Modelltransformation und Codegenerierung. Trotz verschiedener Vorteile, die sich durch den Einsatz von MDSD ergeben, wurde eine Vielzahl an Problemen auf Basis von Erfahrungen der Autoren mit der Entwicklung von komplexen E-Commerce-Anwendungen und anderen Arbeiten (bspw. in [Uh08]) identifiziert. Diese Probleme werden im Folgenden näher beschrieben.

Werkzeugzentrierung: Für die Umsetzung des MDSD-Paradigmas ist die Entwicklung einer komplexen Werkzeuginfrastruktur notwendig. Dadurch entstehen u.a. große Anfangsinvestitionen, die nur durch eine maximale Wiederverwendung der einzelnen Komponenten in verschiedenen Softwareprojekten gerechtfertigt werden kann.

Synchronisierung: Die Herauslösung der Architektur aus einem Softwareprojekt in ein Entwicklungswerkzeug führt zu Problemen, wenn Entwickler zwar die gleiche Codebasis bearbeiten, aber dabei verschiedene Versionen des MDSD-Werkzeugs benutzen, also in Folge aus dem gleichen Modell verschiedenen Code generieren.

Wiederholbarkeit: Eine besondere Schwierigkeit entsteht bei der Weiterentwicklung von (älteren) Kundenprojekten. Es ist oft nicht möglich die Generierungsinfrastruktur auf Basis der alten Modelle und der inzwischen veralteten und nicht mehr verfügbaren Werkzeuge wiederherzustellen.

Stabilität: Generierter Code entspricht einer statischen Momentaufnahme der aktuellen Kombination der Modelle, Transformatoren und Generatoren. Änderungen in einer dieser Komponenten haben meistens globale Auswirkungen und erfordern eine (aufwändige) Neugenerierung des gesamten Systems.

Korrigierbarkeit: Wenn Codegenerierung genutzt wird, können Änderungen im generierten Code (z.B. nach einem Bugfix in einem Generatortemplate) nicht einfach als kompilierter binärer Patch an den Kunden ausgeliefert werden. Stattdessen erfordert jegliche Änderung ein aufwändiges Upgrade der MDSD-Werkzeuginstallationen aller beteiligten Entwickler, die Neugenerierung aller darauf aufsetzenden Projekte, jeweils mit den kompletten Zyklen zum Kompilieren, Testen und Installieren.

Anpassbarkeit: Werden im generierten Code manuelle Änderungen oder Ergänzungen durchgeführt, muss auch der generierte Code im Versionierungssystem für das Softwareprojekt verwaltet werden, welches wiederum den Verwaltungsaufwand erhöht. Weiterhin kann es später schwierig werden, manuelle Änderungen in der Masse an Code wiederzufinden.

Redundanz: Die Erhöhung des Abstraktionsniveaus in der MDSD soll dazu dienen, Redundanzen im Code zu vermeiden bzw. zu verringern. Dies wird im MDSD-Ansatz durch Codegeneratoren bzw. Modelltransformatoren erreicht. Das Sicherstellen der Konsistenz von Generatoren und Transformatoren stellt ein Problem dar, da oftmals komplexe Abhängigkeiten zwischen den verschiedenen Transformationen bestehen.

Neben den genannten Problemen gibt es noch zahlreiche weitere Schwierigkeiten, die aber aus Platzgründen nicht weiter diskutiert werden können. Um die Probleme zu lösen, wurde ein alternatives Konzept entwickelt, welches die Vorteile von MDSD beibehält, aber die Nachteile vermeidet. Dieses Konzept wird im folgenden Kapitel 2 erläutert.

2 Konzept für eine modellbasierte Schichtenarchitektur

2.1 Lösungsansatz

Ausgangspunkt bei der Entwicklung des Lösungsansatzes sind die folgenden vier Prinzipien: (1) **Ausnutzung von Selbstähnlichkeit:** Unabhängig vom Abstraktionsgrad sind Metamodelle für ein bestimmtes Problemfeld (Daten, Prozesse, …) strukturell ähnlich. Das Schichten-Konzept soll ein vereinheitlichtes Metamodell verwenden. Damit verliert man zwar an Semantik, gewinnt aber an Universalität. (2) **Problemzerlegung:** Komplexe Probleme können in einfache Probleme zerlegt werden. Jedes dieser Teilprobleme soll durch eine spezielle Schichtenimplementierung auf Basis einer einheitlichen Infrastruktur gelöst werden. (3) **Modellinterpretierung:** Statt auf statischem generiertem Code soll der Schichten-Ansatz auf der Interpretierung von Modellen basieren. (4) **Wiederverwendung von Infrastruktur-Code:** Verschiedene Schichten benötigen immer wieder den gleichen Infrastruktur-Code, z.B. für die Initialisierung, für die Erzeugung und Abfrage von Objekten, für die Bereitstellung von Metainformationen usw. Das neue Konzept soll eine einheitliche Infrastruktur beinhalten, auf der die verschiedenen Abstraktionsebenen aufsetzen können.

2.2 Schichtenarchitektur

Auf Basis dieser Entwicklungsziele wurde ein Konzept entwickelt, welches auf einer Schichtenarchitektur [Fo03] beruht, bei der die einzelnen architektonischen „Bausteine“ übereinander gestapelt und über Domänenmodelle [Ev04] konfiguriert werden können. Dabei findet keine Generierung von Code aus Modellen statt, sondern die gesamte Implementierung des Traversierens durch die Modelle und der Steuerung des Verhaltens einer Schicht wird aus dem Entwicklungswerkzeug in die Anwendung verlagert. Eine Schicht kann somit als ein Interpreter für ein Modell einer bestimmten Abstraktionsstufe betrachtet werden [Vö07].

Zunächst wurde eine einheitliche Schnittstelle (Core-API) definiert, welche für die verschiedenen Schichten genutzt werden kann und durch diese implementiert wird. Im Fokus stand dabei die Abbildung und Verarbeitung von Daten. Die API besteht aus den folgenden fünf Elementen: (1) **Layer:** Repräsentiert eine Schicht und erlaubt den Zugriff auf Node Spaces. (2) **Node Space:** Stellt die Menge aller Nodes eines bestimmten Typs dar und dient ihrer Verwaltung. (3) **Node:** Stellt ein konkretes Objekt dar. Ein Node hat einen Typ, eine eindeutige ID sowie beliebig viele Properties und Relationen mit anderen Nodes. (4) **Property:** Ist ein einfaches atomares Attribut eines Nodes. (5) **Relation:** Ist eine Beziehung zwischen Nodes.

Weil es für die Anwendungsprogrammierung ungünstig ist, nur reflexiv auf die Datenstrukturen der Core-API zuzugreifen, wurde zusätzlich ein Mapping eingeführt, welches Typen aus der Anwendungsdomäne in Form von Java-Interfaces ausdrückt und diese über Proxies automatisch auf die Core-API, also auf Nodes, Properties und Relations, abbildet. Zusätzlich gibt es eine generische Factory, welche zur Erzeugung, zum Entfernen und zur Suche von Domänenobjekten dient und die dazu auf dem jeweiligen Node Space des entsprechenden Typs operiert.

Die Core-API wird von verschiedenen Schichten (Layers) implementiert (siehe Abbildung 1), welche jeweils ein bestimmtes Verhalten bereitstellen. Eine Schicht löst dabei genau ein einzelnes Teilproblem, welches entweder technischer Natur (z.B. Übertragung über ein Kommunikationsprotokoll) oder fachlicher Natur (z.B. Unterstützung von Mehrsprachigkeit) sein kann. Das Problem wird dabei immer bezüglich der Core-API gelöst. So kann beispielsweise eine Schicht implementiert werden, welche die Versionierung von Node Spaces und Nodes sowie ihren Relations und Properties implementiert. Dadurch ist die Schicht in der Lage, die Versionierung für jegliche Domänenobjekte zu übernehmen. Dabei kann die Schicht wiederum einer bestimmten Problemdomäne zugeordnet werden, d.h. sie besitzt selbst ein schichtenspezifisches Domänenmodell. Im Beispiel der Versionierungsschicht könnte dies Begriffe wie „Branch“, „Revision“, „VersionedObject“ oder „ObjectVersion“ umfassen. Dieses Domänenmodell wird, wie bereits für die oberste Ebene geschildert, in genau der gleichen Weise mittels Java-Interfaces implementiert und kann damit generisch auf die Core-API der nächsten unterliegende Schicht abgebildet werden.

2.3 Komposition von Anwendungen

Aufgrund der einheitlichen Core-API können Schichten beliebig miteinander kombiniert und übereinander gestapelt werden. Eine Schicht kann dabei entweder die oberste Schicht (*top layer*), eine Zwischenschicht (*intermediate layer*) oder die unterste Schicht (*bottom layer*) in einem Stapel bilden. Je nach ihrer Funktion kann sie Aufrufe an unterschiedlich viele andere Schichten delegieren. Das Verhalten der gesamten Applikation ergibt sich damit aus der Gesamtheit des Verhaltens der einzelnen übereinander gestapelten Schichten.

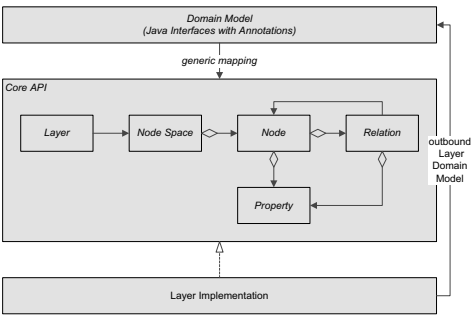

Abbildung 1: Core-API und Layer-Implementierung

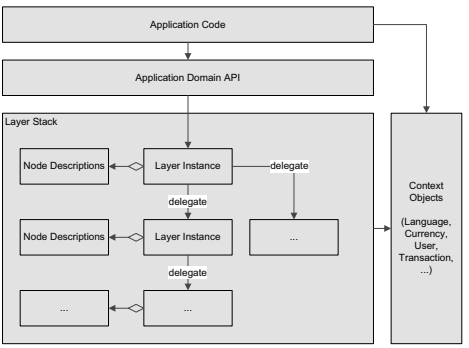

Abbildung 2: Layer Stack

2.4 Anwendungsbeispiel

Um das vorgestellte Schichtenkonzept zu testen, wurden mehrere Schichten aus verschiedenen Bereichen sowie eine gemeinsame Infrastruktur zur Verwaltung der Schichten implementiert. Mit den Schichten wurde eine Beispielanwendung für ein Online-Shopping-System umgesetzt. Dabei können die benötigten Schichten gemäß den gewünschten Features der Anwendung aus der zur Verfügung stehenden Bibliothek ausgewählt und zusammengesetzt werden. Dafür ist keinerlei Codegenerierung notwendig, sondern es werden lediglich fertige Komponenten miteinander verschaltet und konfiguriert. Im Folgenden sollen zwei konkrete Layer-Implementierungen und deren Kombination genauer beschrieben werden.

Currency Layer: Der Currency Layer ist eine Schicht, welche eine zusätzliche Währungsdimension für Properties einführt, d.h. sie erlaubt die Speicherung von währungsspezifischen Werten. Dies könnte z.B. der Preis eines Produktes sein, der in Euro und in US-Dollar anzugeben ist. Im Prototyp werden nur wenige verschiedene Währungen erwartet, die sich nicht zur Laufzeit ändern, darum werden entsprechend markierte Properties einfach in separate Properties im Delegate-Layer gemappt, welche den ISO-Code der repräsentierten Währung als Bestandteil des Property-Namens enthalten (siehe Abbildung 3). Nicht-währungsspezifische Properties bleiben unverändert.

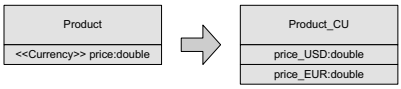

Abbildung 3: Strukturveränderung durch den Currency Layer

Abbildung 4: Strukturveränderung durch den Localization Layer

Localization Layer: Der Localization Layer erlaubt die Einführung von mehrsprachigen Properties, wie z.B. dem Namen oder der Beschreibung eines Produkts. Im Prototyp wurde angenommen, dass eine große Zahl von Sprachen unterstützt werden muss, darum werden lokalisierte Properties auf eine andere Art behandelt als durch den Currency Layer. Sie werden in eine zusätzliche Klasse ausgelagert, welche alle Werte einer bestimmten Sprache enthält (siehe Abbildung 4).

Kombination von Schichten: Die beiden Schichten können nun in einer konkreten Anwendung alleine oder gemeinsam benutzt werden. Sollen in der Zielanwendung sowohl mehrere Währungen als auch mehrere Sprachen unterstützt werden, so werden beide Schichten miteinander kombiniert (siehe Abbildung 5). Durch die sequentiellen Transformationen ergibt sich eine resultierende Datenstruktur, welche am unterliegenden Delegate-Layer registriert wird (z.B. einer Persistenzschicht). Das Domänenmodell der Anwendung ist von der Schichtenkombination selbst nicht betroffen und muss nicht geändert werden. Die Layer-Implementierungen bleiben ebenfalls unverändert und können auch für andere Anwendungen benutzt werden.

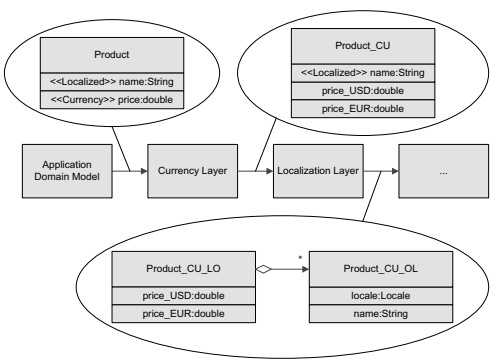

Abbildung 5: Kombination von Currency Layer und Localization Layer

3 Zusammenfassung und Diskussion

Im Beitrag wurden verschiedene Probleme genannt, welche beim praktischen Einsatz von MDSD zur Entwicklung komplexer Softwaresysteme auftreten. Als Lösung wurde die Komposition von Anwendungen auf Basis einer modellbasierten Schichtenarchitektur vorgeschlagen, bei der die einzelnen Schichten mit Modellen konfiguriert werden können und interpretativ arbeiten. Bezugnehmend auf die in der Einleitung geschilderten Probleme, konnten diese mit der Schichtenarchitektur verbessert werden.

Werkzeugzentrierung: Für eine Umsetzung des Konzeptes sind keine speziellen Werkzeuge notwendig, d.h. der Entwicklungsaufwand für MDSD-Werkzeuge entfällt vollständig. Die Beispielanwendung wurde komplett in Java mit den dafür verfügbaren ausgereiften Entwicklungswerkzeugen umgesetzt.

Stabilität: Eine Anpassung der Anwendung an geänderte Anforderungen kann sehr leicht durch Neukombination oder Austausch einzelner Schichten erreicht werden. Dazu ist keinerlei Änderung im eigentlichen Domänenmodell der Anwendung notwendig. Anderseits führen Änderungen im Domänenmodell zu keinen weiteren Änderungsaufwänden in den unterliegenden Schichten, denn die Schichtenimplementierungen bleiben davon unberührt.

Synchronisierung: Die Entwicklung von Anwendungen oder Schichten ist unabhängig vom verwendeten Entwicklungswerkzeug. Jeder Entwickler kann eine eigene Umgebung nutzen.

Wiederholbarkeit: Ältere Projekte können einfach weiterentwickelt werden, weil keine komplexen Werkzeuge gebraucht werden.

Korrigierbarkeit: Schichtenbausteine können in binärer Form ausgeliefert werden. Sie können jederzeit durch das Einspielen von Patches korrigiert werden. Dabei ist keinerlei Neugenerieren der Kundenprojekte notwendig.

Anpassbarkeit: Schichten sind generisch, d.h. sie können derzeit nicht an Spezialfälle angepasst werden. Es ist aber möglich, eine Schicht komplett durch eine andere Schicht zu ersetzen.

Redundanz: Redundanz kann vermieden werden, indem oft benötigte Features als einzelne Schicht implementiert werden. Diese Schicht kann dann in unterschiedlichen Konstellationen wiederverwendet werden. Weiterhin kann eine gemeinsame Infrastruktur für alle Schichten benutzt werden und muss nicht mehrfach implementiert werden.

4 Literaturverzeichnis

[SV06] Stahl, T.; Völter, M: Model Driven Software Development: Technology, Engineering, Management. John Wiley & Sons, 2006.

[Uh08] Uhl, A.: Model-Driven Development in the Enterprise. In: IEEE Software, Vol. 25, No 1; 2008.

[Ev04] Evans, E: Domain-Driven Design: Tackling Complexity in the Heart of Software. Pearson Education, 2004.

[Fo03] Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley, 2003.

[Vö07] Völter, M.: Generieren vs. Interpretieren – Die andere Seite von MDSD, Vortrag auf der OOP 2007, http://www.voelter.de/data/presentations/GenerierenVsInterpretieren.pdf, Letzter Aufruf 14.06.2010