Direktbank: J2EE-Entwicklung mit generativen Techniken

von Marc Thomas und Karsten Thoms

Die MDA-Bank

Der „J2EE vs. .NET-Vergleich" im *Java Magazin* 12.2003 [2] startete eine Vergleichsreihe von Java-Entwicklungsumgebungen anhand des Direktbank-Beispiels. In diesem Artikel wird nun nicht mehr der Tool-Vergleich um ein weiteres Exemplar ergänzt, sondern die methodische Alternative MDA (Model Driven Architecture) im Blickpunkt stehen. In dem bekannten fachlichen Rahmen „Direktbank" werden die Möglichkeiten sowie die Grenzen von MDA deutlich.

Die aktuelle Speerspitze der Object Management Group (OMG) ist MDA [1]. Mit MDA soll der Softwareentwicklungsprozess neu definiert und auf einem abstrakteren Niveau fortgeführt werden. Ziel der modellgetriebenen Entwicklung ist einerseits eine schnellere Entwicklung, andererseits ist vielleicht noch eine höhere Softwarequalität bedeutender, die durch die Verwendung von MDA erwartet wird. Ansatz von MDA ist, die Designmodelle in UML eindeutig zu formalisieren, sodass die Designelemente eine feste Semantik erhalten. Zur Auszeichnung und Formalisierung von Designelementen, wie Klassen, Attribute, Methoden, Aktivitäten etc., lassen sich sowohl Stereotypen als auch *Tagged Values* einsetzen. Somit sind die Typen <<*Presentation*>> und <<*Controller*>> in ihren Bedeutungen eindeutig unterschieden. Dies bedeutet, dass eine eigene Designsprache kreiert wird, deren Syntax als UML-Profil festgehalten wird. Ist dieser Schritt vollzogen, können mithilfe von Generatoren automatische Modelltransformationen durchgeführt werden. Durch die Transformation wird ein plattformunabhängiges Modell (PIM = Plattform Independent Model) in ein plattformabhängiges Modell (PSM = Plattform Specific Model) überführt. Letzteres kann sowohl ein weiteres UML-Modell (Model to Model) oder auch Programmcode (Model to Code) sein. Mit dieser Basis lassen sich spezifische Transformationsregeln anwenden, um in diesem Beispiel auf der Plattform Struts die Form Bean und JSP sowie den Struts Controller und die Referenzen in der *struts-config.xml* zu generieren.

MDA mit dem open Generator FrameWork

Im vorliegenden Beispiel wird das open Generator FrameWork [3] eingesetzt. Dieses Open-Source-Tool ist ein Vertreter der Model-to-Code-Transformatoren und dient der direkten Codegenerierung aus dem vorliegenden PIM. Die gängigen UML-Tools bieten einen Export der Modelle in dem mehr oder weniger standardisierten XMI-Format. Diese Information wird nun durch das Generator FrameWork eingelesen und in einen internen Java-Objektgraphen umgewandelt. Die Java-Objekte sind Instanziierungen der Metaklassen, welche die Generator-interne Repräsentation des UML-Metamodells (siehe unten) darstellen. Mithilfe von Templates werden aus dieser Repräsentation der Java-Code oder sonstige Dateien erzeugt. Eine in sich konsistente Gruppe von Templates und Metaklassen wird als Anwendungsfamilie bezeichnet und konzentriert sich in der Regel auf einen technologischen Bereich, wie z.B. die Nutzung des Struts-Frameworks oder die Entity-Schicht.

Der Umfang der definierten Designsprache ist als UML-Profil oder auch UML-Metamodell zu beschreiben. Leider werden Profile bislang von den Modellierungs-Tools noch nicht unterstützt. Damit könnte bei der Modellierung selbst bereits sichergestellt werden, dass die Designsprache korrekt verwendet wird. So liefert momentan erst der Generator das Ergebnis der formalen Prüfung des UML-Modells.

Zur Unterstützung der verschiedenen XMI-Derivate wird beim Import ein spezifischer, aber austauschbarer Tool-Adapter verwendet. Für die gängigen UML-Tools kommen diese mit dem open Generator FrameWork gleich mit. Das Konzept ist jedoch flexibel genug, nahezu beliebige Datenquellen für die Generierung heranzuziehen. Auf der Ausgabeseite lassen sich mit der proprietären Template-Sprache Xpand alle Arten von textbasierten Dateiformaten erzeugen.

Die im open Generator FrameWork verwendete Technik bietet kein Roundtrip Engineering, das aufgrund der unterschiedlichen Ebenen der Abstraktion von PIM und erzeugtem Code ohnehin mit Vorsicht zu genießen wäre. Im hier verfolgten Ansatz werden Änderungen, die die Abstraktionsebene des PIM betreffen, allein im Modell vorgenommen und durch erneutes Generieren auf die Codeebene übertragen. Die praktische Anwendbarkeit der wiederholten Generierung wird durch das open Generator FrameWork durch geschützte Bereiche (*PROTECTED REGION ID ... PROTECTED REGION END*) im Generat unterstützt. Die Inhalte innerhalb der geschützten Bereiche bleiben bei einem erneuten Generator-Lauf unverändert erhalten, während Modifikationen außerhalb dieser überschrieben werden. Das Konzept des open Generator FrameWorks zeigt sich somit flexibel genug, sowohl unterschiedliche Quellen als auch verschiedene Ziel-

plattformen zu unterstützen. So kann basierend auf einem einzigen PIM zum Zeitpunkt der Generierung die Entscheidung zwischen J2EE und .NET getroffen werden oder, etwas kleiner gedacht, zwischen CMP und einem Persistenz-Framework. Weitere Details zur Theorie von MDA und die vorliegende Interpretation werden sich demnächst in [4] finden.

Architektur und Entwicklungsumgebung

Die Architektur gleicht der in den vergangenen Direktbank-Beispielen eingesetzten (Abb. 1). In der vierschichtigen Architektur finden sich ein Web Gateway mit Struts-Framework sowie ein Web Service Gateway, die beide auf die Business-Schicht aus Session Beans zugreifen. Die Datenanbindung erfolgt über Entity Beans mit Container Managed Persistence. Aufgrund der Ähnlichkeit zu vorhergehenden Varianten soll die Architektur hier nicht weiter erörtert werden.

Die Entwicklungsumgebung stützt sich rein auf frei verfügbare Komponenten. So wird Eclipse als IDE eingesetzt und JBoss mit der Hypersonic-Datenbank als Zielplattform. Für die Modellierung kommt die Community Edition von Poseidon for UML zum Einsatz. Das Zusammenspiel der Komponenten für Generierung, Build und Deployment wird mit Ant koordiniert. Die Bearbeitung der open Generator FrameWork Templates erfolgt über das dazugehörige Eclipse-Plug-in (Abb. 2). Diese Konfiguration entspricht soweit der in der Hands-on-Session der JAX [5] [6] verwendeten. Ergänzt ist sie um die Nutzung der *JBoss.net*-Implementierung für Web Services.

Generieren, aber was?

Bei dem Thema Codegenerierung stellt sich unmittelbar die Frage, welche Bestandteile generiert werden sollen und in welcher Detaillierungstiefe. Insbesondere die Vorstellung von Programmierung in UML ruft nicht ganz unberechtigte Skepsis hervor. In der bestehenden Tool-Landschaft bewegen sich UML-Tools und IDE auf unterschiedlichen Abstraktionsniveaus und zeigen ihre Stärke in dieser Spezialisierung. Heutige Erfahrungen zeigen zudem, dass die Modellierung bis ins letzte Detail sehr

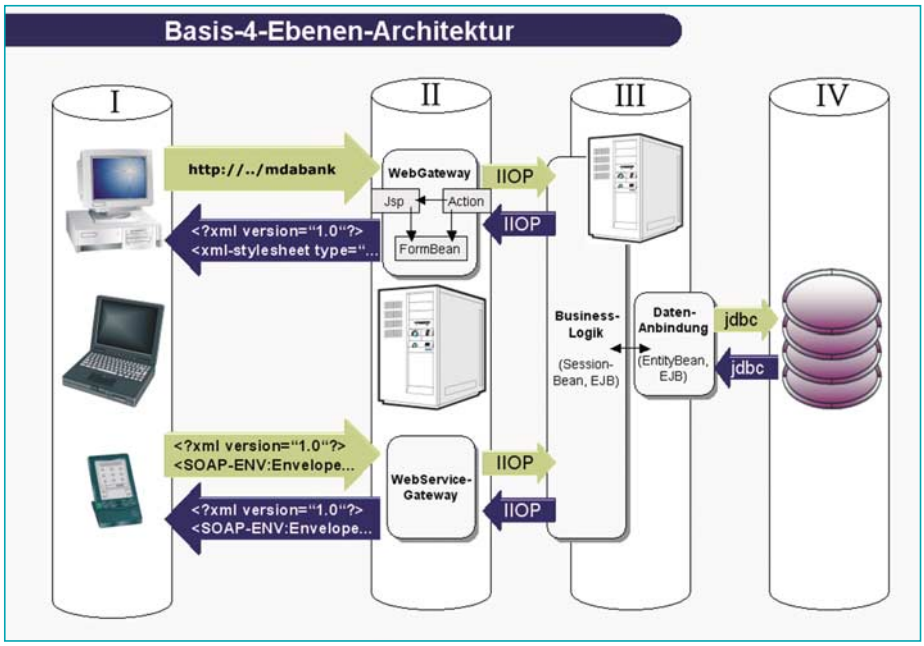

Abb. 1: Vierschichtige Architektur der MDA-Bank

aufwendig, komplex und damit fehleranfällig ist. Das Ziel ist also, nicht alles zu generieren, aber was dann?

Da die Erstellung der Metaklassen und Templates zunächst mit einem zusätzlichen Aufwand verbunden ist, eignet sich zur Generierung jeglicher schematischer Code, der mindestens zwei- bis dreimal verwendet wird. Bei nur einmaliger Nutzung sprechen schließlich neben dem Aufwand die zusätzliche Komplexität durch Modell und Code, die einem leichten Verständnis entgegenstehen kann, gegen die Generierung.

In architekturzentrierten Anwendungsfamilien, wie sie hier verwendet werden, finden sich die schematischen Anteile in dem technischen Code. Dieser umfasst Standard-Deployment-Deskriptoren und Interfaces, wie sie für Enterprise JavaBeans von vielen IDEs ebenfalls erzeugt werden. Hinzu kommen Architektur-Patterns wie Delegates, Service Locator etc. sowie deren Beziehungen. Der fachliche Code im Gegenzug beschreibt die anwendungsspezifischen Anteile wie das Verhalten in den Business-Methoden und die Darstellung in den Views. Dieser lässt sich in der Regel

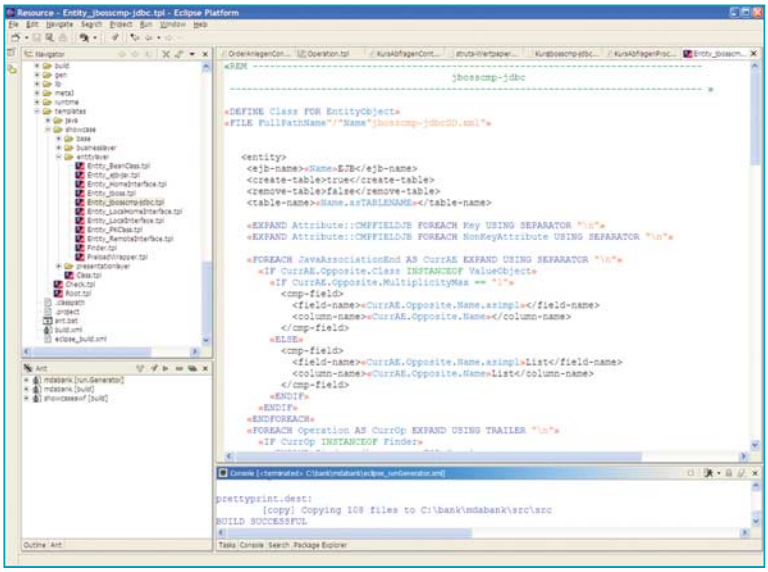

Abb. 2: Generator Template für EJB Deployment Descriptor

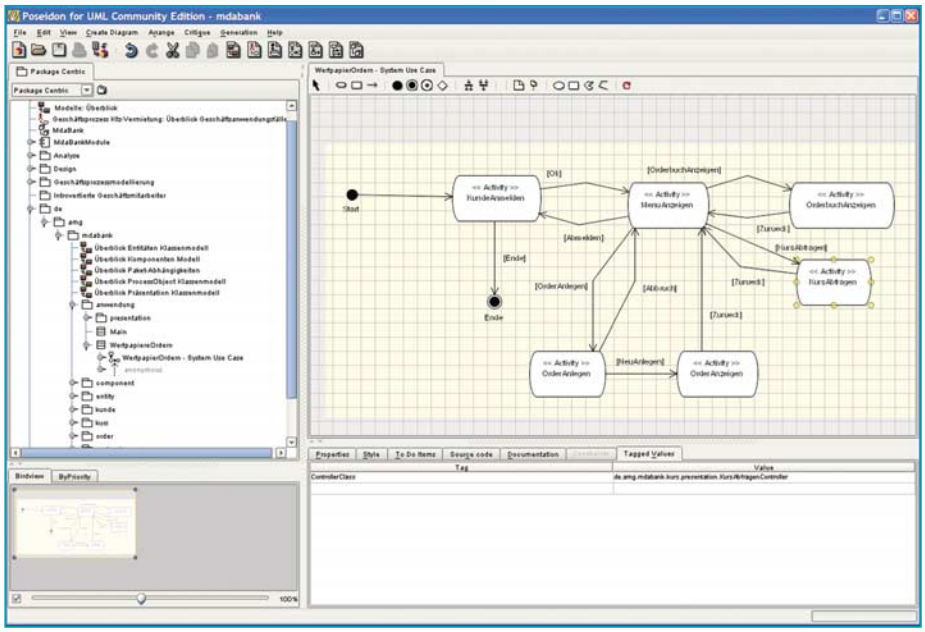

Abb. 3: Aktivitätsdiagramm des *WertpapierOrdern* Use Case

schwer in UML modellieren – und wer möchte schon String-Operationen in UML beschreiben!

Generat und Implementierung

Anhand des Funktionsumfanges „Kurs abfragen“ der MDA-Bank werden wir im Detail in die Schritte Modell, Generat und zu implementierende Fachlichkeit einsteigen. Doch zuvor noch ein Blick auf die gesamte Anwendung, deren Zusammenspiel durch einen ausgezeichneten Main Struts Controller koordiniert wird. Für jede Aktivität wird dann ein weiterer spezifischer Controller eingesetzt. Der Ablauf der Anwendung findet sich als Aktivitätsdiagramm in Abbildung 3 mit der Aktivität *KursAbfragen*. Die Realisierung dieser Aktivität erfolgt durch den *KursAbfragenController*, wobei die Verbindung durch das TaggedValue *ActivityController* hergestellt wird.

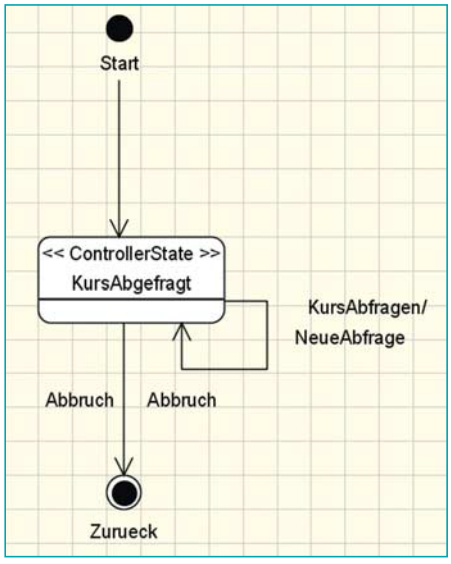

Abb. 4: Zustandsdiagramm des *KursAbfragenController*

Die Steuerung innerhalb einer Aktivität wird mittels eines Zustandsdiagramms (Abb. 4) definiert. Mit Eintritt in die Aktivität wird der Zustand *KursAbgefragt* eingenommen. Für diesen wird eine Initialisierungmethode *initKursAbgefragt ()* generiert, in der die Inhalte der Anzeigeelemente der JSP über die Struts Form vorbereitet werden. Die weitere Navigation nach dieser JSP bzw. aus diesem Zustand heraus wird durch Transitionen und deren Trigger bestimmt. Der Trigger findet sich hierbei als Struts Action wieder. Sind mit der Navigation noch Datenmanipulationen verbunden, ist neben dem Trigger noch ein Effekt, z.B. *abfragenKurs()* im Falle von *NeueAbfrage*, einzutragen. Im Generat befindet sich im *AbfragenKursController* ein entsprechender Methodenrumpf, der mit der Fachlogik zu füllen ist (Listing 1).

Aus dem Zustand *KursAngezeigt* heraus bleiben zwei Möglichkeiten für den Fortgang. Erstens über *NeueAbfrage*, die wiederum in denselben Zustand führt. Zweitens *Abbrechen*, um die Aktivität zu verlassen und der weiteren Navigation entsprechend dem Aktivitätsdiagramm zu folgen.

Mit den bislang genannten Modellelementen ist das zentrale Glied, der Kontrollfluss, vollständig beschrieben. Es fehlt bislang noch der View sowie die Anbindung an die Business-Logik und Datenhaltung. Hierzu ein Blick auf das *KursAbfragen*-Design-Klassenmodell (Abb. 5).

Mit der <<*Presentation*>> werden sowohl die Form Bean als *ViewModel* zwischen Controller und View als auch die JSP generiert. Die einzelnen Anzeigeobjekte lassen sich in der vorliegenden Designsprache noch weiter spezifizieren. Auswahlelemente werden als <<*ComboBox*>> typisiert und reine Anzeigeelemente als <<*Header*>>. Für das Direktbank-Szenario genügt die so erreichte Flexibilität, sodass die JSP nicht weiter angepasst werden muss. Das <<*ProcessObject*>> beinhaltet die Business-Logik und wird als Session

Listing 1

```
...
 public void abfragenKurs() throws Exception {
  //PROTECTED REGION ID
                  (lsmbee0074bfd46b44142ba72a4) START
  KursAbfragenForm cForm = (KursAbfragenForm) form;

  // enter your own initialization code here ...
  //Listeninitialisierung
  KursAbfragenProcessObjectBD businessDelegate =
    KursAbfragenProcessObjectBD.getInstance();
  KursAbfragenValueObject valueObject = null;

  String handelsplatz = cForm.getHandelsplatzauswahl();
  String wertpapier = cForm.getWertpapierauswahl();

  try {
   valueObject = businessDelegate.abfragenKurs
                                        (handelsplatz,
     wertpapier);
  } catch (ApplicationException ae) {
   ActionErrors errors = new ActionErrors();

   errors.add(ActionErrors.GLOBAL_ERROR,
     new ActionError("error.KursAbfragen.Abfrage"));
   saveErrors(request, errors);
  }
  cForm.setWertpapier(valueObject.getWertpapier());
  cForm.setHandelsplatz(valueObject.getHandelsplatz());
  cForm.setWert(valueObject.getWert());
  cForm.setZeit(valueObject.getZeit());
  //PROTECTED REGION END
 }
...
```

Bean mit sämtlichen Bean-Klassen, Interfaces, Deskriptoren für das Deployment sowie das Business Delegate generiert. Manuell implementiert werden muss lediglich der Inhalt der modellierten Business-Methoden *getAbfrageParameter()* und *abfragenKurs()* in der *KursAbfragenProcessObjectBean*. Die Schnittstelle zwischen den Schichten 2 (Web und Service Gateway) und 3 (Business-Logik) wird mittels der modellierten *Value*-Objekte schlank gehalten.

Als Letztes ist noch die Persistenzschicht offen. Jedes *EntityObject* wird als Entity Bean realisiert bzw. generiert, wiederum mit allen Bestandteilen, wie Deployment-Deskriptoren und die *jbosscmp-jdbc.xml*. Erforderliche Finder werden als entsprechend typisierte Methoden modelliert. Lediglich die Datenbankabfrage muss in JBossQL manuell formuliert werden (Listing 2).

Fazit

Der Unterschied zur klassischen codebasierten Softwareentwicklung und der Nutzen des MDA-Ansatzes sollten anhand des betrachteten Beispiels klarer geworden sein. Der Vorteil von MDA, jederzeit ein konsistentes, d.h. der realisierten Wirklichkeit entsprechendes Modell vorweisen zu können, zeigt sich in der Kommunikation mit dem Kunden, da das fachliche Modell für ihn verständlich bleibt. Die in diesem Zusammenhang häufig getätigte Aussage zum prozentualen Anteil der generierten Codezeilen möchten wir uns hier ersparen, da diese Messgröße nur bedingt aussagekräftig ist. Die Entwicklung des fachlichen und somit manuell erstellten Code ist natürlich mit einem größeren Aufwand verbunden als die gleiche Menge manuell geschriebener technischer Code. Dennoch wird MDA dem Anspruch nach schnellerer Entwicklung und insbesondere nach qualitativerer Software (im Sinne von Architekturkonformität) gerecht. Es bleiben jedoch Fragen aufgrund der fehlenden Unterstützung von Profilen durch die UML-Tools und die mehrstufige Entwicklung offen. Auch wird die Fehlersuche durch die verteilten Quellen Modell und fachlicher Code für ungeübte Entwickler nicht gerade vereinfacht. Hier gilt es, das Entwicklungsteam geeignet aufzustellen. Dies führt zur Frage der Organisation von MDA-Projekten, die in diesem Beispiel außen vor geblieben sind. ■

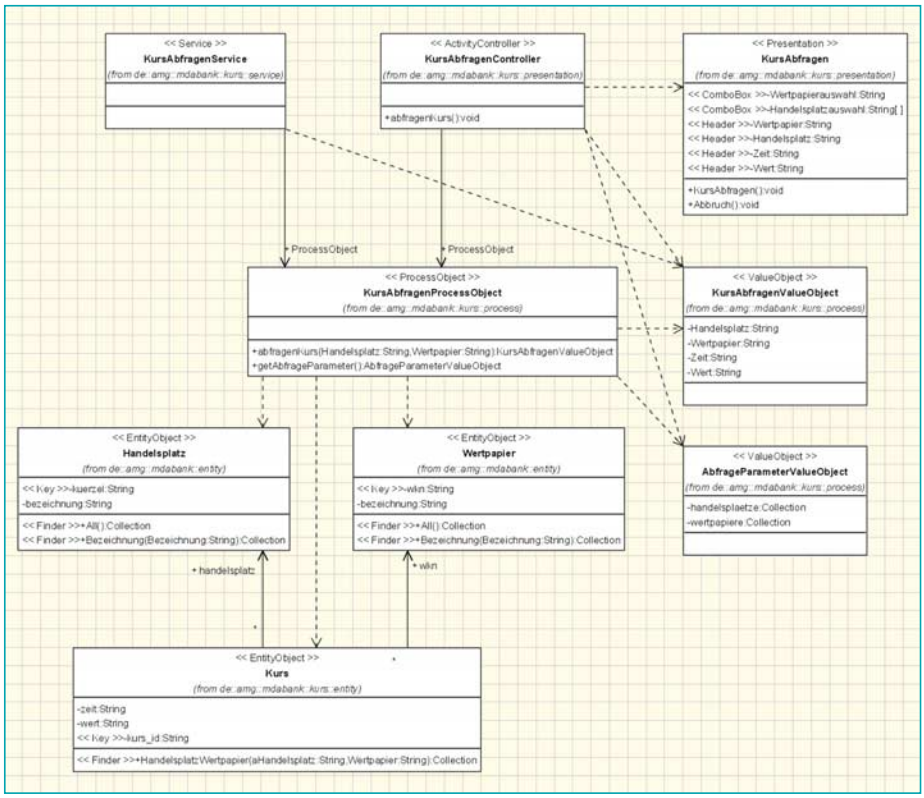

Abb. 5: Klassenmodell für *KursAbfrage*

Listing 2

Ausschnitt der *jbosscmp-jdbc.xml*

```
...
  <query>
   <query-method>
    <method-name>findByHandelsplatzWertpapier
                                    </method-name>
    <method-params>
     <method-param>java.lang.String</method-param>
     <method-param>java.lang.String</method-param>
    </method-params>
   </query-method>
   <!--
   // PROTECTED REGION ID(lsmb14f1726bfd5bbcbfcdba
                    7e88jbossquery_MethodBody) START
   -->
   <jboss-ql>
    <![CDATA[
     SELECT OBJECT(k) FROM KursBean k
     WHERE k.implhandelsplatz.implbezeichnung=
              ?1 AND k.implwkn.implbezeichnung=?2 ]]>
   </jboss-ql>
   <!--
   // PROTECTED REGION END
   -->
  </query>
...
```

Dr. Marc Thomas ist bei itemis als Projektleiter und Coach tätig und begleitet MDA-Einführungsprojekte sowohl organisatorisch als auch technisch.

Karsten Thoms ist Architekt und Trainer bei itemis. Seine Schwerpunkte liegen im Bereich J2EE, in komponentenorientierten Architekturen und der Frameworkentwicklung.

Links & Literatur

[1] MDA Specifications: www.omg.org/mda

[2] Daniel Reiberg: Die Java Bank, in *Java Magazin* 12.2003

[3] open ArchitectureWare: sourceforge.net/projects/architecturware

[4] Tom Stahl, Markus Völter, Jorn Bettin: Modellgetriebene Softwareentwicklung, dpunkt, Herbst 2004

[5] Wolfgang Neuhaus, Martin Schepe, Peter Roßbach: Hands-on-Session: MDA – Schritt für Schritt auf dem eigenen Notebook, JAX 2003

[6] Martin Schepe, Wolfgang Neuhaus: Night School: MDA live auf dem Notebook!, JAX 2004

[7] Martin Schepe, Wolfgang Neuhaus, Peter Roßbach: Praktische Entwicklung mit MDA/D, in *Java Magazin* 9.2003

[8] Architecture Management Group: www.a-m-group.de