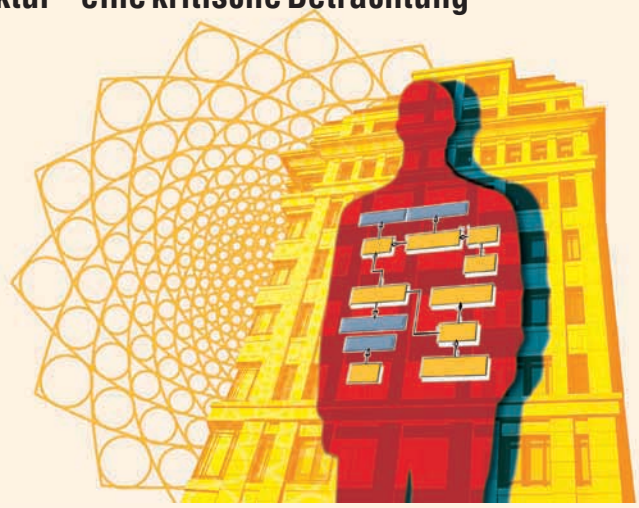
Jenseits des Tellerrands, Teil 4: Softwarearchitektur – eine kritische Betrachtung

Architektur ohne Hype



VON MARKUS VÖLTER UND ARNO HAASE

Speaker @ w-jax

Softwareentwicklung ist weit mehr als Programmierung und Programmierung ist weit mehr als das bloße Beherrschen von Programmiersprachen und Werkzeugen. Die Artikel dieser Serie greifen wichtige Themen, aktuelle Fragen und auch grundsätzliche Probleme auf, beleuchten diese und stellen „die gängigen Antworten" immer wieder in Frage. Subjektivität ist dabei keineswegs verpönt, sondern das notwendige Salz in der technischen Einheitssuppe.
Allein die Erwähnung des Wortes „Architektur" führt bei vielen Entwicklern zu heftigen Reaktionen. Während sich die einen um die richtige Definition des Begriffs streiten, diskutieren die anderen die Sinnhaftigkeit der implizierten Analogie zwischen dem Bauen von Häusern und dem Erstellen von Software. Unbestreitbar ist jedoch, dass „Softwarearchitektur" zunehmend inflationär gebraucht wird: Jeder Programmierer mit einem abgeschlossenen Projekt ist heutzutage „Softwarearchitekt" und jedes Sammelsurium zahlloser Technologien wird zur „Standardarchitektur" erklärt. Markus Völter und Arno Haase versuchen im Folgenden, das ganze Blendwerk abzureißen und die Frage nach dem Wesentlichen bei der Erstellung und Umsetzung einer Architektur zu beantworten.

Johannes Link
(johannes.link@andrena.de)

Softwarearchitektur ist heutzutage ein sehr stark von Hypes und Technologien geprägtes Thema. Man redet von Java EE-Architekturen oder Web-Services-Architekturen. Dies hat jedoch eine ganze Reihe von Nachteilen. Wir zeigen im Folgenden diese Nachteile auf und stellen einen Ansatz vor, der das Thema Softwarearchitektur etwas systematischer und konzeptioneller und weniger technologiezentriert angeht. Dabei kommt modellgetriebene Softwareentwicklung zum Einsatz.

Software(-entwicklung) ist kompliziert. Dies liegt unter anderem daran, dass man es nicht schafft, Technologiedetails zu isolieren – sie nerven die Entwickler während des gesamten Softwareentwicklungsprozesses. Die starke Fokussierung auf (aktuelle) Hypes hat zur Folge, dass man sich schwer tut, die Substanz vor lauter bunten Wörtern zu erkennen. Viele aktuelle Hypes sind weder genau definiert noch sind sie essenziell neu. Man fange nur mal eine Diskussion an, was der Unterschied zwischen einer serviceorientierten Architektur und einer (guten) komponentenbasierten ist ... Sinnvolle Diskussionen über die Frage, wie das System zu realisieren ist, werden abgeblockt, weil ständig unklar definierte Buzzwords in die Diskussion geworfen werden.

Ein verwandtes Problem ist, dass man sich zu stark an Standards orientiert. Heutzutage gibt es ja zu allem und jedem mindestens einen Standard. Standards haben allerdings die Eigenschaft, dass sie oft eine Obermenge aller möglichen Lösungen eines Problems darstellen oder zumindest eine allumfassende. Im Gegensatz zu früher (da war ja alles besser ...) werden Standards heute oft nicht mehr definiert, *nachdem* sich eine Lösung als gut herausgestellt hat, sondern Standards sind der größte gemeinsame Nenner der zwar existierenden, aber in der Praxis noch nicht getesteten Lösungen eines Problems verschiedener Anbieter. Im Extremfall entstehen die Standards sogar, bevor es überhaupt Produkte im fraglichen Bereich gibt. Die Verwendung eines Standards kann daher zu großen Schwierigkeiten führen, da er für die jeweilige Situation möglicherweise ungeeignet ist und das System damit komplizierter macht, als es nötig wäre.

Dies alles hat zur Folge, dass statt Architektur- Technologiediskussionen geführt werden und der Blick für die im System zu verwendenden Architekturkonzepte und Ansätze verloren geht. Die Architektur wird zu stark an die Technologien gebunden, die Software wird

Jenseits des Tellerrands – die Serie

Was bisher geschah und Sie zukünftig erwartet

- Teil 1: Das nächste Java?
- Teil 2: Typisierung in Java und darüber hinaus
- Teil 3: Warum gibt es eigentlich Softwareentwicklungsprojekte?
- **Teil 4: Softwarearchitektur: eine kritische Betrachtung**
- Teil 5: Textuelle domänenspezifische Sprachen
- Teil 6: Moderne System- und Abnahmetests

schwer zu testen und zu warten. Das Programmiermodell wird übermäßig kompliziert, da immer und überall Technologie durchscheint. Die Entwicklung ist zu teuer und die Entwickler müssen alle möglichen Technologien kennen, was sie natürlich nicht tun, was dann wiederum zu weiteren Problemen führt. Es ist außerdem schwierig, das System an sich weiterentwickelnde Technologien anzupassen – beispielsweise um von besserer Skalierbarkeit oder Wartbarkeit einer neuen Plattform profitieren zu können. Das muss sich ändern.

Ein weiteres Problem in diesem Zusammenhang ist, dass es in größeren Projekten mit mehr als zwanzig Entwicklern fast unmöglich ist, mit klassischen Mitteln die Architektur „am Leben" zu halten – also dafür zu sorgen, dass sie vom gesamten Team richtig und konsequent umgesetzt wird. Dies liegt an verschiedenen Dingen (zu wenig Kommunikation, Kompetenz, andere politische Schwerpunkte, Ignoranz) und wird potenziell zu einem großen Problem: Um die Versprechen einer Architektur einlösen zu können, ist es wichtig, dass die Architektur konsequent und richtig umgesetzt wird.

Es ist uns wichtig, etwas klarzustellen: Wir argumentieren hier *nicht* dafür, dass ein „Architekt" sich eine Architektur ausdenkt, welche die Entwickler dann, ohne darüber nachzudenken, dogmatisch umsetzen. Vielmehr geht es darum, dass man – wenn man eine architekturelle Entscheidung getroffen hat – diese dann konsequent, durchgängig und gleichförmig umsetzt, *bis* sie sich als Fehler herausstellt. Dann wird die Entscheidung gegebenenfalls revidiert oder angepasst, aber danach wird die *neue* Entscheidung konsequent umgesetzt. – Ob man dieses Refactoring auf einmal macht oder schrittweise (wenn man sowieso gerade an der betreffenden Stelle im Code zu tun hat) hängt von dem verwendeten Vorgehen ab. Wichtig ist eben nur, dass man sicherstellen kann, dass das Refactoring überall entsprechend durchgeführt wird.

Wie man zu solchen architekturellen Entscheidungen kommt, sei an dieser Stelle nicht das Thema. Sinnvollerweise tut man das im Team und iterativ.

Programmiermodell

Der Begriff Programmiermodell hat nichts mit „Modellen" im Sinne von UML oder MDA zu tun. Insofern ist der Begriff in diesem Zusammenhang etwas ungeschickt – er hat aber bereits eine sehr lange Tradition. Das Programmiermodell beschreibt, wie man basierend auf der Architektur Anwendungen baut. Es definiert also APIs und Idiome im Umgang mit diesen, beantwortet also die Frage *„wie* setze ich folgende Anforderung um" aus Sicht des Anwendungsentwicklers. Eine Architektur(-beschreibung) beantwortet ja auch die Fragen *„warum* lösen wir die Anforderung XY so oder so" bzw. *„warum* nicht anders". Das Programmiermodell ist der für den Anwendungsentwickler sichtbare Teil der Architektur.

Was ist Architektur?

Bevor wir den Lösungsansatz im Detail betrachten, möchten wir noch kurz etwas zu dem Begriff Architektur sagen. Definitionen des Begriffes gibt es viele, in der Praxis nützliche allerdings wenige. Eine Sache, die immer wieder zu Diskussionen führt und hier besonders wichtig ist, ist die folgende, zunächst beschrieben anhand zweier Beispiele:

- Architektur bestimmt, welche Arten von Bausteinen der Anwendungsprogrammierer zur Verfügung hat, um eine Anwendung zu bauen. Beispielsweise wird definiert, dass es Komponenten gibt, wie diese miteinander reden, welche Eigenschaften Interfaces haben etc.
- Auf Basis der Architektur definiert das Anwendungsdesign oder die Anwendungsstruktur die konkreten Ausprägungen obiger Bausteine, aus denen eine Anwendung besteht; also beispielsweise, dass es die Komponente „Bestellannahme" gibt, die mit Komponenten „Kundenverwaltung" per Interface „Kundenabfrage" redet.

Beide Aspekte sind wichtig. Architektur ist eher technisch, Anwendungsdesign eher fachlich. Die Architektur definiert das Vokabular des Anwendungsdesigns. (Hinweis für „in Modellen Denkende": Aspekt *eins* ist das Metamodell, Aspekt *zwei* ein Modell – dazu später mehr): Das Anwendungsdesign kann aber erst festgelegt werden, wenn die Architektur definiert wurde. Das kann man daran erkennen, dass im zweiten Satz die Begriffe vorkommen, die im ersten definiert wurden – Komponente, Interface. Wenn diese nicht definiert sind, kann ich den zweiten Satz nicht formulieren. Natürlich kann man auch erst mal einen experimentellen Spike durchführen, um sich über die Konzepte klar zu werden. Außerhalb dieses Artikels wird übrigens oft auch das Anwendungsdesign als Architektur bezeichnet.

Lösungsansatz

Um die oben geschilderten Probleme in den Griff zu bekommen, gibt es verschiedene Möglichkeiten – aus unserer Erfahrung allerdings nicht sehr viele, die wirklich funktionieren und mit der Projektgröße skalieren. Wir möchten hier einen Ansatz vorstellen, der in verschiedenen Umfeldern (Web, Embedded, Prozesssteuerung) und in Projekten verschiedener Größen umgesetzt wurde und sich dabei wirklich bewährt hat. Der Ansatz untergliedert sich in drei Phasen und – pro Phase – in mehrere Schritte. Es versteht sich von selbst, dass man diese Schritte iterativ und inkrementell ausführt, auch wenn sie hier sequenziell beschrieben sind. Übrigens ist dieser Artikel aus einer Mustersprache (Pattern Language) extrahiert [1], welche auch ein ausführliches Beispiel enthält. Das Beispiel ist separat auch in [2] zu finden.

Phase 1: Elaboration

In dieser Phase geht es darum, sich über die zu bauende Architektur klar zu werden. Diese Phase sollte von einer kleinen, überschaubaren Anzahl Leuten, einer Art Kernteam, durchgeführt werden. Erst nach dieser Phase sollte die Architektur in einem größeren Umfeld verwendet werden. Dies ist wichtig, weil die Erfahrung zeigt, dass exploratives, problemlösendes Arbeiten mit einem kleinen Team effizienter ist – Architekturentwicklung lässt sich nur schwer parallelisieren. Wenn während Phase 1 schon zu viele Menschen im Projekt sind, entsteht automatisch ein ungesunder Druck, vorschnell – also ohne Architektur – mit der Implementierung der Anwendung zu beginnen, weil ja sonst

ein Teil des Projektteams „unproduktiv herumsitzen“ würde.

Schritt 1: technologieunabhängige Architektur. Es ist wichtig, dass sich die Architektur gut an alle Stakeholder kommunizieren lässt und die Entwickler die fachlichen Anforderungen leicht – also möglichst ungestört von technologischen Aspekten – implementieren können. Außerdem soll die Architektur im Projektalltag durchgehalten werden. Dafür ist es auch wichtig, dass die Architektur die klassischen Hype-Zyklen überstehen kann. Schlussendlich muss man in der Lage sein, das System bzgl. nicht funktionaler Anforderungen weiterzuentwickeln, also z.B. zu skalieren.

Es macht daher Sinn, die Architekturkonzepte zunächst unabhängig von ihrer technologischen Umsetzung zu definieren. Ein Glossar oder ein Architekturmetamodell (s.u.) kann dabei behilflich sein. Erst in einem späteren Schritt (s.u.) wird die Abbildung auf technische Plattformen definiert. Hier wird man übrigens auf die wohlbekannten Architektur-Patterns und -stile zurückgreifen, um einen Startpunkt für die Definition der eigenen Architektur zu haben [3] [4]. Auch diese sind rein konzeptionell und nicht technologiespezifisch. Konsequenzen dieses Ansatzes sind u.a. eine einfachere – weil technologiefreie – Beschreibung der Architektur.

Fragt sich, ab wann ein Lösungsansatz „Technologie“ wird. Wie viel Technologie steckt in einer technologieunabhängigen Architektur? Unsere Abgrenzung ist dabei dergestalt, dass alles erlaubt ist, was zusätzliche Ausdrucksmöglichkeiten bringt. Objekte, Komponenten, Aspekte, Dependency Injection beziehungsweise Message Queues sind solche Kandidaten.

Übrigens wird man hier zur Dokumentation oft Modelle einsetzen – allerdings informelle. Box-and-Line-Diagramme mit Visio oder PowerPoint bzw. Whiteboard und Digitalkamera leisten dabei gute Dienste zur Illustration und Kommunikation der Architekturkonzepte. Dabei ist zu beachten, dass man – auch wenn die Diagramme nicht formal sind – trotzdem definieren muss, was die Kästchen und Linien bedeuten („Die Kästchen stellen Komponenten dar, die Linien bedeuten eine ‚verwendet‘-Beziehung“).

Schritt 2: Programmiermodell. Nachdem wir die Architektur nun konzeptionell definiert haben, ist der nächste Schritt die Festlegung, wie die Entwickler mit der Architektur umgehen sollen, wie also die fachlichen Anforderungen basierend auf der Architektur umgesetzt werden. Oft ist die Architektur nämlich nicht einfach zu verstehen bzw. zu verwenden, da sie nicht funktionale Anforderungen berücksichtigen muss. Bei der Definition des Programmiermodells muss sichergestellt werden, dass die Architektur von den Entwicklern „richtig“ verwendet wird. Falsche Verwendung sollte durch das Programmiermodell so weit als möglich ausgeschlossen werden. Der entstehende Anwendungscode als programmiersprachliche Umsetzung des Anwendungsdesigns sollte sich dabei möglichst leicht lesen lassen, da er fachlich gereviewt werden muss. Außerdem ist es wichtig, dass der Code testbar bleibt (s.u.). Das Programmiermodell sollte also möglichst einfach verständlich sein und die typischen Anwendungsfälle der Architektur (also typische Elemente des Anwendungsdesigns) möglichst einfach implementierbar machen. Die seltenen Fälle darf es nicht ausschließen. Das Programmiermodell muss gut dokumentiert werden, und zwar in Form eines Tutorials oder eines Walkthrough. Es sollte – wie bereits erwähnt – möglichst stark von der Technologie abschirmen. Die Technologieabbildung (s.u.) wird separat definiert. Es gibt dabei ein paar Richtlinien für die Definition des Programmiermodells, deren Einhaltung eine unten beschriebene Abbildung auf verschiedene Technologien erleichtern. Dazu zählen:

- immer gegen ein Interface entwickeln, nicht gegen Implementierungen
- immer Factories verwenden, um (wichtige) Objekte zu erzeugen, bzw. auf Dependency Injection zurückgreifen, um externe Konfigurierbarkeit zu ermöglichen
- immer mittels Factories auf Ressourcen zugreifen
- Stateless Design ist – zumindest in Enterprise-Systemen – meist eine gute Idee
- sicherstellen, dass ein Artefakt eine Sache tut, nicht fünf. AOP kann hier helfen (Beispiel: Eine Entität, die Daten ver

Anzeige

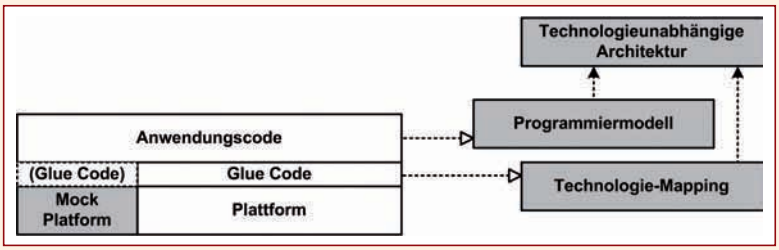


Abb. 1: Zusammenspiel/Struktur der Schritte von Phase 1

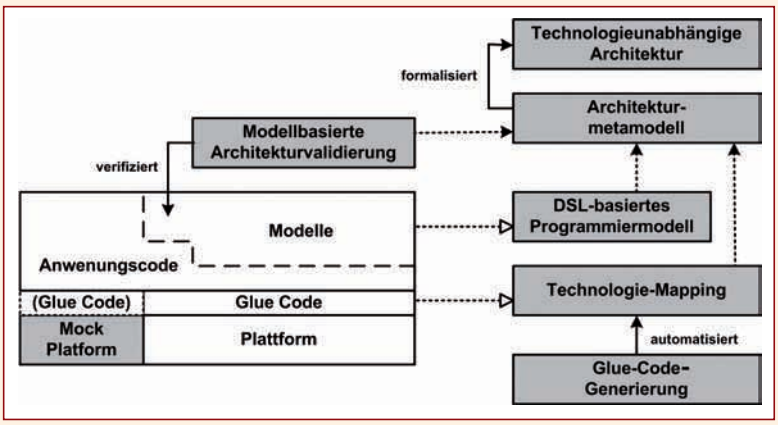


Abb. 2: Zusammenspiel/Struktur der Schritte von Phase 1 und 2

waltet, sollte nicht selbst für ihre Persistenz sorgen – der Persistenzaspekt sollte extern gehandhabt werden)

Bei der Definition des Programmiermodells – und auch bei der technologieunabhängigen Architektur – sind die Ansätze aus Eric Evans Buch „Domain Driven Design" [8] extrem nützlich.

Schritt 3: Technologieabbildung. Warum verwendet man überhaupt Technologien wie Java EE, Websphere, COM+, Websphere MQ oder CCM? Sicherlich nicht, weil sie so angenehm zu verwenden wären (oder in anderen Worten: wegen ihres tollen Programmiermodells). Sondern weil sie bewiesen haben, dass sie in der Lage sind, bestimmte nichtfunktionale Anforderungen wie Skalierbarkeit, Ausfallsicherheit, Konfigurierbarkeit, Managebarkeit oder Transaktionalität zuverlässig umzusetzen. Das will man in aller Regel nicht selbst nachbauen. Folglich definieren wir nun Abbildungsregeln, wie die Konzepte der technologieunabhängigen Architektur und des Programmiermodells auf bestimmte Plattformen abgebildet werden. Dabei bilden wir auf gerade die Plattform ab, welche die für uns nötigen nichtfunktionalen Eigenschaften aufweist.

Hier ist auch der Punkt, wo man über die Verwendung von Standards nachdenken sollte. Der Portabilität wegen ist es oft sinnvoll, standardisierte Plattformen als Abbildungsziel zu verwenden – wobei man auch hier vorsichtig sein muss: Es kommt praktisch nie vor, dass ein großes System auf eine andere Plattform portiert wird. Die Verwendung von Standards ist aber auch dann sinnvoll, wenn es um die Kompatibilität von Systemen geht (z.B. bei der Integration verschiedener Systeme oder bei der Frage, auf welche Infrastruktur sich ein Rechenzentrum zum Betrieb der Softwarelösungen einlässt). Allerdings sollte man bei der Verwendung von Standards immer im Hinterkopf behalten, dass sie Mittel zum Zweck (Kompatibilität, Integrationsfähigkeit, Know-how etc.) ist und nicht Selbstzweck.

Im Gegensatz zu den bereits weiter oben Verwendung findenden Architektur-Patterns ist nun auch die Zeit, technologiespezifische Design-Patterns und Idiome einzusetzen – z.B. die Java EE Blueprints [5]. Diese helfen bei der Beantwortung der Frage, wie man die Plattform richtig einsetzt.

Die explizite Definition der Abbildung auf die Zielplattform erlaubt es außerdem, den gleichen Anwendungscode auf mehreren Plattformen laufen zu lassen, was immer dann Sinn macht, wenn man die gleiche Fachlichkeit mit unterschiedlichen QoS-Eigenschaften betreiben muss, also zum Beispiel

- den Webshop mit einer reinen Spring-Lösung auf Tomcat/Apache im Intranet und
- geclustert, ausfallsicher, managebar für den Shop im Internet mittels EJB und Weblogic.

Eine weitere, besonders wichtige Plattform ist dabei die im nächsten Schritt vorgestellte.

Schritt 4: Mock-Plattform. Nachdem wir nun das Programmiermodell und eine (oder mehrere) Technologieabbildung(en) definiert haben, stellt sich die Frage, wie die Entwickler ihre Fachlichkeit testen können. Tests sollten möglichst schmerzfrei implementierbar und ausführbar und *nicht* auf große Mengen technischer Infrastruktur angewiesen sein. Bildlich gesprochen: Man soll keinen Application Server hochfahren müssen, nur um die korrekte Addition zweier Geldbeträge testen zu können. Dies ist essenziell, damit auch wirklich regelmäßig getestet wird.

Die Mock-Plattform ist daher das einfachst mögliche Technologie-Mapping. Ein einfaches Framework mockt die technische Infrastruktur, sodass der Anwendungscode läuft. Nichtfunktionale Anforderungen werden dabei komplett ignoriert und können mit den darauf laufenden Tests logischerweise auch nicht getestet werden. Das Programmiermodell muss dafür wirklich technologieunabhängig sein, sonst wird es nur schwer möglich sein, eine Mock-Plattform zu de-

Metamodelle

Ein Metamodell beschreibt die Bestandteile von Modellen. Wenn man Modelle „malen" möchte, braucht man dafür immer ein Metamodell, damit definiert ist, was man in das Modell reinmalen darf und was das dann bedeuten soll. Beispielsweise kann man in UML-Modellen das Konzept „Klasse" oder „Assoziation" verwenden. Das UML-Metamodell beschreibt nun formal, was eine „Klasse" oder eine „Assoziation" ist. Ein Architekturmetamodell beschreibt die Konzepte, die in der Architektur vorhanden sind. Dies ist nötig, da man in Phase 3 die Anwendungsstrukturen mittels Modellen beschreiben wird.

finieren. Hier zeigt sich also ein weiterer Vorteil der Technologieunabhängigkeit.

Schritt 5: Vertikaler Prototyp. Wir haben jetzt eigentlich alles zusammen. Die technologieunabhängige Architektur, welche die Konzepte definiert, das Programmiermodell, welches den Umgang mit der Architektur beschreibt, die Plattformabbildung(en) sowie eine Mock-Plattform für Unit-Tests. Nun müssen wir sicherstellen, dass all dies auch sinnvoll zusammen funktioniert. Zu diesem Zweck implementieren wir einen vertikalen Prototyp, also ein repräsentatives Stück des Anwendungsdesigns, welches alle Aspekte der vorangegangenen Schritte benutzt. Vor allem die „schwierigen" Aspekte der Architektur müssen repräsentativ implementiert werden.

Da man hier nun die „richtige" Plattform verwendet, kann man mittels des vertikalen Prototyps nun auch Tests der nichtfunktionalen Anforderungen durchführen – und das sollten wir auch ausgiebig tun, da wir vor allem dadurch erkennen, wenn wir eine falsche Plattform ausgewählt haben oder unsere Abbildungsregeln auf diese Plattform nicht oder nicht performant genug funktionieren. Also: Jetzt ist der Zeitpunkt für Lasttests und Performanceoptimierungen, nicht früher, aber auch nicht später. Als Folge dieser Tests wird man gegebenenfalls die vorigen Schritte noch mal wiederholen und das eine oder andere anpassen. Zu beachten ist, dass dieser Schritt nicht nur das Technologie-Mapping testet, sondern auch die Praktikabilität des Programmiermodells.

Phase 2: Iterieren

Überhaupt wird man die verschiedenen Aspekte vermutlich nicht beim ersten Mal richtig hinbekommen, weswegen man explizit Zeit einplanen sollte, die Schritte iterativ zu wiederholen, bis ein vernünftig funktionierender Stand erreicht ist – was man an einem leicht zu implementierenden sowie korrekt und performant funktionierenden vertikalen Prototyp erkennen kann.

Phase 3: Automatisieren

Nachdem in der ersten und zweiten Phase die Architektur also solche „festgelegt" wurde, geht es nun vor allem in größeren Projekten und im Rahmen von Softwaresystemfamilien darum, die Architektur, das Programmiermodell und im ersten Schritt vor allem den Umgang mit den Plattform-Mappings so weit wie möglich zu automatisieren. Dazu ist zunächst eine gewisse Formalisierung der Architektur notwendig.

Schritt 6: Architekturmetamodell. Um Automatisierungspotenzial zu schaffen, müssen die Architekturkonzepte formal beschrieben werden. Formal heißt in diesem Zusammenhang, dass sie von Werkzeugen (Modellcheckern und Codegeneratoren) verarbeitbar sind. Daher muss ein formales Architekturmetamodell definiert werden (siehe Kasten „Metamodelle"). Dies tut man üblicherweise mittels UML. Wichtig ist, dass dieses Modell dann auch wirklich von den entsprechenden Tools verarbeitet werden kann und nicht als „UML-Bild" im Regal (oder Wiki) verstaubt. Die Frage, wie man zu einem solchen Metamodell kommt, ist im Allgemeinen nicht so einfach zu beantworten. Man braucht dazu insbesondere Erfahrung (der eine oder andere Tipp findet sich auch in [7]). In unserem aktuellen Zusammenhang ist das Problem aber weit weniger dramatisch, da wir ja die Architekturkonzepte bereits definiert haben. Die Konzepte, die man im Metamodell finden sollte, sind also bereits mehr oder weniger klar. Diese Konzepte müssen nur weiter präzisiert und als Metamodell notiert werden. Dieser Zwang zur weiteren

Anzeige

Präzisierung ist in diesem Zusammenhang gewollt – die Architekturkonzepte werden dadurch präziser definiert.

Schritt 7: Glue-Code-Generierung. Wir haben nun ein formales Architekturmetamodell. Auf dessen Basis können wir nun das zu bauende System architekturell beschreiben. Man definiert also Modelle, die die Struktur des Systems *formal* beschreiben. Diese Modelle sind natürlich Instanzen des Metamodells, da sie die Begriffe aus dem Metamodell verwenden. Außerdem haben wir eine wohl definierte Technologieabbildung. Die Anwendung dieser Regeln ist also idealerweise eine stupide, sich wiederholende Angelegenheit. Solche Dinge automatisiert man am besten, dann schleichen sich keine Flüchtigkeitsfehler ein und man spart sich (langweilige) Arbeit.

Es ist nun also an der Zeit, auch diese Abbildungsregeln in Form von Codegenerierungs-Templates zu formalisieren. Als Folge dessen können wir dann den Glue Code automatisch generieren. Das ist der Code, der nötig ist, um die Artefakte, die – wie von Programmiermodell definiert – von Hand programmiert wurden, auf der Plattform zum Laufen zu bringen. Durch diese Automatisierung sinkt der Arbeitsaufwand für die Entwicklung des Systems erheblich. Vor allem lassen sich Änderungen an den Abbildungsregeln sehr leicht systemweit umsetzen – es sind ja nur Änderungen an den Templates nötig. Dadurch werden Änderungen an der Architektur möglich, die sonst einen viel zu großen Refactoring-Aufwand bedeutet hätten. Übrigens: Letztendlich ist dieser Schritt nur die konsequente Fortsetzung der Build-Automatisierung, denn dort werden schon immer aus bestimmten Artefakten andere Artefakte erzeugt. An dieser Stelle ist man nun bei einem modellgetriebenen Ansatz [7] gelandet, der sich ganz natürlich auch den Anforderungen von Softwarearchitektur ergeben hat.

Schritt 8: DSL-basiertes Programmiermodell. Für die Generierung von Glue Code muss entsprechender Input für den Generator vorhanden sein. Im einfachsten Fall kann man die nötigen Informationen direkt aus den Artefakten des Programmiermodells extrahieren. Üblicherweise muss man diese dazu mit Annotationen (seit Java 5 bzw. .NET ja aus der Nischenecke aufgetaucht) ausstatten, da nicht alle nötigen Informationen direkt im Code unterzubringen sind. Allerdings kommt man damit auch nicht wirklich weit, weil man sich immer noch fundamental auf der Abstraktionsebene des Quellcodes bewegt. Dies ist oft ein Problem, vor allem auch dann, wenn man Nicht-Programmierer wie z.B. Domänenexperten enger in den Entwicklungsprozess einbinden will. Auch für bestimmte architekturelle Aspekte eignen sich nicht programmiersprachliche Notationen oft besser. Man denke z.B. an Box-and-Line-Diagramme für Komponenten und ihre Beziehungen.

Eine Verbesserung an dieser Stelle bringen nun domänenspezifische Sprachen (DSLs). Diese können sich, was die Notation angeht, für bestimmte Aspekte erheblich besser eignen als Programmcode. Natürlich muss man entsprechende Editoren haben, um die Modelle zu erstellen. Vor allem aus diesem Grund beginnt man oft mit UML-basierten Modellen. Auch wenn sich andere Notationen möglicherweise besser eignen würden, gibt es für UML eine breite Palette ausgereifter Werkzeuge. Das Metamodell der DSL ist übrigens zunächst das bereits oben beschriebene Architekturmetamodell, da wir ja (zunächst) architekturrelevante Dinge beschreiben wollen. In späteren Phasen wird man auf diese architekturzentrierte Modellierung tatsächlich „fachlichere" Modelle kaskadieren; dafür werden dann entsprechende zusätzliche Metamodelle sowie meist Modell-zu-Modell-Transformationen benötigt [6]. Man will ja die fachlichen Konstrukte auf die bereits definierten architekturellen Konstrukte abbilden.

Nachdem wir uns im Laufe des Artikels nun vor allem mit der Architektur beschäftigt haben, sind wir nun an der Stelle angelangt, wo man auch ganz greifbare Vorteile für das Anwendungsdesign bekommt: Man wird feststellen, dass ein DSL-basiertes Programmiermodell es viel besser erlaubt, das Anwendungsdesign explizit darzustellen. Eine Diskussion und eine Analyse des Designs werden damit erheblich erleichtert und nebenbei wird es damit auch automatisch dokumentiert.

Schritt 9: modellbasierte Architekturvalidierung. Vor allem in größeren Projekten ist es essenziell, dass die Konventionen und Regeln auch eingehalten werden. Die per DSL spezifizierten architekturellen Aspekte müssen also auf Korrektheit geprüft werden. Manuelle Reviews, vor allem auf Codeebene, sind zeitaufwendig und skalieren schlecht. Konsequenterweise sollte also vor irgendwelchen Generierungsschritten zunächst das Modell auf Korrektheit bezüglich des Metamodells und den darin enthaltenen Constraints geprüft werden. Auf dieser Ebene lassen sich nicht erlaubte Abhängigkeiten oder Schichtenzugriffsverletzungen sehr leicht finden. Im Code ist das schon erheblich schwieriger. Wichtig ist dabei auch, dass man durch entsprechende Konventionen, Checks und „kreative" Verwendung des Compilers zu erreichen versucht, dass man bei eventuell manuell zu schreibendem Code nichts mehr falsch machen kann. Ein Beispiel: Angenommen, man definiert im Modell eine Reihe von Abhängigkeiten zwischen zwei Komponenten und diese werden vom Modellcheck als „in Ordnung" angesehen. Dann darf es nicht möglich sein, dass der Entwickler beim Schreiben des manuellen Codes weitere Abhängigkeiten schafft, aus Versehen oder beabsichtigt, da sonst natürlich die Aussage des Modellvalidators („alles OK") nichts bringt.

Reihenfolgen

Obige Auflistung und auch der Begriff „Schritt" klingen sehr nach sequenzieller Abarbeitung. Das ist aber nur teilweise so gemeint. Zunächst zu den drei Phasen: Der Artikel geht davon aus, dass man nicht von vornherein weiß, dass man modellgetrieben vorgehen wird. Phase eins und zwei sind auch unabhängig von modellgetriebener Softwareentwicklung sinnvoll. Wenn sich allerdings im Laufe des Projekts herausstellt, dass es Potenzial für Automatisierung gibt – beispielsweise weil man die Architektur in mehr als einer Anwendung verwenden will –, beschreibt Phase drei, wie man dies erreicht. Falls man in einem Projekt allerdings von vornherein weiß, dass man modellgetrieben vorgehen will, so kann man die beiden Phasen verschmelzen, was zu folgenden Schritten führen wird:

- technologieunabhängige Architektur beschrieben mittels eines Architekturmetamodells
- Programmiermodell inkl. DSLs und modellbasierte Architekturvalidierung
- Technologieabbildung inkl. Glue-Code-Generierung
- Mock-Plattform inkl. Glue-Code-Generierung
- vertikaler Prototyp

Wenn wenig Erfahrung mit MDSD vorliegt, ist es übrigens ratsam, auch die getrennten Phasen zu verwenden, selbst *wenn* man weiß, das man im Endeffekt auf MDSD hinaus will, und zwar aus folgendem Grund: In Phase drei kann man sich ausschließlich mit den Herausforderungen befassen, die sich aus dem modellgetriebenen Ansatz ergeben; die Architektur ist inhaltlich bereits in Phase eins und zwei ausgearbeitet worden. Wenn man es richtig macht, kann man die nötigen Codegenerierungs-Templates quasi direkt aus dem vertikalen Prototyp ableiten.

Eine andere Sache betrifft die Schritte in Phase eins. Auf den ersten Blick läuft ja im Endeffekt alles darauf hinaus, die Architektur anhand des vertikalen Prototyps zu verifizieren – die Erstellung vertikaler Prototypen ist ja sowieso gute Praxis. Warum also nicht einfach mit dem vertikalen Prototyp anfangen? Bei genauerer Betrachtung stellt sich heraus: Erfahrungsgemäß kann man mit einem vertikalen Prototypen zwar schnell die Technologieabbildung und die groben Strukturen der Anwendungen validieren, die anderen wichtigen Artefakte (wie technologieunabhängige Architektur und Programmiermodell, wohl definierte Technologieabbildungen oder Mock-Plattform) fallen aber nicht „einfach so" nebenbei heraus. Es ist daher wichtig, der Definition von Konzepten einen hohen Stellenwert einzuräumen. Deswegen ist der vertikale Prototyp das Ergebnis von Phase eins und nicht der Startpunkt! Es sei noch mal erwähnt, dass man die Schritte in Phase eins auf keinen Fall wasserfallartig durchführen sollte (also erst das komplette Architekturkonzept, dann das komplette Technologiemapping etc.), sondern auch hier ist iteratives und inkrementelles Vorgehen notwendig. In der Praxis passieren daher die Schritte in Phase eins quasi parallel.

Zusammenfassung

Der in diesem Artikel gezeigte Ansatz zur Softwarearchitektur hat sich in diversen Projekten bewährt. Er ist natürlich nicht wirklich neu. Gute Architektur hat schon immer Konzepte und Technologie getrennt. Wir denken aber, dass modellgetriebene Softwareentwicklung und das heute vorhandene Tooling einen solchen Ansatz erheblich praktikabler machen, als dies noch vor einigen Jahren der Fall war. Es gibt eigentlich gar keinen Grund mehr, sich so stark an Technologien zu binden. Obiger Ansatz ist nicht nur sinnvoller, sondern auch schneller und macht mehr Spaß. Es gibt also keine Ausreden mehr!

Markus Völter arbeitet als freiberuflicher Berater für Softwaretechnologie und -Engineering. Seine Schwerpunkte liegen dabei auf Softwarearchitektur, modellgetriebener Entwicklung und Middleware.
Arno Haase arbeitet ebenfalls als freiberuflicher Berater. Seine Tätigkeitsschwerpunkte liegen dabei auf Softwarearchitektur, insbesondere modellgetriebener Entwicklung – sowie agilen Methoden und Projektreviews.

Links & Literatur

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

[2] Markus Völter: Software Architecture Patterns – Examples Only: www.voelter.de/data/articles/ArchitecturePatterns-ExampleOnly.pdf

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: Pattern-Oriented Software Architecture, Vol. 1: A System of Patterns, Wiley, 1996

[4] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann: Pattern-Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects, Wiley, 2000

[5] J2EE Blueprints: bpcatalog.dev.java.net/nonav/solutions.html

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

[7] Thomas Stahl, Markus Völter: Modellgetriebene Softwareentwicklung, dpunkt, 2005

[8] Eric Evans: Domain Driven Design, Addison-Wesley,

Anzeige