von jorn bettin

mehr zum thema:
http://www.softmetaware.com/whitepapers.html

PROZESSAUSWIRKUNGEN VON MDSD

Im Vergleich zu herkömmlicher Softwareentwicklung stellt sich das Model-Driven Software Development (MDSD) auf den ersten Blick als grundlegend neues Paradigma dar. Bei näherem Hinsehen handelt es sich allerdings eher um eine neue Bewertung von Prioritäten. Ganz wie beim Übergang von Assemblern zu Kompilern liegt das Hauptakzeptanzproblem oft nicht auf der technologischen, sondern vielmehr auf der kulturellen Seite. Es ist also nicht verwunderlich, dass sich modellgetriebene Verfahren nicht über Nacht auf breiter Front durchsetzen. Veränderungen an Softwareentwicklungsprozessen müssen sorgfältig eingeführt werden, da das weitere Umfeld aller an der Softwareentwicklung Beteiligten betroffen ist und es nicht einfach um das Erlernen von neuen APIs oder einer neuen Sprachsyntax geht. Dieser Artikel beschreibt, wie sich modellgetriebene Softwareentwicklung von traditioneller iterativer Softwareentwicklung unterscheidet.

der autor

Jorn Bettin (E-Mail: jorn.bettin@softmetaware.com) arbeitet seit 1993 mit modellgetriebenen Verfahren und ist Gründer von SoftMetaWare, einer international arbeitenden Unternehmensberatung mit Spezialisierung auf strategische Unterstützung bei Softwareentwicklung im Großen.

Frameworks sind in MDSD von zentraler Bedeutung, weshalb es sich lohnt, erst einmal die Benutzung und Entwicklung von Frameworks zu untersuchen. Hier gilt die Lektion *Small is Beautiful* auch für modellgetriebene Ansätze.

In MDSD besteht der Grundansatz in der manuellen Entwicklung einer konkreten Referenzimplementierung, welche alle architektonischen Schichten umfasst und sehr schmal gehalten werden kann. Eine solche Referenzimplementierung sollte bereits die Anbindung an geeignete kommerzielle oder Opensource-Frameworks enthalten. Dann kann der notwendige Frameworkanbindungscode abstrahiert und in Form von Codetemplates festgehalten werden. Dieser Schritt erhöht die Referenzimplementierungskosten gegenüber normalem *Proof Of Concept*-Prototyping nur um etwa 15%-25%. Wenn man sich zum Beispiel überlegt, wie viel Quellcode in typischen J2EE-Anwendungen aus repetitivem Frameworkanbindungscode besteht, dann sieht man schnell, warum der Einsatz von MDSD viel lästige Arbeit ersparen kann.

Entwicklung domänenspezifischer Frameworks

Der Begriff *Domäne* wird in erster Linie in Zusammenhang mit industriespezifischen Fachbereichen wie Banking, Produktionsplanung usw. verwendet. In MDSD sind jedoch auch technische Domänen, d.h. Unterdomänen der Softwareentwicklungsdomäne, von Bedeutung. Gerade wenn ein Softwareteam ohne tiefe Kenntnisse einer fachlichen Domäne startet, sollten zumindest auf der technischen Ebene alle Register der Automatisierung gezogen werden.

In MDSD muss jede Investition in Frameworks durch konkrete Anwendungsanforderungen motiviert sein. Diese Vorgehensweise schließt von vornherein aus, dass ein Frameworkteam im luftleeren Raum über Anforderungen spekuliert und sich dann an die Entwicklung der berüchtigten Eier legenden Wollmilchsau macht.

In größeren MDSD-Projekten bietet sich allerdings trotzdem die Trennung von Anwendungs- und Frameworkteams an, nicht etwa um Frameworkentwickler von der Realität abzuschirmen, sondern zur Vereinfachung von Projektmanagement und Releasemanagement. Ein bewährtes Rezept ist die Einrichtung einer Architekturgruppe, welche sich mit der Erstellung von Vorgaben für die Anwendungsplattform befasst und sich aus den Teamleitern/Architekten der Anwendungsteams zusammensetzt. Im Frameworkteam befinden sich technische Spezialisten, welche Referenzimplementierungen entwickeln, die Anbindung an externe Frameworks vornehmen, Codetemplates ableiten und bei Bedarf kleine Frameworks entwickeln. Die Ergebnisse werden von der Architekturgruppe evaluiert, welche in Bezug auf das Frameworkteam die Rolle des Kunden spielt und somit auch über das Budget und die Prioritäten entscheidet.

Wieso Generierung, wenn doch Frameworks die meiste Arbeit tun?

Selbst bei kleinen Frameworks ist oft der Quellcode und weitere Dokumentation nicht geeignet, um Frameworkbenutzern in kurzer Zeit das notwendige Wissen über die korrekte Benutzung des Frameworks zu vermitteln. *Zeit ist Geld* gilt insbesondere in Softwareprojekten.

Hier zeigen sich MDSD und modellgetriebene Generierung von Frameworkanbindungscode von ihrer stärksten Seite. Frameworkautoren können durch Erstellung einer kleinen Beispielanwendung und durch die Extraktion von Codetemplates die korrekte Benutzung von Frameworks weitest gehend automatisieren und gleichzeitig Frameworkbenutzer von lästigen Details fernhalten. Es gibt natürlich auch diejenigen, die behaupten, dass dies nur zur Verdummung der Anwendungsentwickler führe und eher gefährlich als nützlich sei. Diese Argumentation gleicht jedoch verdächtig der Verteidigung von Assembler- gegenüber Hochsprachen. MDSD-Werkzeuge ermöglichen es Frameworkentwicklern „ganze Arbeit" zu leisten. Hier ist es an der Zeit, die Messlatte an die Qualität von Frameworks höher zu schrauben. Frameworkbenutzer wollen nicht die Komplexität des Frameworks bestaunen, sondern benötigen solide Bausteine, welche die Anwendungsentwicklung erleichtern.

MDSD bietet die Möglichkeit Frameworkspezialwissen auf wenige Personen zu

konzentrieren. Dies bedeutet, dass tiefes technisches Wissen allen Anwendungsteams in optimaler Form zur Verfügung gestellt werden kann.

Domänenengineering

In der Product Line Engineering-Terminologie (vgl. [SEI]) wird die Entwicklung einer Anwendungsplattform und dazugehörender Werkzeuge und Prozessbausteine auch als *Domänenengineering* bezeichnet (siehe Abb. 1).

Die Praktiker wissen es schon lange, es gibt keinen allgemeingültigen Anwendungsentwicklungsprozess. Breitspurig angelegte Prozess-Frameworks können zwar als Anfangspunkt benutzt werden, haben jedoch eine Reihe von nicht zu unterschätzenden Schwächen:

- Der Erfahrungsstand einzelner Teammitglieder bleibt unberücksichtigt. Dieses Problem wird sehr schön von Alistair Cockburn (vgl. [Coc01]) beschrieben.
- Oft fehlt Teams das notwendige Expertenwissen, um effizient die relevanten Prozessteile zu identifizieren und in verdaubarer Form als Projektstandard zu etablieren.
- Nur Methoden, die explizit auf Software Product Line Engineering zugeschnitten sind, unterscheiden explizit zwischen Anwendungsentwicklung und der Entwicklung einer Anwendungsproduktionsstraße (Domänenengineering).
- Die durch Benutzung von domänenspezifischen Frameworks und Automatisierung erzielbaren Prozessvereinfachungen bleiben unberücksichtigt.
- Entscheidungsverfahren bezüglich der Verwendung von Off-the-Shelf- oder Opensource-Komponenten als Alternative zu Eigenbau werden in vielen Prozessen nur am Rande erwähnt. Dieses schwierige Thema wird anscheinend gerne übergangen.

MDSD versucht hier durch die Bereitstellung von Best Practices (vgl. [Bet04a], [Bet04b]) Abhilfe zu schaffen. Ziel ist es, die Kostenschwelle und die Risiken bei der Einführung von Product Line Engineering-Verfahren erheblich zu senken.

Sprachendesign

Der Erfolg des MDSD-Ansatzes lässt sich an der Kompaktheit der daraus resultierenden Anwendungsmodelle messen: Wenn die Modelle mühsam zu pflegen sind, wird sich kaum ein Entwicklungsteam auf ein solches Verfahren einlassen.

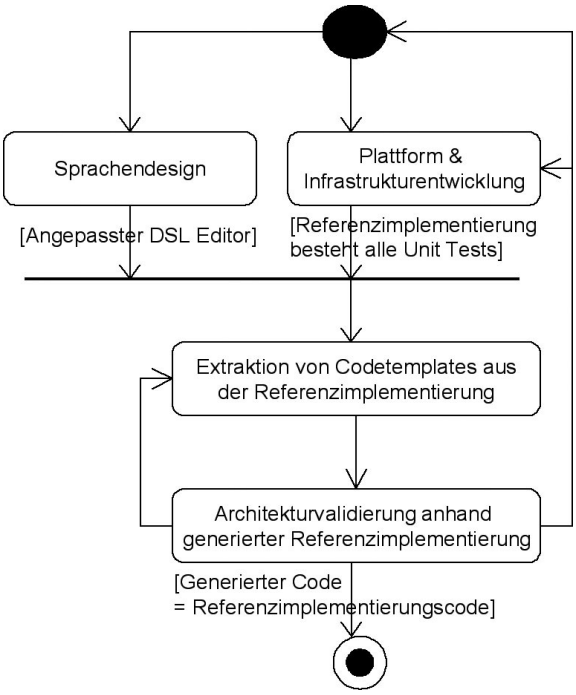

Abb. 1: Domänenengineering-Workflow

Sobald eine *Domain Specific Language* (DSL) in Form eines Metamodells formal definiert ist, kann entweder mit Hilfe von MDSD-Generatorwerkzeugen ein spezieller DSL-Editor erzeugt werden, oder es werden einfach normale Texteditoren bzw. UML-Werkzeuge zur Anwendungsmodellierung benutzt. Wie komfortabel ein DSL-Editor sein muss, hängt ganz von der Projektsituation ab. Wenn größere Teams mit dem DSL-Editor tagein tagaus über längere Zeiträume arbeiten müssen, wie z. B. in der Produktentwicklung, dann lohnt es sich, einen entsprechend komfortablen Editor als *Anwendungsmodellierungswerkzeug* zu entwickeln. Die heute verfügbaren MDSD-Toolkits lassen in Bezug auf die Generierung von hochqualitativen Modellierungswerkzeugen noch einiges zu Wünschen übrig. Allerdings bietet hier MDSD das ideale Mittel zur Selbsthilfe: Durch Verwendung von entsprechenden Codetemplates lässt sich schnell ein einfacher DSL-Editor generieren. Der DSL-Editor kann anschließend manuell verfeinert beziehungsweise durch Einbindung von graphischen Modellierungskomponenten an entsprechenden Stellen inkrementell zu einem komfortablen Werkzeug ausgebaut werden.

Plattform- und Infrastrukturentwicklung

Wie bereits erwähnt, sollte die Entwicklung von DSLs als die konsequente Vervollständigung und Abrundung der Entwicklung von domänenspezifischen Frameworks betrachtet werden. Wenn entsprechende Frameworks aus der Referenzimplementierung herausgeschält worden sind, muss die Referenzimplementierung natürlich angepasst und getestet werden.

Der abschließende Schritt vor dem Echteinsatz der neuerstellten, maßgeschneiderten Anwendungsentwicklungsumgebung besteht typischerweise aus der Definition einer nicht-trivialen Testanwendung, die interessante Grenzfälle beinhaltet, welche aus ökonomischen Gründen nicht von der ursprünglichen Referenzimplementierung abgedeckt werden konnten. Die Testanwendungsdefinition wird benutzt, um Codetemplates und die gesamte Anwendungsarchitektur realistisch „in der Breite" zu testen. Dies ist gegenüber herkömmlicher Softwareentwicklung ein entscheidender Vorteil, und ist ganz im Sinne iterativer Verfahren, wo die frühestmögliche Risikominimierung im Vordergrund steht.

Modellgetriebene Anwendungen

Es stellt sich die interessante Frage, in wie weit fachlich motivierte DSLs in MDSD eine Rolle spielen. Tiefes Fachdomänenwissen kann dazu benutzt werden, um Endbenutzern *modellgetriebene Anwendungen* zur Verfügung zu stellen, d.h. Anwendungen, die selbst zum Teil aus einer DSL bestehen, welche zur Laufzeit ▸

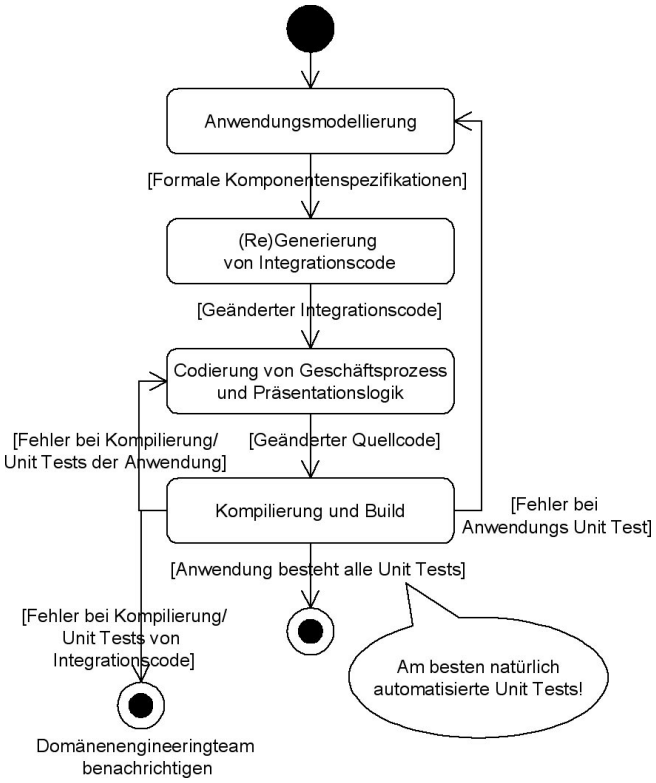

Abb. 2: Anwendungsentwicklungs-Workflow

direkt vom Endbenutzer eingesetzt wird. Konkrete Beispiele: Spreadsheets, Elektrizitätsnetzwerk-Modellierungswerkzeuge, Workflowdefinitionskomponenten, usw. In diesen Beispielen werden Endbenutzer in die Lage versetzt, beliebig komplexe Modelle in einer vertrauten bzw. leicht erlernbaren Notation zu definieren. Der Wert solcher Systeme liegt in der maschinellen Verarbeitung der Modelle. Erfahrungsgemäß haben fachlich motivierte DSLs eine sehr hohe Stabilität und Lebensdauer.

Technisch motivierte DSLs werden in erster Linie im Rahmen des *Anwendungsentwicklungsprozesses* in Kombination mit generativen Techniken eingesetzt. Fachlich motivierte DSLs werden tendenziell eher direkt in die *Anwendungsgeschäftsprozesse* von Endbenutzern eingebettet und kommen dann zur Laufzeit mit interpretativen Verfahren (oder durch Generierung *on-the-fly*) zum Einsatz.

Hier fängt die Grenze zwischen Anwendungsentwicklungs- und Anwendungsgeschäftsprozess an zu zerfließen. Für die meisten Softwareentwickler und Anwender ist dies sicher ein ungewöhnlicher Gesichtspunkt. In welcher Form MDSD zum Einsatz kommt, ergibt sich mehr oder weniger „von selbst" aus sorgfältiger Domänenanalyse (vgl. [Cle01]).

Anwendungsentwicklung

Bis jetzt haben wir uns auf den Domänenengineering-Prozess konzentriert. Mindestens ebenso interessant sind die Auswirkungen von MDSD auf den Anwendungsentwicklungsprozess (siehe Abb. 2), denn schließlich liefert dieser die echten Anwendungen.

Sehr wichtig für erfolgreiche MDSD ist das reibungslose Zusammenspiel zwischen Domänenengineering und Anwendungsentwicklung. Hier bringen agile Vorgehensweisen und der Einsatz von *Timeboxed Iterations* die besten Resultate. Die durch MDSD erzwingbare architektonische Konsistenz ist der Schlüssel zur Skalierbarkeit von agilen Methoden.

Die notwendigen Anwendungsmodellierungsschritte hängen von den Eigenschaften der definierten DSL(s) ab, und liegen somit ganz in der Hand der Domäneningenieure. Wenn Anwendungsentwickler Schwächen in der Modellierungssprache entdecken und Verbesserungsmöglichkeiten aufzeigen, dann können diese iterativ und inkrementell realisiert und ausgeliefert werden – gewissermaßen als Upgrade der Produktionsstraße. Genau dieser Freiraum zur Anpassung des Entwicklungsprozesses an den spezifischen Kontext ermöglicht Effizienzsteigerungen gegenüber „steiferen" Entwicklungsprozessen, welche versuchen alle Softwareentwicklungsprojekte auf einen einzigen Hauptnenner zu bringen.

Codierung von Geschäftsprozesslogik

MDSD setzt nicht voraus, dass komplette Anwendungen auf Modellebene definiert werden. Vielmehr können in MDSD ganz pragmatisch gewisse Aspekte manuell auf traditionelle Weise implementiert werden. Dies bietet sich zum Beispiel zur Implementierung von Geschäftslogik an, welche sich elegant in herkömmlichen Sprachen in Form von logischen Ausdrücken realisieren lässt. MDSD stellt zu diesem Thema eine Reihe von Best Practices zur Integration von handgeschriebenem und generiertem Code zur Verfügung (vgl. [Bet04a]).

Kompilierung und Build

In diesem Punkt unterscheidet sich MDSD nicht von anderen Verfahren, es gelten die gleichen Regeln: Der Build-Prozess sollte so weit wie möglich automatisiert werden. Im Bezug auf MDSD heißt dies konkret, dass neben den üblichen automatisierten Unit Tests auch entsprechende Aufrufe von Generatoren eingebaut werden sollten.

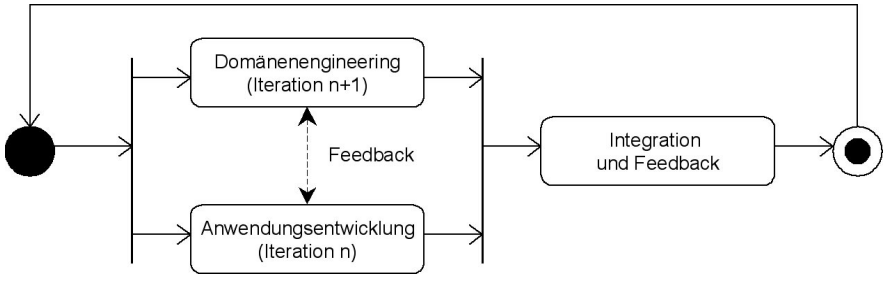

Abb. 3: Die Anwendungsentwicklung hinkt gewissermaßen immer eine Iteration hinter den Iterationen des Domänenengineering her. Hier wird deutlich, warum Timeboxing essentiell ist, und es wird auch deutlich, dass MDSD keineswegs Big Design Up-Front bedeutet.

Fazit

Ich habe in diesem Artikel einige wesentliche Elemente von modellgetriebenen Softwareentwicklungsprozessen skizziert. Es wird deutlich, dass MDSD einen Gegenpol zu universell gültigen Anwendungsentwicklungsprozessen darstellt, und somit überall da von Bedeutung ist, wo tiefe Domänenkenntnisse fachlicher oder technischer Natur vorhanden sind und zur Prozessoptimierung ausgenutzt werden können. Allerdings soll nicht verschwiegen werden, dass die Einführung von MDSD durchaus eine Investition darstellt – in erster Linie eine Investition in die Ausbildung von Mitarbeitern – und nur mit entsprechender Vorbereitung zum gewünschten Erfolg führt. Technisch orientierte Architekten und Softwareentwickler mögen die größten Schwierigkeiten in der Verfügbarkeit entsprechender Werkzeuge vermuten, dies ist jedoch ein Trugschluss:

- Es gibt mittlerweile eine ganze Reihe von praxistauglichen kommerziellen und Opensource-MDSD- und MDA-Toolkits.
- MDSD bedeutet per Definition die Entwicklung eines maßgeschneiderten Anwendungsentwicklungsprozesses und maßgeschneiderter Werkzeuge, sodass die Qualität der verwendeten Generator-Toolkits zwar eine Rolle spielt, aber durchaus nicht der einzige zu berücksichtigende Faktor ist. Ein großer Teil des Erfolgsrisikos hängt von der sorgfältigen und gewissenhaften Integration der einzelnen Teilwerkzeuge ab.
- Softwareentwicklung besteht nur zum Teil aus Programmierung, und kein Werkzeug kann harte Analysearbeit ersetzen. Das eigentliche Potential von MDSD liegt in der konzeptionellen Unterstützung bei der Domänenanalyse und schlägt sich in der Qualität von Frameworks und DSLs nieder. Wenn dort „gespart" wird, dann hilft der beste Generator nichts.
- Und letztendlich liegen größere Hürden erfahrungsgemäß im kulturellen Bereich. Schließlich gilt es zu bedenken, dass die Einführung von MDSD eine erhebliche Änderung des gewohnten Softwareentwicklungsprozesses darstellt. Das erforderliche Umdenken braucht Zeit. Der MDSD-Einführungsprozess (vgl. [Sta04] und [Bet04b]), auf den ich hier aus Platzgründen nicht näher eingegangen bin, und auch die ersten Iterationen des Domänenengineering sollten am besten von Experten mit MDSD-Erfahrung begleitet werden. ■

Literatur & Links

[Bet04a] J. Bettin, Model-Driven Software Development: An emerging paradigm for industrialized software asset development, siehe: www.softmetaware.com/mdsd-and-isad.pdf

[Bet04b] J. Bettin, Model-Driven Software Development Teams, Building a Software Supply Chain for Distributed Global Teams, siehe: www.softmetaware.com/distributed-software-product-development.pdf

[Bet04c] J. Bettin, Model-Driven Software Development Activities, The Process View of an MDSD Project, siehe: www.softmetaware.com/mdsd-process.pdf

[Cle01] C. Cleaveland, Program Generators with XML and Java, Prentice Hall, 2001

[Coc01] A. Cockburn, Agile Software Development, Addison-Wesley, 2001

[SEI] Carnegie Mellon Software Engineering Institute, Product Line Practice, siehe: www.sei.cmu.edu/plp/

[Sta04] T. Stahl, M. Völter, J. Bettin, Modellgetriebene Softwareentwicklung, dPunkt, 2004 (noch nicht veröffentlicht)