MDA: Weg oder Irrweg?

Sören Teurich-Wagner

sd&m Research
sd&m AG
Carl-Wery-Straße 42
81739 München
soeren.teurich@sdm.de

Abstract: MDA wird als Zukunft des Software Engineering dargestellt. Trotz positiver Erfahrungsberichte vertreten wir die Auffassung, dass noch ein weiter Weg vor uns liegt, bis MDA in großem Maßstab eingesetzt wird. Dies ist bedingt durch die heute noch unbefriedigenden Möglichkeiten zur Dynamikmodellierung in UML. Wir präsentieren ein Architekturprinzip, das die von MDA postulierte Trennung von Anwendung und Technik ebenfalls umsetzt, aber mit den heute verfügbaren Sprachen und Technologien ein- und umsetzbar ist. Außerdem stellen wir ein Paradigma für den Entwurf von Komponenten vor, das ebenfalls massiv auf den Einsatz von Modellen setzt, aber einen viel kleineren Scope als MDA hat. Diese Techniken sind bei sd&m erfolgreich im Einsatz.

1 Was ist MDA?

Die Model Driven Architecture ([MDA], [Fr03]) ist eine Initiative der Object Management Group mit dem Ziel, die verschiedenen Middleware- und Implementierungstechniken, denen sich der Softwareentwickler heute stellen muss, zu integrieren und den Übergang auf neu entstehende Standards (derzeit besonders im Bereich Web Services) zu erleichtern. Dabei ist das Grundprinzip die Trennung von Anwendung und Technik. Die Anwendung wird auf einem höheren Abstraktionsniveau als dem konventioneller Programmiersprachen modelliert. Der Weg dort hin ist eine gestufte Erstellung eines Softwaresystems:

- Der erste Schritt ist die technikunabhängige Modellierung (im Platform Independent Model, PIM) der Fachlichkeit des Systems. Dieses Modell bleibt während der ganzen Entwicklung führend; es wird in UML formuliert.

- Daraus wird ein technikabhängiges Modell (Platform Specific Model, PSM) generiert. Bei diesem Übergang wird die Zielumgebung (Sprache, Middleware etc). festgelegt. Dieser Generatorschritt benötigt – je nach Werkzeug – zusätzliche Informationen zu den Modellelementen, z.B. Eigenschaften von Session-Beans.

§ Der abschließende Generatorschritt transformiert das PSM in die Zielsprache.

2 Mängel von MDA

Wir sehen vier Mängel, die den Nutzen von MDA beeinträchtigen:

1. Für die Entwicklung von Anwendungen durch Modellierung sind Statik und Dynamik gleich wichtig. In der UML liegt der Fokus aber auf der Statik; die Sprachmittel zur Dynamikmodellierung sind nicht so weit entwickelt. Die Generierung von Code aus Statikmodellen ist keine besondere Neuerung und stellt keine Verbesserung gegenüber existierenden CASE-Werkzeugen dar.

2. Es gibt zwar eine Reihe von Ansätzen für UML-Dynamiksprachen, z.B. xUML [KC02] oder Executable UML [Me99], aber diese sind weder ausgereift noch standardisiert. Anforderungen an eine solche Sprache sind unter anderem eine erhebliche Menge und Mächtigkeit von unterstützten Datentypen und Funktionen darauf. Ein Beispiel dafür ist eine breite Funktionalität für Datumsberechnungen, Kalenderdarstellungen, Unterstützung für international verwendbare Kalender inkl. Feiertagen etc., um Modelle auf einem abstrakten Niveau zu erstellen. Außerdem bedarf es einer einfachen Abbildbarkeit auf gängige Sprachen wie C# und Java.

   Das Resultat der Bemühungen um eine solche Sprache kann nach unserer Einschätzung nur eine Sprache sein, die Java oder C# stark ähnelt. Bei der Entwicklung von Programmiersprachen kann man feststellen, dass Verbesserungen gegenüber den existierenden Sprachen selten sind und viele der neuen Sprachen keine relevante Verbreitung finden. Es braucht viele Iterationen und ein erhebliches Investment des Entwicklers, bis sie für die Erstellung großer, unternehmenskritischer Systeme taugen. Die Entwicklung von Java ist für dieses Muster ein prominentes Beispiel. Andere Sprachen sind bereits bei Markteintritt sehr weit entwickelt; um dies zu erreichen, muss der Erfinder einer solchen Sprache ebenfalls viel Zeit und Geld investieren und sich zusätzlich an den erfolgreichen Konzepten anderer, bereits ausgereifter Sprachen orientieren. Genau dieses Muster hat Microsoft mit C# und dem .NET-Framework erfolgreich verfolgt. Um erfolgreich zu sein, müsste sich die Entwicklung einer Dynamiksprache für die UML genau an diesem Muster orientieren. Das Abstraktionsniveau dieser Sprache entspräche aber dann dem von Java oder C#.

3. In dem entstehenden Code finden sich bei den derzeit vorhandenen Werkzeuge mehr oder wenige große Lücken, die manuell ausprogrammiert werden (unter Berücksichtigung der Architektur auf Codeebene des darum herum existierenden generierten Codes). Diese Lücken entstehen durch das Defizit der UML hinsichtlich der Dynamikmodellierung. Ändert sich die Fachlichkeit des Systems, wird diese Änderung im PIM nachvollzogen und der ganze Generierungsprozess beginnt von neuem. Dabei ist die Erhaltung des handgeschriebenen Codes im Generat besonders kritisch.

4. Auch das Framework und die Patterns, aus denen das PSM und der Code generiert werden, sind ein wesentliches Erfolgskriterium für MDA. Der Aufwand für Erstellung und Pflege dieser Patterns entspricht mindestens dem für Erstellung und Pflege des technischen Rahmens eines konventionell erstellten Systems, damit ist mit MDA hier kein Vorteil erzielbar.

Insgesamt bewerten wir die Tauglichkeit von MDA zurückhaltend. Mit dem derzeitigen Stand der Entwicklung von MDA bleibt zum einen das Problem des unvollständigen Codes, den eine MDA-basierte Entwicklung liefert und die manuell ausgefüllt werden. Eine vollständige Modellierung muss nicht nur die Statik, sondern auch die Dynamik eines Systems abdecken; derzeit konzentriert sich MDA aber auf die Statik, für die Dynamik gibt es keine befriedigende Lösung. Eine solche Lösung würde auf eine Sprache auf dem Niveau von Java oder C# herauslaufen und damit keinen echten Fortschritt darstellen. Der von der MDA angestrebte Rundumschlag, ein ganzes System in einem Modell erfassen zu wollen, scheitert an den daraus erwachsenden hohen Ansprüchen an die Modellierungssprache.

Es gibt jedoch andere Ansätze, die das Ziel von MDA erreichen und dabei mit existierender Technik auskommen. Dabei wird die Trennung von Anwendung und Technik durch den Komponentenschnitt erzielt. Die höhere Abstraktionsebene wird durch das lokal begrenzte Herausziehen der Modellierung aus einer Komponente erreicht.

3 Quasar – Eine Alternative zu MDA

Die in [Si00] und [Si03] vorgestellte Quality Software Architecture (Quasar) ist eine Methode zur Trennung von Anwendung und Technik, die mit bestehenden Sprachen und Technologien wie Java/J2EE und C#/.NET einfach umzusetzen ist.

Das Grundprinzip von Quasar ist der Komponentenschnitt nach dem Blutgruppen-Kriterium. Jede Komponente hat eine eindeutig identifizierbare Blutgruppe. Folgende Blutgruppen gibt es:

- A-Komponenten sind bestimmt durch die Anwendung und unabhängig von der Technik. A-Software ist der eigentliche Daseinszweck des Systems und wird in der Regel den größten Teil ausmachen. Natürlich sollte sich jede A-Komponente nur mit einem Anwendungsthema befassen.

- T-Komponenten sind bestimmt durch eine Technik und unabhängig von der Anwendung. Eine T-Komponente kennt mindestens ein technisches API wie JDBC oder CORBA. Eine gute T-Komponente sollte sich nur mit einem API befassen. T-Software ist unabdingbar, denn ohne Datenbank, Middleware und Betriebssystem kann kein System existieren.

- 0-Komponenten sind weder von der Anwendung noch von Technik abhängig. Sie sind ideal wieder verwendbar, für sich alleine aber nutzlos. Klassenbibliotheken wie die Collections aus dem JDK sind ein Beispiel.

- AT-Komponenten befassen sich mit Anwendung und Technik gleichzeitig. Sie sind daher schwer zu warten, widersetzen sich Änderungen und können kaum wieder verwendet werden. Um die Trennung von Anwendung und Technik zu realisieren, müssen solche Komponenten vermieden werden.

- R-Komponenten sind eine milde (und damit akzeptable) Form von AT-Komponenten. Sie sind spezialisiert auf die Transformation von Anwendungsobjekten in externe Darstellungen und wieder zurück. R-Software kann häufig aus Metainformationen generiert werden.

Eine A-Komponente darf nicht direkt die Schnittstelle einer T-Komponente aufrufen, denn dann wäre sie ja wieder von dieser T-Komponente abhängig (dies gilt auch in der anderen Richtung). Die Verbindung wird geschaffen durch Adapter, die der A-Komponente gegenüber eine Schnittstelle auf A-Ebene bieten unter Nutzung der Dienste der T-Komponente. Diese Adapter sind zwar R-Software, aber für gewöhnlich relativ klein und häufig generierbar.

Das Einhalten dieser Kategorien beim Komponentenschnitt führt dazu, dass jeder Komponente ein abgegrenztes und überschaubares Aufgabenfeld zugeordnet wird. Technik und Anwendung werden von einander getrennt. Damit wird die Austauschbarkeit von Technik unterstützt: es muss nur gelingen, die neue Technik hinter der bestehenden Schnittstelle anzubieten. Dies wird durch entsprechendes Schnittstellendesign sichergestellt, Details dazu siehe [Si03].

Die Verwendung dieses Designparadigmas ist unabhängig von konkreten Programmiersprachen und Werkzeuge, denn Komponentenbildung kann auf alle möglichen Programmiersprachen abgebildet werden (auch wenn die Sprache eigentlich keine Unterstützung dafür hat wie z.B. COBOL). Bei sd&m werden die Quasar-Prinzipien in zahlreichen Projekten mit völlig unterschiedlichen Technologien und Anwendungsdomänen erfolgreich eingesetzt.

4 Modellgesteuerte Komponenten statt MDA

Im Kleinen funktioniert der Grundgedanke der MDA: Einfachere Erstellung von Systemen durch Modellierung statt Programmierung. Die Modellierung ist hier immer auf einen abgegrenzten Aspekt des Systems eingeschränkt und die jeweilige Modellierungssprache kann auf diesen Aspekt hin optimiert werden.

Modellgesteuerte Komponenten verwenden die Interpretation von Modellen als Ersatz für handgeschriebenen oder generierten Code. Das komponentenspezifische Modell wird zur Laufzeit eingesetzt oder als Input für einen lokal beschränkten Generierungsprozess zur Entwicklungszeit verwendet. Es beschreibt einen Aspekt des Systems, z.B. das OR-Mapping für eine Persistenz oder die Ablaufmodelle für eine Workflowkomponente. Durch Kombination verschiedener modellgesteuerter Komponenten (mit unabhängigen oder gekoppelten Modellen) lässt sich ein System weitgehend aus Modellen erstellen.

Modellgesteuerte Komponenten können als 0-Komponenten gestaltet werden, denn Fachlogik enthalten sie selber nicht (die kommt über das Modell hinein) und die Implementierung kann so gestaltet werden, dass Abhängigkeiten zu T-Komponenten über Stützschnittstellen aus der Komponenten isoliert und separat realisiert sind.

sd&m hat im vergangenen Jahr eine modellgesteuerte Komponente veröffentlicht [OQ], die Quasar Persistenz. Sie wird in zahlreichen Projekten bei sd&m erfolgreich eingesetzt. Weitere modellgesteuerte Komponenten befinden sich in der Entwicklung.

5 Fazit

MDA und Quasar verfolgen dasselbe Ziel: Die Trennung von Anwendung und Technik, um die immer komplexer werdenden Anwendungen beherrschbar zu machen, die sich ständig revolutionierende Technik von der Anwendung abzukoppeln und allgemein die Zuständigkeiten besser zu trennen. Die Nutzung von Modellen mit einem höheren Abstraktionsniveau als Code wird dabei in beiden Ansätzen gefördert.

Wir sehen die MDA aufgrund der dargestellten Mängel für einen Einsatz in der Breite derzeit als ungeeignet an. Eine Entwicklung dahin ist an die Unsicherheit geknüpft, ob und bis wann das Problem der Dynamikmodellierung gelöst wird.

Die Quasar-Methode hingegen wird bei sd&m mit den heute verfügbaren Sprachen, Middleware-Standards und Datenbanken erfolgreich eingesetzt. Die Unabhängigkeit von Technik durch konsequente Komponentenorientierung und Trennung von Technik und Anwendung sind bereits Wirklichkeit. Damit ist die spezifische Nutzung einer Technologie in einer wieder verwendbaren Komponente konzentriert, ein Austausch wird von dem Rest des Systems nicht wahrgenommen.

Literaturverzeichnis

[Fr03] Frankel, D.: Model Driven Architecture, Wiley, 2003

[KC02] Supporting Model Driven Architecture with eXecutable UML, Kennedy Carter Ltd., CTN 80 v2.2, 2002

[MDA] Model Driven Architecture – Webseiten der OMG, http://www.omg.org/mda/

[Me99] Mellor, S. et al.: Software-Platform-Independent, Precise Action Specification for UML, http://www.projtech.com/pubs/xuml/uml98.pdf

[Me03] Mellor, S.: Executable and Translatable UML, Embedded Systems Programming, Januar 2003.

[OQ] OpenQuasar-Webseiten, http://www.openquasar.de

[Si00] Siedersleben, J., Denert, E.: Wie baut man Informationssysteme?. Informatik-Spektrum 24(1), 2000

[Si03] Siedersleben, J. (Hrsg.): Quasar: Die sd&m Standardarchitektur Teil 1, http://www.sdm.de/de/unternehmen/fe/quasar/index.html