Lebendige Architekturen

Markus Völter, voelter@acm.org, www.voelter.de

Dass die Architektur eines Softwaresystems maßgeblich über dessen Qualität und Wartbarkeit entscheidet ist wohlbekannt. Wie eine gute Softwarearchitektur aussieht ist (spätestens nach Ende des Projektes) meist auch klar. Wie man aber im Laufe eines – meist unter Zeitdruck stehenden – Projektes dafür sorgen kann, dass eine Architektur konsequent umgesetzt wird, und wie man die Architektur eines Systems im Laufe der Entwicklung praktisch weiterentwickelt ist nicht so klar. Dieser Artikel soll einige Anregungen in dieser Hinsicht geben.

Einleitung

Es hat sich mittlerweile herumgesprochen, dass Softwarearchitektur ein wichtiges Thema ist. Ein halbwegs großes System welches nicht auf eine „vernünftige" Architektur fußt wird nicht funktionieren. Nun sind in diesem Satz mindestens zwei sehr weiche Begriffe enthalten: *vernünftig* und *funktionieren*. Was ist eine vernünftige Architektur, und wann funktioniert sie?

An Softwaresysteme werden in der Regel bestimmte funktionale und nicht-funktionale Anforderungen gestellt. Primärer Zweck einer Softwarearchitektur ist es, diese Anforderungen realisieren zu helfen. Wenn sie das mit akzeptablem Aufwand erreicht, ist sie vernünftig. Nun sind die nicht-funktionalen Anforderungen an ein System typischerweise abhängig von der Anwendungsdomäne des Systems. Beispiele sind Performance (Antwortzeiten/Durchsatz), Skalierbarkeit, Ergonomie, Ressourcenverbrauch oder Echtzeitfähigkeit. Zusätzlich zu diesen (meist explizit geforderten) nicht-funktionalen Anforderungen gibt es meist noch weitere, die mehr oder weniger als selbstverständlich gelten, die aber nicht einfacher zu realisieren sind. Dazu gehört Wartbarkeit/Weiterentwickeltbarkeit (also die Fähigkeit des Systems, neue Anforderungen funktionaler und nicht-funktionaler Art abdecken können) sowie Testbarkeit (ein System welches nicht sinnvoll getestet werden kann, wird wahrscheinlich nie richtig funktionieren.) Ein weiterer wichtiger Aspekt ist der, den ich hier mal als „Programmierbarkeit" bezeichne: Die Entwickler müssen in der Lage sein, die Architektur in der Praxis umzusetzen. Wird die Architektur nicht richtig umgesetzt, werden die versprochenen Eigenschaften eben nicht erreicht, egal wie gut die Konzepte sind – da kann sicherlich jeder Entwickler entsprechende Anekdoten liefern. Es ist also essentiell, dass die Architektur auch „gelebt" wird.

In kleinen Projekten mit einer handvoll meist guter, Leute ist dies recht einfach. Alle verstehen die Randbedingungen die die Architektur adressiert, und wenn etwas unklar ist fragt man über den Tisch hinüber und klärt es. Wenn nun aber 20, 50, 200 oder 1000 Entwickler auf Basis der Architektur entwickeln sollen wird die Sache erheblich komplizierter. Der vorliegende Artikel soll einige Anregungen geben, wie man trotzdem zu einer lebendigen Architektur kommt.

Einfache und passende Architektur

Zunächst zwei Definitionen: ich unterscheide zwischen der technischen und der fachlichen Architektur. Die fachliche Architektur definiert fachliche Konzepte und Mechanismen, die fachliche Modularisierung, Abhängigkeiten, etc. Die technische Architektur beschreibt ist Umsetzung auf einer bestimmten technischen Basisplattform (Middleware). Die technische Architektur sollte idealerweise „hinter der fachlichen versteckt" sein – für die fachlichen Entwickler also nicht relevant.

Bevor wir uns damit beschäftigen, wie man eine Architektur in der Praxis lebendig halten kann, sei zunächst ein Grundsatz angemahnt: Je einfacher und passender eine Architektur ist, desto einfacher ist es, sie in der Praxis umzusetzen. In diesem Zusammenhang ist es nützlich, einen weiteren Begriff einzuführen: das Programmiermodell. Das Programmiermodell ist die Sicht, die der Entwickler auf die Architektur hat, wie die Architektur also zur Anwendungsentwicklung zu verwenden ist. Idealerweise ist das Programmiermodell einfach und intuitiv, selbst wenn die Architektur *intern* eine gewisse, vielleicht unvermeidbare Komplexität aufweist.

Es ist also vor allem wichtig, ein einfaches und für die Entwickler der Fachlichkeit passendes Programmiermodell zu definieren. Natürlich sind die beiden Aspekte nicht unabhängig. Man kann für eine komplexe Architektur nicht ohne weiteres ein einfaches Programmiermodell definieren. Auf der anderen Seiten gibt es aber auch keinen Grund, beispielsweise Fachentwickler im Bankenumfeld mit EJBs, Deployment-Deskriptoren und Applikationsserver-Deployment zu befassen.

Was zeichnet also nun eine passende und einfache Softwarearchitektur aus? Zunächst muss sie eine begrenzte Menge wohl definierter Konzepte enthalten. Für ein bestimmtes fachliches Konzept/Problem sollte es *einen* (keine fünf!) wohl definierten Weg geben, das Problem (oder die Klasse von ähnlichen Problemen) umzusetzen. Außerdem ist es wichtig, dass diese begrenzte Zahl wohl definierter Konzepte die Probleme der Fachdomäne tatsächlich effizient löst. Anregungen in dieser Richtung finden sich beispielsweise in den POSA-Patterns [BMRSS96] und bei Eric Evans [EE03]. Schlussendlich sollte man noch anmerken dass große Systeme oft aus verschiedenen Subsystemen bestehen, die sinnvollerweise auf verschiedenen Architekturen basieren können – es macht also Sinn, verschiedene technische Domänen im Rahmen eines Systems zu definieren, die dann jeweils mit einer passenden Architektur angegangen werden. Auch verschiedene Granularitätsstufen zu unterscheiden macht Sinn: Auf Makro-Ebene kann man beispielsweise eine Komponentenarchitektur verwenden. Innerhalb von Komponenten kann beispielsweise Pipes & Filters (für Bildverarbeitungskomponenten) oder MVC (für GUIs) verwendet werden.

Noch eine Randbemerkung sei mir gestattet: Zu einer guten Architektur kommt man immer durch Erfahrung! Man kann von den Erfahrungen anderer profitieren (siehe Patterns), man kann Erfahrungen aus früheren Projekten übernehmen und man kann natürlich die Architektur im Rahmen des Projektes iterativ weiterentwickeln [RL04]. Im Rahmen von Softwaresystemfamilien und Produktlinien [BKPS04] ergeben sich außerdem

Möglichkeiten zur systematischen Evolution einer Architektur über mehrere Projekte. Eine Architektur entsteht nie dadurch, dass ein „Zertifizierter Softwarearchitekt“ Powerpoint startet, und, na ja, sie wissen schon…. (siehe [MV02])!

Das wichtigste Architekturprinzip…

… ist meines Erachtens *Separation of Concerns*. Nur wenn man verschiedene Belange in einem System vernünftig trennt, kann man sie vernünftig adressieren. Separation of Concerns erlaubt es insbesondere, Dinge, die für die Anwendungsentwickler irrelevant sind, aus dem Programmiermodell herauszufaktorisieren. All die Dinge, die für die Programmierung der reinen fachlichen Anforderungen nicht essentiell sind, sollten sie auch tatsächlich ignorieren können. Dies macht den Entwicklern die Arbeit – also die Konzentration auf die Fachlichkeit – erheblich einfacher. Da die allermeisten dieser querschnittlichen Belange technischer Natur sind, bedeutet dies, dass die technische Architektur möglichst alle dieser querschnittlichen Belange adressieren sollte.

Lebendige Architekturen

Die obigen Erläuterungen stellen sozusagen die *notwendigen* Voraussetzungen für eine gelebte Softwarearchitektur dar. Was sind nun aber die *hinreichenden*? Meines Erachtens braucht es zur Lösung dieses Problems sowohl technologische als auch prozessuale Herangehensweisen, die ich im Folgenden erläutern möchte.

Technologische Aspekte

Zunächst ist es entscheidend, dass man die Anwendungslogik konsequent trennt von der Plattform gegen die entwickelt wird. Die Plattform enthält (fachliche und technische) Basisfunktionalitäten die wieder verwendet werden. Sie stellen daher ein wertvolles Asset dar. Wenn man seine Applikationen auf ausgereiften technischen Plattformen wie J2EE (plus z.B. Struts) aufbaut, ist diese Trennung schon fast zwangsläufig vorgegeben. Allerdings ist dies in den seltensten Fällen ausreichend. Technische Plattformen werden im Rahmen einer Architektur meist nur in einer bestimmten Art und Weise verwendet. Für diese Verwendungen sollten man zusätzliche Hilfsmittel entwickeln, typischerweise kleine Frameworks oder Bibliotheken. Auch verwendet eine Anwendung (oder eine Anwendungsfamilie) meist bestimmte fachliche Basisfunktionalitäten immer wieder. Auch hier sollte man sich wieder verwendbare und getestete Hilfsmittel vorhalten. Es ergibt sich damit eine fachliche Basis zur Entwicklung von Anwendungen [EE03]. Abb. 1 zeigt eine typische Referenzarchitektur.

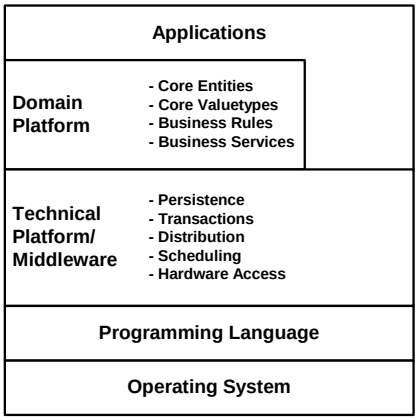

Abbildung 1: Verhältnis Plattform/Anwendung

Eine Plattform kann außerdem bereits bestimmte querschnittliche Aspekte adressieren. Dies kann mit den bekannten architekturellen Mitteln passieren; beispielsweise mittels des Interceptor-Patterns [SSRB00]. Beispiele im Java-Umfeld sind J2EE Applikationsserver (die sich ja grade um querschnittliche technische Belange wie Transaktionen, Lastverteilung und Sicherheit kümmern) oder Servlet-Filter. Anwendungsentwickler müssen sich um diese Aspekte bei der Entwicklung der Fachlogik nicht mehr kümmern.

In den Fällen wo man nicht auf eine ausgereifte technische Plattform aufbaut, bzw. die Aspekte, die die Plattform adressiert nicht die „richtigen" sind, muss man selbst Hand anlegen. Das bedeutet, dass man in der eigenen Architektur Hooks vorsehen muss mittels derer man querschnittliche Belange lokalisieren kann – verschiedene Patterns wie z.B. Proxy [GHJV95], Interceptor [SSRB00], Invocation Context [VSW02], etc. können hier helfen. Reflection bietet in Java außerdem weitere Möglichkeiten: damit lassen sich beispielsweise dynamisch Proxies erzeugen, in die man Interceptoren einhängen kann.

Reicht dies alles nicht aus (oder ist es zu kompliziert oder inperformant) so wird man auf die Mittel der Aspektorientierten Programmierung[1] zurückgreifen. Mit einem Tool wie AspectJ lassen sich Aspekte auf praktisch beliebig feiner Granularität definieren. AOP erfordert zwar ein radikales Umdenken (bzw. Dazulernen); hat man es aber erst einmal verstanden eröffnen sich gerade im Bezug auf „Lebendige Architekturen" völlig neue Möglichkeiten und man fragt sich, wie man jemals ohne auskam.

Mit Blick auf das Programmiermodell haben wir also bereits einige Vereinfachungen erreicht:

[1] Ich unterscheide zwischen Aspektorientierung im Allgemeinen und der Aspektorientierten *Programmierung* (siehe [MV03]. Im letzteren Fall werden Aspekte auf dem Level der Programmiersprache behandelt, wohingegen AO auch die Behandlung von querschnittlichen Belangen z.B. auf Ebene des Design und der Architektur einschließt. Oben erwähnte Plattformen und Patterns Beispiele für AO auf dieser Ebene.

- Technische Aspekte erledigen die technische Plattform und die Aspekte (im Sinne von AOP); Die technische Plattform bietet generell Wiederverwendungspotentiale
- Die fachliche Plattform stellt eine Basis für die Vereinfachung der fachlichen Programmierung im Sinne von Eric Evans's Domain Driven Design [EE03] dar.

Es lassen sich mit diesen Mitteln allerdings auch nicht alle problematischen Aspekte des Programmiermodells vermeiden, teils auch aufgrund von Limitierungen der Programmiersprache Java. Als Beispiel dient wieder einmal J2EE: Auch wenn die Entwicklung von Enterprise Anwendungen ohne Applikationsserver als technische Plattform definitiv komplexer wäre, so ist das J2EE Programmiermodell nicht wirklich intuitiv oder gar einfach.

Eine weitere signifikante Verbesserung des Programmiermodells – die Basis für „lebendige Architekturen" – kann nur durch konsequente Abstraktion und domänenspezifische Konzepte erreicht werden. Modellgetriebene Softwareentwicklung [SVB04] ermöglicht genau dies. Der Kern des Modellgetriebenen Ansatzes besteht aus den folgenden Punkten:

- Der Entwickler spezifiziert die Anwendungslogik mittels Domänenspezifischen (Modellierungs-)Sprachen; diese Modelle beschreiben fachliche Konzepte der Anwendungsdomäne
- Zur Modellierung wird eine Domänenspezifische Sprache verwendet. Diese basiert auf den Konzepten der fachlichen Plattform.
- Mittels Transformationen werden diese Modelle abgebildet auf den Implementierungscode auf der betreffenden Plattform.

Durch die Verwendung von domänenspezifischen Sprachen lassen sich „Programme" um einiges knapper und prägnanter formulieren. Der Aufwand, dies in den Implementierungscode auf der betreffenden Zielplattform zu überführen wird durch Generatoren weitestgehend automatisiert. In diesen Transformationen stecken also die Regeln im Umgang mit der Architektur (Plattform). Der Begriff Programmiermodell muss also nun weiter untergliedert werden (siehe Abb. 2):

- Das Wissen über das fachliche Programmiermodell bzw. den „richtigen" Umgang der (technischen) Plattform steckt in den Transformationen. Weil diese „ausführbaren Code" darstellen, ist sichergestellt, dass das Programmiermodell der Plattform korrekt umgesetzt wird.
- Der Entwickler bekommt ein viel einfacheres, domänenspezifisches Programmiermodell welches darin besteht, dass er mittels der DSL die Anwendungslogik „programmiert". Gegebenenfalls werden einzelne Aspekte weiterhin direkt in der Implementierungssprache implementiert und mittels vorgegebener Patterns in den generierten Code eingehängt.

Die zweistufige Darstellung in Abb. 2 zeigt, dass modellgetriebene Codegenerierung und die fachlichen Plattformen zwei Seiten derselben Medaille sind und unzertrennlich zusammen gehören. Um es nochmals klar zu sagen: Die Transformationen vom Modell auf die Plattform definieren Sie selbst! Sie können damit ihr Wissen im Umgang mit der Architektur in diese Transformationen verpacken und im Rahmen der Anwendungsentwicklung tatsächlich verwenden. Wieviel sie dabei in der fachlichen Plattform realisieren, und wie viel sie direkt für die technische Plattform generieren, ergibt sich im Laufe des Projektes und wird sich auch immer wieder verschieben.

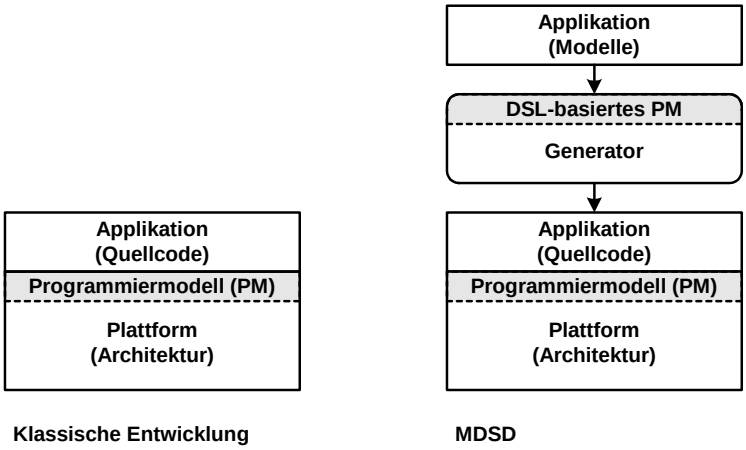

Abbildung 2: Klassische Entwicklung vs. MDSD

Ein weiterer wichtiger Grund warum MDSD das Programmiermodell für den Anwendungsentwickler signifikant verbessert ist der, dass der Generator vor der Codegenerierung das Modell (also ihr „Programm“) gegen die Regeln der Domäne bzw. Ihrer Architektur abprüft. Werden Verstöße erkannt, so werden diese als Generatorfehlermeldungen ausgegeben. Da Sie die Constraints der Domäne selbst in ihren Generator einpflegen, kann die Fehlermeldung sehr aussagekräftig sein; viel aussagekräftiger als die Compiler- bzw. Laufzeitfehler bei traditioneller Entwicklung. Der Grund hierfür liegt darin, dass die Modelle um einiges mehr Semantik enthalten als typischer 3GL Code. Versuchen Sie einmal, mittels Frameworks die Constraint umzusetzen, dass Data Transfer Objects nur Getter und Setter, aber keine weiteren Methoden haben dürfen. Mittels MDSD ist das (z.B. unter Verwendung der openArchitectureWare Generators [oAW]) eine Sache von 5 Zeilen (Generator-)Code.

Zuletzt sei noch ein weiterer sehr wichtiger Aspekt erwähnt: Modellgetriebene Entwicklung erlaubt insbesondere die einfache Architektur-Evolution. Wenn Sie feststellen, dass sich bestimmte (initiale) architekturelle Entscheidungen nicht bewährt haben, können Sie diese leicht ändern: in den meisten Fällen müssen Sie lediglich Ihre Transformationen anpassen und die Anwendung neu generieren (sowie ggfs. entsprechende Anpassungen an der Plattform durchführen). Durch diesen Ansatz wird vor allem die technische Architektur (in gewissen Grenzen) selbst zum Aspekt im Sinne

von AO: Sie ändern an einer Stelle und es wirkt sich querschnittlich aus. Die Fähigkeit eines Projektes die technische Architektur an neue Anforderungen (oder neue Erkenntnisse!) anzupassen ist absolut entscheidend für Lebendige Architekturen, zum Beispiel, wenn die Anforderungen an die Skalierbarkeit steigen. In vielen Projekten ist der Aufwand, Änderungen an der Architektur eines Softwaresystems durchzuführen [RL04] so hoch, dass man lieber Hacks einbaut oder mit einer schlechten Architektur lebt (oder zumindest so lange zu leben versucht bis es gar nicht mehr geht… Dann hat man allerdings ein dickes Problem!).

MDSD hat noch eine ganze Reihe weiterer Auswirkungen auf die Softwareentwicklung; [MV04] gibt einen Überblick.

Prozessuale Aspekte

Kein Problem in der Softwareentwicklung lässt sich durch Technologie und Tools alleine lösen. Der Entwicklungsprozess muss passen; im Kontext dieses Artikels ist es insbesondere wichtig, dass ein Prozess etabliert wird, der die oben geschilderte Herangehensweise unterstützt.

Im Rahmen der oben erläuterten Vorgehensweise ist es zunächst einmal essentiell, dass man Plattformentwicklung (ggfs. unter Einschluss von Generatoren, DSLs, etc.) konsequent von der Anwendungsentwicklung trennt, denn die beiden Entwicklungsstränge haben verschiedene Ziele:

- Die Plattformentwickler erstellen wieder verwendbare, querschnittliche Infrastruktur die die Umsetzung der nicht-funktionalen Anforderungen garantiert und zentrale fachlichen Konzepte umsetzt. Die Plattform ist idealerweise über mehrere Anwendungen in der gleichen Domäne wieder verwendbar.
- Die Anwendungsentwickler bauen konkrete Anwendungen die funktionale Anforderungen des Kunden umsetzen. Wiederverwendung ist hier auf Anwendungsebene nicht essentiell; die wieder zu verwendenden fachlichen Konzepte finden sich im fachlichen Teil der Plattform.

Die beiden Stränge zu trennen heißt natürlich nicht, dass diese nicht miteinander koordiniert werden müssen; ganz im Gegenteil. Zum einen sind natürlich auch die nicht-funktionalen Anforderungen zur Zufriedenheit des Kunden umzusetzen. Zum anderen hilft es auch nichts, wenn die Plattformentwickler eine „coole“ Plattform bauen, die aber leider nicht den Erfordernissen der Anwendungsentwickler entspricht. Die Mitarbeiter die die Plattform bauen müssen sowohl technische Kompetenz besitzen; sie müssen aber auch die Domäne sehr gut verstehen, da sie ja die zentralen fachlichen Konzepte und Mechanismen umsetzen sollen, bzw. eine für die fachliche Arbeit nützliche DSL definieren sollen. Um all diese Ziele zu erreichen, sind die folgenden agilen Prozessbausteile essentiell:

- Iterative Entwicklung mit wohl definierten Timeboxen, Releases und Meilensteinen sorgt dafür, dass der Kunde fühestmöglich nützliche Software bekommt.
- On-site Customer stellt sicher, dass der Kunde früh in die Evaluierung der erstellten Software eingebunden wird. Im Falle der Verwendung einer geeigneten DSL kann der Kunde sogar in die fachliche Entwicklung eingebunden werden.
- Scope Trading zwischen Kunden und Entwicklern sorgt dafür, dass die wichtigsten Features implementiert werden
- Die Anwendungsentwickler spielen die Rolle des Kunden bzgl. der Plattform! On-Site Customer und Scope Trading ist auch in dieser Kunde-Lieferant Beziehung essentiell. Wenn mehrere Anwendungen parallel auf derselben Plattform entwickelt werden, so gibt es eben mehrere Kunden für den Plattformstrang.

Es hat sich außerdem bewährt, Anwendungsentwickler zeitweise in die Plattformabteilung „abzuordnen", um die Bodenhaftung nicht zu verlieren und allen Mitarbeitern direkten Einfluss auf die Weiterentwicklung der Plattform zu erlauben.

Fazit

Die oben erläuterten technischen und prozessualen Aspekte können sicherstellen, dass Softwarearchitektur erstens den Stellenwert bekommt den sie verdient, zweitens nicht auf Powerpoint-Folien beschränkt ist, drittens im Projekt gelebt wird und sich viertens im Laufe des Projektes weiterentwickeln kann. Keines der erläuterten Konzepte ist wirklich neu: die Kombination ist aber extrem nützlich und praxistauglich.

Referenzen

[BKPS04] Böckle et. al. (Hrsg.), Software Produktlinien, dPunkt, 2004

[BMRSS96] Buschmann et. al., Pattern Oriented Software Architecture, Vol. 1, Wiley 1996

[EE03] Eric Evans, Domain Driven Design, Addison-Wesley 2003

[GHJV95] Gamma et. al., Design Patterns, Addison-Wesley 1995

[MV02] Markus Völter, Hope, Belief and Wizardry – three perspectives on project management, http://www.voelter.de/data/pub/hbw.pdf

[MV03] Markus Völter, Handling Cross-Cutting Conerns: AOP and beyond, http://www.voelter.de/data/pub/AspectLevels.pdf

[MV04] Markus Völter, Modellgetriebene Softwareentwicklung, ObjektSpektrum 04/2004

[oAW] openArchitectureWare Generator, http://sourceforge.net/projects/architecturware/

[RL04] Stefan Roock, Martin Lippert; Refactorings in großen Softwareprojekten, dPunkt 2004

[SSRB00] Schmidt et. al., Pattern Oriented Software Architecture, Vol. 2, Wiley 2000

[SVB04] Stahl, Völter, Bettin, Modellgetriebene Softwareentwicklung, dPunkt 2004 (noch nicht veröffentlicht)