Spring und modellgetriebenes Vorgehen: Gegensatz oder Ergänzung?

Markus Völter, voelter@acm.org, www.voelter.de
Eberhard Wolff, eberhard.wolff@saxsys.de
Peter Friese, peter.friese@LHsystems.com

Die leidvollen Erfahrungen mit komplexen Plattformen wie J2EE oder CORBA haben zur Entstehung zweier neuer Strömungen im Bereich des Software Engineering geführt: Auf der einen Seite sprießen leichtgewichtige Frameworks – allen voran Spring [1] – aus dem Boden. Gleichzeitig erfreuen sich Modellgetriebene Konzepte wie MDA [2] oder MDSD [3] wachsender Beliebtheit, deren Anspruch aber über das Erleichtern des Umgangs mit einer komplexen Plattform weit hinausgehen. Beide Strömungen nehmen allerdings für sich in Anspruch, die Entwicklung von Anwendungen deutlich zu vereinfachen. So stellt sich die Frage, ob beide Ansätze komplementär einsetzbar sind oder ob sie Gegensätze darstellen.

Zunächst scheinen die Ansätze dieselben Probleme zu lösen. Spring hat als Ziel, die Menge des zu schreibenden Glue Codes deutlich zu reduzieren, also solchem Code, der durch die zugrunde liegenden technischen Infrastruktur motiviert ist, und nicht von der Fachlichkeit . Dadurch stellt es eine Erleichterung gegenüber der üblichen Programmierung vor allem im EJB Umfeld dar, die zurzeit kaum sinnvoll ohne einen Generator wie XDoclet durchzuführen ist. Hier kann leicht der Verdacht aufkommen, dass Codegenerierung nur dazu dient, die Unzulänglichkeiten der Plattform zu beheben.

Diese Argumentation ist bei einfachen Codegeneratoren wie XDoclet sicher richtig. Allerdings ermöglichen Plattformen wie Spring an vielen Stellen einen eleganteren Ansatz. Bei modellgetriebenen Ansätzen steht aber neben Erleichterungen im Umgang mit der Plattform vor allem im Vordergrund, dass man bessere, zur Fachdomäne oder zur Architektur des Systems passende Abstraktionen zur Verfügung stellen kann, durch die eine Umsetzung der Anforderungen in ausführbaren Code deutlich vereinfacht wird. Es wird eben nicht nur Glue Code generiert, sondern es werden neue Ausdrucksmittel zur Verfügung gestellt, aus denen man die nötigen Konstrukte der Zielplattform (EJB, Spring, etc.) ableiten kann. Außerdem werden Validierungen machbar, die ohne das System beschreibende Modelle nicht möglich wären. Da aber auch modellgetriebene Ansätze letztendlich auf Generierung von Code basieren, ergibt sich die Frage, ob man nicht die Plattform so weit optimieren kann, dass man schließlich alles gleich in der Programmiersprache ausdrücken kann und eine Generierung aus Modellen damit überflüssig wird. In diesem Falle wäre die Ausdrucksmächtig der Programmiersprache so

groß, dass man keine besseren Ausdrucksmittel durch die DSL bzw. die damit beschriebenen Modelle hinzugewinnt.

Vor diesem Hintergrund bietet es sich also an, Spring als Beispiel einer modernen Plattform zu betrachten und zu untersuchen, ob und wie ein modellgetriebener Ansatz trotzdem noch sinnvoll ist, oder ob es realistisch ist, davon auszugehen, dass man mit einer weiter verbesserten Plattform auf einen Modellgetriebenen Ansatz verzichten kann.

Eine Frage – Zwei Szenarien

Wir wollen zur Klärung dieser Frage zwei Szenarien beleuchten. In dem einen Szenario wurde der Generator im Rahmen des Projektes an die dort definierte konzeptionelle Architektur angepasst, es wurden also Metamodelle, Transformationen und Generierungstemplates erstellt. Im anderen Falle wurde ein bereits fertiger Generator (eine „Cartridge“) verwendet.

Custom made…

Im ersten Szenario wurde Spring als Plattform für einen modellgetriebenen Ansatz mit dem openArchitectureWare Generator verwendet. Dabei kam folgende Infrastruktur zum Einsatz:

- Tomcat mit Spring als Server
- Rich Client auf Basis Eclipse RCP als Client
- Struts Web Application Framework für die Unterstützung von Web Clients
- Hessian zur Kommunikation zwischen Rich Client und Serveranwendung (Tomcat)
- Hibernate für die serverseitige Persistenz
- Spring für Dependency Injection und zur Erleichterung des Umgangs mit Hibernate und Hessian.

Damit steht als Basis eine moderne Infrastruktur bereit, mit der man recht effizient entwickeln kann. So ist es ist nahezu trivial, mit Hilfe von Spring ein normales Java Objekt als Hessian Service zur Verfügung zu stellen. Auch der Umgang mit Hibernate wird durch Spring deutlich vereinfacht. Insbesondere kann den einzelnen Objekten die Hibernate Session zur Verfügung gestellt werden, ohne dass man sich hier um viel kümmern müsste – wichtig für die Implementierung von DAOs.

Trotz dieses einfachen und produktiven Programmiermodells wurde ein modellgetriebener Ansatz auf Basis des openArchitectureWare Generators gewählt. Da der Generator von sich aus keine Unterstützung für Spring mitbringt – es gibt keine Spring „Cartridge“ – musste der Generator im Rahmen des Projektes entwickelt werden. Dabei wurde architekturgetrieben vorgegangen: Die wesentlichen Elemente der konzeptionellen Architektur (siehe [4]) wurden in Form eines Architekturmetamodells formal beschrieben. Der Generator generiert aus Instanzen dieses Metamodells Code für die technische Plattform.

Als zentrale Architekturbausteine wurden hier Komponenten und Interfaces ausgewählt. Ein System besteht zunächst aus einer Menge von Komponenten, die alle eine Reihe von Interfaces anbieten oder verwenden. Die Komponenten- und Interfacestruktur eines Systems wurde als profiliertes UML Modell beschrieben. Daraus wurden Spring-konforme Komponentenimplementierungen generiert. Beispielsweise wurden aus Abhängigkeiten von Komponenten zu Interfaces („Required Interfaces“) die entsprechenden Dependency-Injection-Properties generiert. Abhängigkeiten zwischen (Spring-)Komponenten müssen also im Modell spezifiziert sein – eine wichtige Eigenschaft, um in größeren Projekten die Abhängigkeiten unter Kontrolle zu behalten: Auf Modellebene kann man sehr leicht Validierungen durchführen um ungültige Abhängigkeiten zu erkennen.

Neben Komponenten und Interfaces wurden als weitere Architekturbausteine die Geschäftsentitäten ausgewählt. Dazu wurde ein UML Stereotyp definiert, mit dem eine Klasse als Geschäftsentität markiert werden kann. Aus einem solchermaßen markierten Modellelement werden folgende Artefakte generiert:

- Eine Klasse als direkte Abbildung einer solchen Entität im Code.
- Eine DAO Klasse, die Methoden zum Zugriff auf einzelne persistente Entitäten erlaubt. Neben den üblichen CRUD-Operationen (Create/Read/Update/Delete) kann man zum Beispiel auch die Suche nach einzelnen Attributen durch Markierung der betreffenden Attribute im Modell generieren lassen.
- Eine Hibernate Konfiguration, um die Abbildung der Attribute auf die Datenbank zu konfigurieren.

Anhand des Entitäten-Beispiels wird offensichtlich, dass selbst bei einer so eleganten Plattform wie Spring noch genügend Dinge übrig bleiben, die man mit Hilfe eines modellgetriebenen Ansatzes elegant lösen kann. Es bleibt offen, ob man möglicherweise die Plattform noch soweit verbessern kann, dass man z.B. mit den generischen Datentypen von JDK 1.5 und geschickter Programmierung die DAOs generisch implementieren kann, so dass eine Generierung dieser Klassen nicht mehr notwendig ist.

Auf keinen Fall ist es jedoch möglich, eine Konfigurationsdatei wie die hier benötigte Hibernate-Konfiguration mit Hilfe generischer Programmierung zu erzeugen und auch die Hibernate-Anfragen für die Suche nach einzelnen Attributen, die in der Hibernate Anfragesprache HQL definiert werden müssen, lassen sich durch eine Verbesserung der Java Plattform wohl kaum umgehen. Hier wird einer der Vorteile des modellgetriebenen Ansatzes deutlich: Nämlich dass man aus dem Modell eben nicht nur Java Code erzeugen kann, sondern auch andere Artefakte. Durch diesen Ansatz kann man zum Beispiel auch Deployment Deskriptoren für die Web Anwendungen erzeugen – und damit das komplette Thema Remoting abhaken. Als Basis hierfür dient ein weiteres Modell, in dem ein System bestehend aus Knoten, Prozessen, Deployments und Komponenten-„Verdrahtungen“ beschrieben wurde. Abb. 1 zeigt ein einfaches Beispielmodell. Das Type Model definiert die unterschiedlichen Komponenten und Entitäten. Mit dem Composition

Model werden Konfigurationen beschrieben, die aus einem oder mehreren solcher Komponenten bestehen können. Das System Model schließlich beschreibt, auf welchen konkreten Knoten die Konfigurationen deployt werden. Hier wird also der Aspekt, wie die Komponenten im Netz verteilt sind und miteinander kommunizieren, in einem Modell definiert und durch Generierung in entsprechende Konfigurationsdateien umgesetzt.

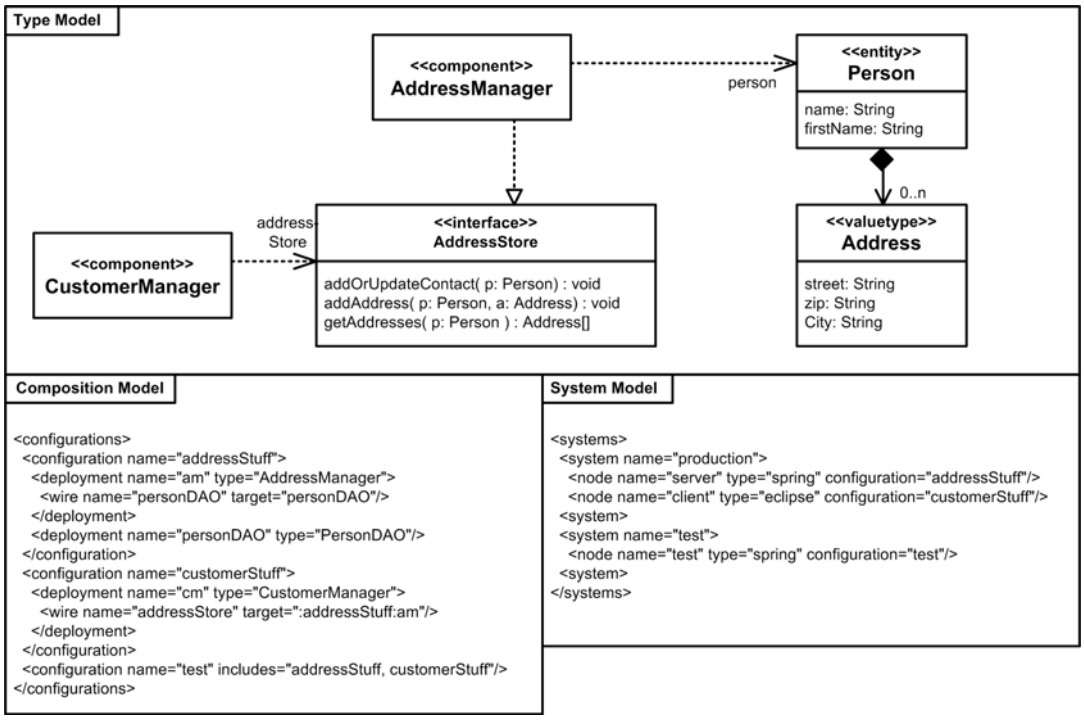

Abbildung 1: Modell zur Beschreibung von Systemen für Szenario 1

Im konkreten Projekt war es außerdem nötig, die Komponenten auch in einer Eclipse-RCP Umgebung laufen zu lassen. Damit dies funktioniert, müssen Eclipse-Plugins generiert werden, und die „Verdrahtung“, die sonst von Spring mittels Dependency Injection erledigt wird, muss anders gelöst werden. Aufgrund der Trennung von konzeptioneller Architektur und deren Beschreibung in plattform-unabhängigen und damit technologieneutralen Modellen ist es sehr leicht, Glue Code auch für diese Plattform zu erstellen – manuelle Programmierung ist nicht notwendig.

Neben der Generierung als solcher – und der damit verbundenen Arbeitsersparnis – hat ein architekturzentriertes, modellgetriebenes Vorgehen weitere positive „Nebeneffekte“. Zum einen wird man gezwungen, die Elemente der Architektur eindeutig festzulegen und zwar im Rahmen der Regeln für die Generierung bzw. im Rahmen der Definition eines Architekturmetamodells. Sie legen fest, mit welchen Mitteln sich die Entwickler ausdrücken können, welche Semantik diese Elemente genau haben, und wie sie auf die

Zielplattform (hier: Spring) abgebildet werden. Würde man auf diesen Ansatz verzichten, so müsste man die Elemente der Architektur wie die erwähnte Abbildung von Entitäten auf Code-Artefakte als Regeln für die Entwickler definieren und die Einhaltung durch Reviews kontrollieren. Bei dem modellgetriebenen Ansatz hingegen muss man die Architektur einhalten, weil die Ausdrucksmittel entsprechend gewählt sind. Dadurch ergibt sich auch zwangsläufig Feedback, falls es Lücken oder Probleme mit der Architektur gibt: Man kann nicht einfach mal „neben der Architektur" entwickeln, sondern man muss Feedback zu dem Generator geben, wenn er benötigte Features nicht implementiert. Als Folge wird die Architektur klarer definiert und den Erfordernissen ständig angepasst – und damit letztendlich besser. Durch die generative Unterstützung wird die Konformität des Codes zur Architektur erhöht. Die Entwicklung eines Generators spielt damit also die Rolle eines „Architekturkatalysators".

Nach so viel Lob für den modellgetriebenen Ansatz stellt sich natürlich die Frage, ob es überhaupt notwendig ist, eine Plattform wie Spring zu nutzen, da man ja um einen Generator offensichtlich nicht herum kommt und dann kann man auch gleich direkt auf eine Infrastruktur wie J2EE aufsetzen. Das erste Argument für Spring ist, dass auch nach der Generierung noch Code von Hand in dem Projekt geschrieben werden muss und hier ist eine gute Plattform wie Spring eine Erleichterung darstellt.

Außerdem ist der Generator als Implementierung der Architektur natürlich einem ständigen Evolutionsprozess unterworfen. Die Eleganz der Spring Lösung schlägt sich hier in einem Produktivitätsvorteil nieder, da die Templates für den Generator bei der Verwendung von Spring sehr einfach werden, so dass man sie auch leicht ändern kann.

In einigen Bereichen können Spring und der Generator jedoch in einem Konkurrenzverhältnis stehen. Im Rahmen des Projekts wurden zum Beispiel auch Pre- und Postconditions für Operationen eingeführt. Den Code für diese Operationen tatsächlich auszuführen, ist ein Aspekt, der bei jedem Methodenaufruf ausgeführt werden sollte. Hier kann man grundsätzlich die Unterstützung der Plattform für Aspekt-orientierte Programmierung verwenden, also in dem Fall das Spring AOP Framework oder auch AspectJ. Alternativ kann man die Aspekte bei der Generierung der Klassen „einweben". Im Projekt wurde die Entscheidung zugunsten des generativen Ansatzes getroffen, aber das muss nicht unbedingt so sein [4].

Bleibt eigentlich nur die Frage, die auch am Ende jedes Verkaufsgesprächs steht: „Was kostet mich das?"

Im Falle des hier skizzierten Vorgehens ist die Frage, wie aufwändig die Implementierung des Generators ist. In diesem konkreten Projekt war die Entwicklung des Generators an sich ein recht geringer Aufwand. Aufwand steckte eher in der Definition, Evolution und der Diskussion über die Architektur – sowohl die konzeptionellen Bausteine, als auch technische Architektur, also die „richtige" Verwendung der Plattform. Ohne die

Entwicklung des Generators hätte diese Diskussion so wahrscheinlich nicht stattgefunden mit entsprechenden Auswirkungen auf die Architektur.

Man braucht also ein Architekturteam, welches die Architektur definiert und den passenden Generator (Metamodell, Templates, DSL) implementiert. Es versteht sich von selbst, dass das Feedback der Entwickler, die den Generator verwenden, Basis der Architekturevolution sein muss. Durch den Generator hat das Architekturteam einen Weg, die Architektur effizient und effektiv auszudrücken und weiterzuentwickeln. Natürlich gibt es noch weitere Vorteile, insbesondere wenn man bei dem Generator nicht auf der Ebene der Architektur stehen bleibt, sondern auch fachliche Abstraktionen anbietet.

Das wichtige Ergebnis unserer Beobachtungen ist, dass selbst bei einer eleganten Plattform wie Spring und einer Ausrichtung ausschließlich auf die Architektur der Aufwand für die Implementierung des Generators durch die Erleichterungen bei der Entwicklung bei weitem aufgewogen wird.

Off the shelf

Hat man erstmal einen Generator für eine bestimmte Zielarchitektur entwickelt, möchte man ihn natürlich gerne weiter verwenden. Im Grunde genommen handelt es sich bei einem Generator ja auch nur um eine Komponente mit einer wohldefinierten Schnittstelle (Eingabemodelle), die anhand eines vorgegebenen Algorithmus (Transformationsregeln) eine Ausgabe (Quellcodes, Deployment Deskriptoren, ...) produziert. Als pragmatische Programmierer wollen wir uns nicht wiederholen, sondern einmal investierte Mühen auch mehrfach ausnutzen.

Um einen Generator sinnvoll in mehreren Projekten nutzen zu können, muss der Generator selbst in zwei Teile zerlegt werden: Zunächst wird ein Generatorkern benötigt, der die notwendigen Basisdienste zur Verfügung stellt. Benötigt werden auf jeden Fall ein Modul zum Einlesen und Instantiieren von Modellen (unanhängig davon, wie sie beschrieben sind, UML-basiert oder nicht) eine Transformationsengine sowie ein Ausgabekanal. Wünschenswert wäre darüber hinaus ein Mechanismus, mit dem der Generator konfiguriert werden kann. Der Generatorkern muss dann so gebaut sein, dass man ihn mit architekturspezifischen Transformationsregeln konfigurieren kann. Diese Transformationsregeln müssen vom Kern vor der Generierung geladen werden, hier bietet sich ein Plug-in-Konzept an. In der Generator-Community hat sich dafür der Begriff „Cartridge“ eingebürgert. Je nach Einsatzzweck wird ein Generator also durch das Zusammenfügen eines Generatorkerns und einer oder mehrerer Cartridges hergestellt.

Genau dieser Ansatz wurde bei den beiden Generatoren openArchitectureWare und AndroMDA gewählt. Während openArchitectureWare sich eher als Generatorbaukasten versteht und nur relativ wenige vorgefertigte Cartridges verfügbar sind, positioniert sich AndroMDA eher als Generatorkern mit einem vorgefertigten Satz an Cartridges für bestimmte Zielarchitekturen. Neben Cartridges für webbasierte Workflows (BPM4Struts)

gibt es solche für Persistenz (EJB bzw. Hibernate) und auch für Spring. Alle Cartridges werden im Quellcode ausgeliefert und können nach Belieben angepasst werden. Die Entwicklung eigener Cartridges erfolgt mit Hilfe der so genannten Meta-Cartridge, die auf Basis eines in UML formulierten Metamodells die entsprechenden Basisklassen sowie die Rahmenkonstruktion der Cartridge erzeugt.

Die AndroMDA Spring Cartridge ist im Rahmen etlicher Projekte zunächst intern eingesetzt und weiterentwickelt worden. Nachdem die Cartridge einen gewissen Reifegrad erreicht hatte, wurde sie der Öffentlichkeit zugänglich gemacht – zunächst im „contrib" Modul von AndroMDA, später als vollwertiger Bestandteil der Distribution.

Von der Funktionsweise ähnelt die Cartridge dem im ersten Szenario beschriebenen Vorgehen: Mittels der Stereotype Entity und Service werden Klassen eines UML-Modells für den Generator markiert. Über Tagged Values lassen sich dann weitere Festlegungen treffen. So ist es z.B. möglich, ein vom Standard abweichendes Mapping für die Entities einzustellen oder das zu verwendende Kommunikationsprotokoll (Hessian, Burlap, RMI, HTTP Invoker) für das Spring Remoting einzustellen. Die in der Cartridge hinterlegten Transformationsregeln bilden dann die abstrakten Konzepte aus dem Modell auf eine konkrete Implementierung ab. Aus einer als Entity markierten Klassen werden folgende Artefakte erzeugt:

- eine Bean mit allen Attributen samt Accessor-Methoden,
- eine DAO für CRUD Operationen
- sowie eine Hibernate Mapping Datei.

Wirklich angenehm ist übrigens die Unterstützung für Queries: eine als *Query* markierte Methode auf einer *Entity* wird automatisch in eine HQL Query transformiert, welche die in der Methode deklarierten Parameter berücksichtigt. Um z.B. einen Finder zu modellieren, der nach allen Attributen der Klasse Person sucht, reicht es, die Methode `findByAllParameters(String vorName, String nachName, int Alter)` zu deklarieren und als *Query* zu markieren. Die Cartridge erstellt die HQL Query, indem sie die Namen der Parameter auf die Namen der Attibute der *Entity* abbildet. Die Logik zur Erzeugung der Query ist übrigens in einer so genannten Metafassade abgelegt. Die Metafassaden werden vom Generator beim Einlesen des Modells erzeugt und stellen eine Instanz des Metamodells der Cartridge dar. Der Vorteil dieser Vorgehensweise ist, dass komplexe Transformationen (wie eben das Bilden einer HQL Query) in Java geschrieben werden können. Die Alternative – komplexe Transformationsregeln mit Hilfe der Templatesprache zu erstellen – führt unweigerlich zu unübersichtlichen und unwartbaren Templates. Durch den Einsatz von Metafassaden können die verwendeten Templates einfach und übersichtlich gehalten werden.

Der vom Generator erzeugte Code nutzt die von Spring gebotenen Erleichterungen wie die Klasse `HibernateTemplate` und die vereinheitlichte Exception-Behandlung zwar aus, doch der eigentliche Mehrwert von Spring entsteht ja bei der Dependency Injection [1]. Auch hier kann die Cartridge punkten: Durch das einfache Ziehen einer Assoziation von einer

als *Service* markierten Klasse zu einer als *Entity* markierten Klasse führt dazu, dass die zur Entity gehörige DAO in der Datei `applicationContext.xml` als Dependency des Service aufgeführt und somit von Spring automatisch zugewiesen wird. Im Service kann man dann ganz einfach mittels `getPersonDao()` auf das zur Entity Person gehörige DAO zugreifen – auch dies entspricht dem Vorgehen in Szenario eins (siehe Abbildung 3). Das gleiche Prinzip funktioniert auch für die Verknüpfung von Services untereinander: einfach eine Assoziation von einem Service zum anderen ziehen – fertig.

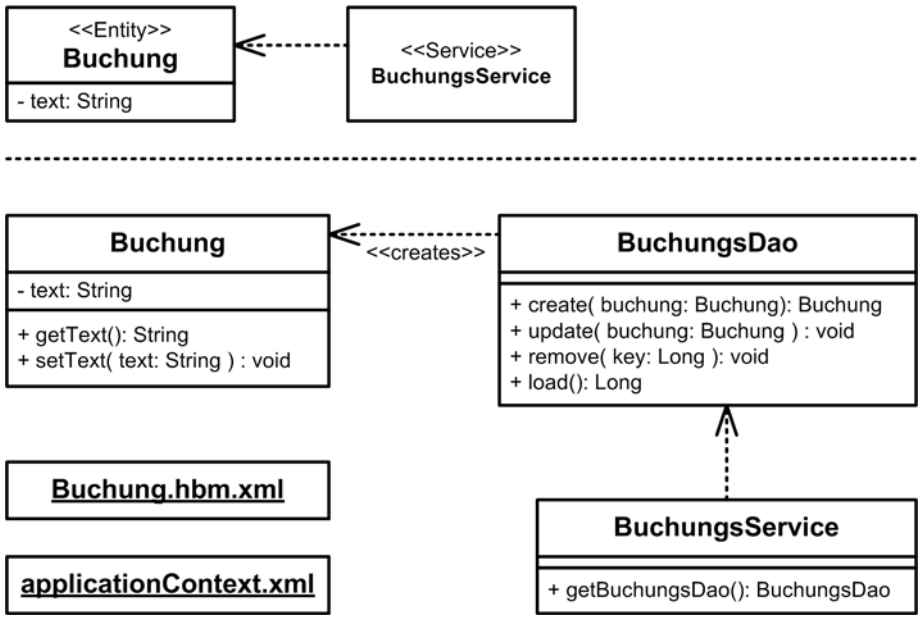

Abbildung 2: PIM und PSM bei der AndroMDA Spring Cartridge.

Eine gute Architektur muss atmen können, d.h. sie muss an die Belange des konkreten Projekts angepasst werden können. Beim modellgetriebenen Vorgehen gibt es drei Bereiche, an denen wir eingreifen können: Als erstes ist hier der von Hand geschriebene Code zu nennen. Architektonische Maßnahmen in diesem Bereich sind allerdings strikt zu unterlassen, da sie bestenfalls lokal begrenzt sind und schlimmstenfalls zu inhomogenen und unwartbaren Systemen führen. Zweiter Einflussbereich sind die verwendeten Klassenbibliotheken, d.h. die technische Plattform. Sofern man diese selbst entwickelt, sind Änderungen recht einfach durchzuführen. Eine größere Mächtigkeit der technischen Plattform kann so zu kompakterem und einfacherem Glue Code führen. Der Siegeszug der Open Source Initiative hat dazu geführt, dass auch diejenigen Komponenten und Bibliotheken verändert werden können, die nicht in der eigenen Organisation entwickelt wurden. Der dritte Einflussbereich ist der Generator bzw. die verwendeten Cartridges. Hier gilt das gleiche wie für Klassenbibliotheken: hat man sie selbst entwickelt, sind Anpassungen und Modifikationen einfacher möglich; wurden sie außerhalb der eigene Firma entwickelt, ist der Anpassungsprozess eventuell langwieriger. Auch hier zeigt sich,

dass eine Open Source Lösung flexibler und schneller an die eigenen Belange anpassbar ist. Letztendlich ist es also gerade bei modellgetriebenen Ansätzen sinnvoll, bei der technischen Plattform und dem Generator auf Open Source zu setzen, um größtmögliche Flexibilität zu erreichen.

Wie bei allen Komponenten gilt es also auch hier, eine Abwägung anzustellen zwischen dem Aufwand einer Eigenentwicklung und der Anpassung einer Fremdentwicklung. Eine Eigenentwicklung ist auf den konkreten Einsatzzweck hin optimiert und trägt z.B. keinen Ballast mit sich herum. Allerdings wird sie auch nicht kontinuierlich von anderen weiterentwickelt. Eine Fremdentwicklung bietet meistens eine gute Ausgangsposition, auf der eigene Erweiterungen und Anpassungen aufgebaut werden können – der anfängliche Aufwand dürfte hier meist niedriger ausfallen als bei einer Eigenentwicklung. Indessen kann sich eine Fremdentwicklung auch in eine unwünschte Richtung fortentwickeln. Gleichzeitig muss man aufpassen, dass eigene Änderungen nach Möglichkeit in die offizielle Entwicklung zurückfließen, damit man nicht langfristig doch bei einem eigenen System landet. Ideal ist es natürlich, einen eigenen Mitarbeiter im externen Team zu installieren, um die Entwicklung im eigenen Sinne beeinflussen zu können.

Fazit

Zunächst lässt sich feststellen, dass modellgetriebenes Vorgehen sich lohnt. Es wird selbst bei einer guten und modernen Plattform wie Spring ein höheres Abstraktionsniveau erreicht, was im Extremfall die Generierung direkt aus fachlichen Anforderungen heraus erlaubt, aber schon bei der Generierung auf Ebene der architektonischen Elemente deutliche Vorteile mit sich bringt. So wird das Wissen um die Umsetzung der Architektur im Generator bzw. in der jeweiligen Cartridge manifestiert. Dadurch wird zum einen die Architektur konsistent im Projekt verwendet und zum anderen auch explizit anpassbar. Zudem ist nur ein kleines Team mit Wissen für den Entwurf einer Architektur notwendig – und der Generator stellt diesem Team einen „Hebel" zur Verfügung, die Architektur in größeren Teams effektiv umzusetzen.

Durch die höhere Abstraktion wird auch der Überblick über das System einfacher – insbesondere, wenn man eine grafische Notation verwendet (die allerdings nicht zwangsläufig UML sein muss). Dadurch ergibt sich insgesamt der Vorteil der höheren Produktivität, mit dem schon zahlreiche technologische Innovationen in der Software-Entwicklung begründet worden sind.

Die beiden in dem Artikel dargestellten Ansätze unterscheiden sich auf den ersten Blick durch die „Make or Buy" Frage in Bezug auf den Generator. Natürlich kann man dazu gezwungen sein, den Generator selbst zu entwickeln, wenn man eine Plattform verwendet, für die es bisher noch keinen Generator gibt, die Möglichkeit des „Buy" (oder Download) bietet sich dann ja gar nicht. Wenn man die Sache genauer betrachtet, steckt allerdings mehr dahinter. Durch die Definition eines technologieunabhängigen Metamodells wird die Architektur eines Projektes *unabhängig von der verwendeten Plattform* formal und klar

definiert. Konsistenzprüfungen sind auf dieser Ebene einfach möglich. Erst in einem zweiten Schritt wird der Generator verwendet, um aus konzeptionellen Modellen dann den Code für eine gegebenenfalls erst jetzt zu wählende Zielplattform zu generieren. Damit macht der modellgetriebene Ansatz – wenn man vom Problemraum her kommt, und nicht vom Lösungsraum her – die Architektur explizit. Vor allem bei komplexeren Architekturen ist dies ein nicht zu unterschätzender Gewinn.

Natürlich hat das Verwenden fertiger Cartridges einige bestechende Vorteile. Insbesondere eben den, dass man einfach ein Modell malt – unter Verwendung der für die Cartridge passenden DSL – und hinten kommt auf der betreffenden Plattform lauffähige Software raus. Man braucht kein Expertenwissen über die Entwicklung des Generators, man muss keine Transformationen und Templates schreiben, auch das Metamodell steht schon fest. Die Anwendung eines Generators ist immer erheblich einfacher als das schreiben eines solchen. Schwierig wird es halt insbesondere dann, wenn man gezwungen ist, mehrere Cartridges zu verwenden, die – da sie vielleicht unabhängig voneinander entwickelt wurden – inkompatible DSLs und Metamodelle verwenden.

Wie kommt man nun aus diesem Dilemma heraus? Die Lösung steckt in der Kaskadierung von Generatoren.

Kaskadierte Modellgetriebene Entwicklung – Die Zukunft

Von Kaskadierung spricht man im Zusammenhang mit MDSD dann, wenn es eine Kette von hintereinander geschalteten Generatoren gibt, wobei der Generator an Position n den Input für den Generator an Stelle n+1 erstellt. Idealerweise lässt man alle diese „Stufen“ im gleichen Generatorkern laufen, sodass die Modelle durch Modell-zu-Modelltransformationen erstellt werden können (siehe auch [5]).

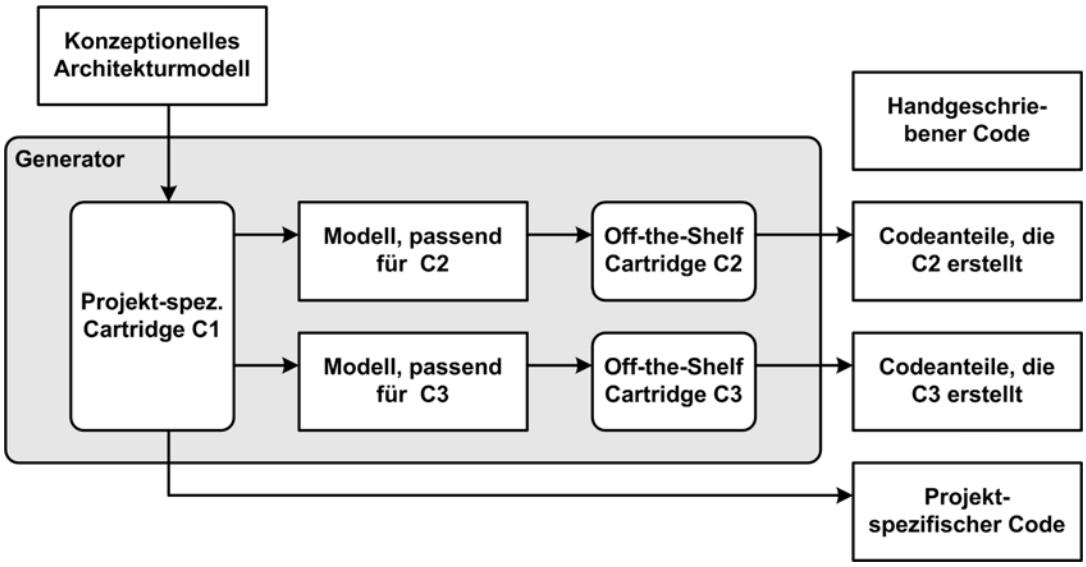

Abbildung 3: Kaskadierung projektspezifischer und Off-the-Shelf Cartridges

Abbildung 3 zeigt das Prinzip. Hier wird im Rahmen des Projektes eine Cartridge erstellt, die projektspezifische Modelle einliest und mittels Modell-zu-Modelltransformationen die von den bereits vorhandenen Zielplattform-Cartridges erwarteten Modelle erzeugt. Diese werden dann verwendet um den Code für die betreffende Plattform zu erstellen. Im hiesigen Szenario wären beispielsweise zwei Off-the-Shelf Cartridges verwendbar, nämlich die für Spring und Hibernate. Die projektspezifische Cartridge erstellt den Input für diese und auch den projektspezifsichen Glue Code, der nötig ist, damit die von den Off-the-Shelf Plugins generierten Artefakte auch zusammen spielen.

Allerdings ist ein derartiges Vorgehen noch nicht ohne weiteres möglich. Es fehlen standardisierte Modelltransformationssprachen, qualitativ hochwertige Cartridges die genau einen Aspekt eines Systems adressieren und trotzdem mit anderen kombinierbar sind. Der Ansatz der Kaskadierung ist aber trotzdem sinnvoll und durchaus auch praktikabel anwendbar, wenn man nicht erwartet, dass man alles Off-the-Shelf bekommt.

Links & Literatur

[1] Eberhard Wolff: Spring Artikel Serie, Java Magazin 04-07/05

[2] http://www.omg.org/mda

[3] Stahl, Völter: Modellgetriebene Softwareentwicklung, dPunkt 2005

[4] Markus Völter, Software Architecture Patterns, http://www.voelter.de/data/pub/ArchitecturePatterns.pdf

[5] Markus Völter, Kaskadierung von MDSD und Modelltransformationen, http://www.voelter.de/data/articles/CascadingAndMT.pdf

Autoren

Peter Friese (*peter.friese@LHsystems.com*, *http://f3.tobject.de*) arbeitet als Software Architekt bei dem IT Dienstleister einer Firma, deren Geschäft das Fliegen ist. Er ist AndroMDA Committer und mitverantwortlich für die AndroMDA Spring Cartidge.

Markus Völter (*voelter@acm.org*) ist passionierter Segelflieger. Außerdem ist er freiberuflich als Berater im Bereich Softwarearchitektur und Modellgetriebener Software-Entwicklung tätig und Comitter von openArchitectureWare.

Eberhard Wolff (*Eberhard.Wolff@saxsys.de*) hat (fast) keine Angst vorm Fliegen und ist als Chef Architekt bei der Saxonia Systems AG tätig. Hauptsächlich beschäftigt er sich dabei mit Java Server Anwendungen.