Softw Syst Model (2004) 3: 314–327 / Digital Object Identifier (DOI) 10.1007/s10270-004-0061-2

Expert's voice

Model driven architecture: Principles and practice

Alan W. Brown

IBM Software Group, 4205 S. Miami Blvd, Durham NC 27703, USA
E-mail: awbrown@us.ibm.com

Published online: 3 August 2004 – © Springer-Verlag 2004

Abstract. Model Driven Architecture (MDA[1]) is an approach to application modeling and generation that has received a lot of attention in recent months. Championed by the Object Management Group (OMG), many organizations are now looking at the ideas of MDA as a way to organize and manage their application solutions, tool vendors are explicitly referring to their capabilities in terms of "MDA compliance", and the MDA lexicon of platform-specific and platform-independent models is now widely referenced in the industry.

In spite of this interest and market support, there is little clear guidance on what MDA means, where we are in its evolution, what is possible with today's technology, and how to take advantage of it in practice. This paper addresses that need by providing an analysis of how modeling is used in industry today, the relevance of MDA to today's systems, a classification of MDA tooling support, and examples of its use. The paper concludes with a set of recommendations for how MDA can be successful in practice.

Keywords: Software architecture – Software design – Unified Modeling Language (UML)

1 Introduction

Building enterprise-scale software solutions has never been easy. The difficulties of understanding highly complex business domains are typically compounded with all the challenges of managing a development effort involving large teams of engineers over multiple phases of a project spanning many months. The time-to-market pressures inherent to many of today's product development efforts only serve to compound the problems.

In addition to the scale and complexity of many of these efforts, there is also great complexity to the software platforms for which enterprise-scale software are targeted. Most IT organizations rely on a complicated assortment of infrastructure technologies that have evolved over multiple years, consist of a variety of middleware software acquired from many vendors, and assembled through various poorly documented integration efforts of varied quality.

To develop applications in this context requires an approach to software architecture that helps architects evolve their solutions in flexible ways, reusing existing efforts in the context of new capabilities that implement business functionality in a timely fashion even as the target infrastructure itself is evolving. Two important ideas are now considered central to how to address this: Service-Oriented Architectures and Software Product Lines.

1.1 Service oriented architectures

An approach gaining a lot of support in the industry today is based on viewing enterprise solutions as federations of services connected via well-specified contracts that define their service interfaces. The resulting system designs are frequently called Service Oriented Architectures (SOAs) [1]. Systems are composed of collections of services making calls on operations defined through their service interfaces. Many organizations now express their solutions in terms of services and their interconnections.

A number of important technologies have been defined to support an SOA approach, most notably when the services are distributed across multiple machines and connected by the Internet. These web service approaches rely on intra-service communication protocols such as the Simple Object Access Protocol (SOAP), allow the web

[1] Model Driven Architecture (MDA) is a Registered Trade Mark of the Object Management Group.

service interfaces (expressed in the Web Services Definition Language – WSDL) to be registered in public directories and searched in Universal Description, Discovery and Integration (UDDI) repositories, and share information in documents defined in the eXtensible Markup Language (XML) and described in standard schemas.

1.2 Software product lines

Organizations are also beginning to recognize that there frequently is a great deal of commonality in the systems they develop. Recurring approaches are seen at every level of an enterprise software project, from having standard domain models that capture core business processes and domain concepts, to the way in which developers implement specific solutions to realize designs in code. A great deal of efficiency can be gained if patterns can be defined by more skilled developers and propagated across the IT organization.

Recognizing this, many organizations are moving toward a software product line view of their development in which planned reuse of assets is supported, and an increasing level of automation can be used to realize solutions for large parts of the systems being developed [2, 3]. More generally, the underpinning of any product line approach can be viewed as a way to transform descriptions of a solution at one level of abstraction into descriptions at a lower level of abstraction by applying well-defined patterns.

1.3 The new development imperative

Recognizing the need for greater flexibility in the development of enterprise-scale solutions, the OMG has created a conceptual framework[2] that separates business-oriented decisions from platform decisions to allow greater flexibility when architecting and evolving these systems. This conceptual framework, and the standards that help realize it, OMG calls Model Driven Architecture (MDA). Application architects use the MDA framework as a blueprint for expressing their enterprise architectures, and employ the open standards that are inherent to MDA to define standard transformations between models as their "future proofing" against vendor lock-in and technology churn.

The OMG's MDA approach provides an open, vendor-neutral approach to system interoperability via OMG's established modeling standards: Unified Modeling Language (UML), Meta-Object Facility (MOF), and Common Warehouse Meta-model (CWM). Platform-independent descriptions of enterprise solutions can be built using these modeling standards and can be transformed into a major open or proprietary platform, including CORBA, J2EE, .NET, XMI/XML, and Web-based platforms.

While this all sounds fine in theory, what is the reality in practice? This paper looks at the principles and practice of MDA, and provides a perspective on the current state of the practice in MDA as realized in IBM's modeling, design, and implementation technologies.

1.4 Related work

The ideas of MDA are becoming widely discussed in the software industry, and there are a number of sources of help for those interested in applying MDA in practice. The three primary sources are:

- *OMG Materials*. The OMG provides the primary source for many MDA ideas, consisting of a number of specifications, whitepapers, and presentation materials that discuss the MDA approach (see www.omg.org). These tend to be at two levels of detail – either they are detailed specifications aimed at technologists implementing those specifications, or they are high level overviews of concepts and standards aimed at positioning the MDA approach. A small, but growing, set of materials are aimed at filling the space between these two extremes for those wanting to understand more about MDA in the context of current development approaches, and how MDA can be applied in practice.
- *Books and papers*. Recognizing the gaps in OMG materials, a number of books and papers are currently appearing in print. The three primary texts available are from Kleppe et al. [4], Frankel [5], and Mellor et al. [6]. These offer perspectives on the key OMG standards and express general comments on their value, supported by brief practical insights into MDA in practice.
- *Broader industry and academic materials*. As the MDA approach gains support, a number of materials are becoming available that addresses practical application of MDA, its strengths, and its limitations. Currently, this material is very variable in focus, depth, and quality. Typically, these discussions focus around specific technology solutions. An Internet search on this topic will reveal many of these materials.

In this paper we complement many of these existing materials with an overview of the principles and practice of MDA appropriate to the needs of today's software practitioners requiring a balanced overview of MDA and its role in development.

2 Why model?

Models provide abstractions of a physical system that allow engineers to reason about that system by ignoring extraneous details while focusing on the relevant ones. All forms of engineering rely on models as essential to un-

[2] In this context a conceptual framework is a set of key concepts and structures that guides the planning, understanding, and realization of an enterprise solution.

derstand complex real world systems. Models are used in many ways: predicting system qualities, reasoning about specific properties when aspects of the system are changed, and communicating key system characteristics to its various stakeholders. The models may be developed as a precursor to implementing the physical system, or they may be derived from an existing system or a system in development as an aid to understanding its behavior [7].

2.1 Views and model transformation

Because there are many aspects of a system that may be of interest, various modeling concepts and notations may be used that highlight one or more particular perspectives, or views, of that system depending on what is considered relevant at any point in time. Furthermore, in some instances the models can be augmented with hints, or rules, that assist in transforming them from one representation to another. It is often necessary to convert between different views of the system at an equivalent level of abstraction (e.g., between a structural view and a behavioral view), and a model transformation facilitates this. In other cases a transformation converts models offering a particular perspective between levels of abstraction, usually from a more abstract to less abstract view by adding more detail supplied by the transformation rules.

2.2 Models, modeling, and MDA

Models and model-driven software development are at the heart of the MDA approach. So it is appropriate to start by looking at what is being practiced when enterprise application developers take advantage of modeling.

In the software engineering world, modeling has a rich tradition from the earliest days of programming. The most recent innovations have focused on notations and tools that allow users to express system perspectives of value to software architects and developers in ways that are readily mapped into the programming language code that can be compiled for a particular operating system platform. The current state of this practice employs the Unified Modeling Language (UML) as the primary modeling notation [8]. The UML allows development teams to capture a variety of important characteristics of a system in corresponding models. Transformations among these models are primarily manual, with tool support for managing traceability and dependency relationships among modeling elements, supported by best practice guidance on how to maintain synchronized models as part of a large-scale development effort [9].

One useful way to characterize current practice is to look at the different ways in which the models are synchronized with the source code they help describe. This is illustrated in Fig. 1[3], which shows the spectrum of mod-

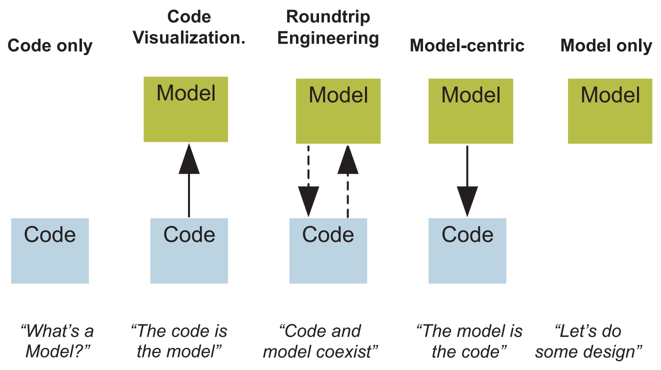


Fig. 1. The modeling spectrum

eling approaches in use by software practitioners today. Each category identifies a particular use of models in assisting software practitioners to create running applications (code) for a specific runtime platform, and the relationship between the models and the code.[4]

Today, a majority of software developers still take a **code-only** approach (see the left end of the modeling spectrum, Fig. 1), and do not use separately defined models at all. They rely almost entirely on the code they write, and they express their model of the system they are building directly in a 3rd generation programming language such as Java, C++, or C# within an Integrated Development Environment (IDE) such as IBM WebSphere Studio, Eclipse, and Microsoft VisualStudio.[5] Any "modeling" they do is in the form of programming abstractions embedded in the code (e.g., packages, modules, interfaces, etc.), which are managed through mechanisms such as program libraries and object hierarchies. Any separate modeling of architectural designs is informal and intuitive, and lives on whiteboards, in PowerPoint sides, or in the developers' heads. While this may be adequate for individuals and very small teams, this approach makes it difficult to understand key characteristics of the system among the details of the implementation of the business logic. Furthermore, it becomes much more difficult to manage the evolution of these solutions as their scale and complexity increases, as the system evolves over time, or when the original members of the design team are not directly accessible to the team maintaining the system.

An addition is to provide **code visualizations** in some appropriate modeling notation. As developers create or analyze an application, they often want to visualize the code through some graphical notation that aids their understanding of the code's structure or behavior. It may also be possible to manipulate the graphical notation as an alternative to editing the text based code, so

[3] Figure 1 is based on a diagram originally created by John Daniels.

[4] Many other important life-cycle artifacts also benefit from a model-driven approach (e.g., requirements lists, test cases, and build scripts). For simplicity we concentrate on the primary development artifact – the code.

[5] For this discussion we shall ignore the fact that the code is itself a realization of a programming model that abstracts the developer from the underlying machine code for manipulating individual bits in memory, registers, etc.

that the visual rendering becomes a direct representation of the code. Such rendering is sometimes called a code model, or an implementation model, although many feel it more appropriate to call these artifacts "diagrams" and reserve the use of "model" for higher levels of abstraction. In tools that allow such diagrams (e.g., IBM WebSphere Studio and Borland Together/J), the code view and the model view can be displayed simultaneously; as the developer manipulates either view the other is immediately synchronized with it. In this approach, the diagrams are tightly coupled representations of the code and provide an alternative way to view and possibly edit at the code level.

Further advantage of the models can be taken through **roundtrip engineering (RTE)** between an abstract model of the system describing the system architecture or design, and the code. The developer typically elaborates the system design to some level of detail, then creates a first-pass implementation from that code by applying model-to-code transformations, usually manually. For instance, one team working on the high level design provides design models to the team working on the implementation (perhaps simply by printing out model diagrams, or providing the implementation team some files containing the models). The implementation team converts this abstract, high-level design into a detailed set of design models and the programming language implementation. Iterations of these representations will occur as errors and their corrections are made in either the design or the code. Consequently, without considerable discipline, the abstract models and the implementation models usually – and quickly – end up out of step.

Tools can automate the initial transformation, and can help to keep the design and implementation models in step as they evolve. Typically the tools generate code stubs from the design models that the user has to further refine.[6] As changes are made to the code they must at some point be reconciled with the original model (hence the term "roundtrip engineering," or RTE). To achieve this some approach to recognize generated versus user-defined code is used such as placing markers in the code. Tools adopting this approach, such as IBM Rational Rose, can offer multiple transformation services supporting RTE between models and different implementation languages.

In a **model-centric** approach, models of the system are established in sufficient detail that the full implementation of the system can be generated from the models themselves. To achieve this, the models may include, for example, representations of the persistent and non-persistent data, business logic, and presentation elements. Any integration to legacy data and services may require that the interfaces to those elements are also modeled. The code generation process may then apply a series of patterns to transform the models to code, frequently allowing the developer some choice in the patterns that are applied (e.g., among various deployment topologies). To further assist in the code generation, this approach frequently makes use of standard or proprietary application frameworks and runtime services that ease the code generation task by constraining the styles of applications that can be generated. Hence, tools using this approach typically specialize in the generation of particular styles of applications (e.g., IBM Rational Rose Technical Developer for real-time embedded systems, and IBM Rational Rapid developer for enterprise IT systems). However, in all cases the models are the primary artifact created and manipulated by developers.

A **model-only** approach is at the far-right end of the modeling spectrum. In this approach developers use models purely as thought aids in understanding the business or solution domain, or for analyzing the architecture of a proposed solution. Models are frequently used as the basis for discussion, communication, and analysis among teams within a single organization, or across multi-organizational projects. These models frequently appear in proposals for new work, or adorn the walls of offices and cubes in software labs everywhere as a way of understanding some complex domain of interest, and establishing a shared vocabulary and set of concepts among disparate teams. In practice the implementation of a system, whether from scratch or updating an existing solution, may be practically disconnected from the models. An interesting example of this approach can be seen in the growing number of organizations who outsource implementation and maintenance of their systems while maintaining control of the overall enterprise architecture.

2.3 From models to MDA

Modeling has had a major impact on software engineering, and it is critical to the success of every enterprise-scale solution. However, there is great variety in what the models represent and how those models are used. An interesting question is: which of these approaches can we describe as "model-driven?" If I create a visualization of some part of a system, does that mean I am practicing MDA? Unfortunately, there is no definitive answer. Rather, there is a growing consensus that MDA is more closely associated with model-driven approaches in which code is (semi-) automatically generated from more abstract models, and which employs standard specification languages for describing those models and the transformations between them. We explore this in the next section.

3 MDA principles

There are many views and opinions about what MDA is and is not. However, the most authoritative view is provided by the Object Management Group (OMG), an

[6] In some cases much more than code stubs can be generated depending on the fidelity of the models.

industry consortium of more than 800 companies, organizations, and individuals [10]. Why does the OMG's view of MDA matter so greatly? As an emerging architectural standard, MDA falls into a long tradition of OMG support and codification of numerous computing standards over the past two decades. The OMG has been responsible for the development of some of the industry's best-known and most influential standards for system specification and interoperation, including the Common Object Request Broker Architecture (CORBA), OMG Interface Definition Language (IDL), Internet Inter-ORB Protocol (IIOP), Unified Modeling Language (UML), Meta Object Facility (MOF), XML Metadata Interchange (XMI), Common Warehouse Model (CWM), and Object Management Architecture (OMA). Additionally, OMG has enhanced these specifications to support specific industries such as healthcare, manufacturing, telecommunications, and others.

The OMG has refocused its strategy, standards, and positioning to support the MDA approach. They are promoting MDA as a way to develop systems that more accurately satisfy customers' needs, and offer more flexibility in system evolution. The MDA approach builds on the earlier system specification standards work, providing a comprehensive interoperability framework for defining interconnected systems.

3.1 The principles of MDA

There are four principles that underlie the OMG's view of MDA:

– Models expressed in a well-defined notation are a cornerstone to understanding systems for enterprise-scale solutions.
– The building of systems can be organized around a set of models by imposing a series of transformations between models, organized into an architectural framework of layers and transformations.
– A formal underpinning for describing models in a set of metamodels facilitates meaningful integration and transformation among models, and is the basis for automation through tools.
– Acceptance and broad adoption of this model-based approach requires industry standards to provide openness to consumers, and foster competition among vendors.

To support these principles, the OMG has defined a specific set of layers and transformations that provide a conceptual framework and a vocabulary for MDA. Notably, OMG identifies four types of models: Computation Independent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM) described by a Platform Model (PM), and an Implementation Specific Model (ISM) [11].

It is important to point out that for MDA a "platform" is meaningful only relative to a particular point of view – one person's PIM is another person's PSM. For example, a model may be a PIM with respect to choice of communication middleware if that model does not prescribe a particular choice of middleware technology. However, when a decision is made to use a particular middleware such as Common Object Request Broker Architecture (CORBA), the model is transformed to a CORBA-specific PSM. The new model may still be a PIM with respect to choice of Object Request Broker (ORB) – and certainly with respect to target operating system and hardware. This is illustrated in Fig. 2.

As a result, an MDA tool may support transforming a model in several steps from initial analysis model to executable code. For instance, the pattern facility of IBM Rational XDE supports this type of multi-transformation development. Alternatively, a tool (such as IBM Rational Rose Technical Developer) may transform a model from UML to an executable code in a single step.

MDA practitioners recognize that transformations can be applied to abstract descriptions of aspects of a system to add detail, make the description more concrete, or convert between representations. Distinguishing among different kinds of models allows us to think of software and system development as a series of refinements between different model representations. These models and their refinements are a critical part of the development methodology for situations that include refinements between models representing different aspects of the system, adding further details to a model, or converting between different kinds of models.

Three ideas are important here with regard to the abstract nature of a model and the detailed implementation it represents:

– **Model classification:** We can classify software and system models in terms of how explicitly they represent aspects of the platforms being targeted. In all

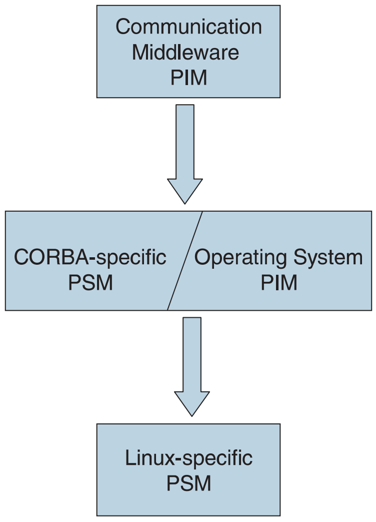


Fig. 2. An example of PIM to PSM transformations

software and system development there are important constraints implied by the choice of languages, hardware, network topology, communications protocols and infrastructure, and so on. Each of these can be considered elements of a solution "platform". An MDA approach helps us to focus on what is essential to the business aspects of a solution being designed separate from the details of that "platform."

- **Platform independence:** The notion of a "platform" is rather complex and highly context dependent. For example, in some situations the platform may be the operating system and associated utilities; in some situations it may a technology infrastructure represented by a well-defined programming model such as J2EE or .Net; in other situations it is a particular instance of a hardware topology. Whatever we consider the "platform," it is important to think more in terms of models at different levels of abstraction used for different purposes, rather than be too distracted by defining the "platform."
- **Model transformation and refinement:** By thinking of software and system development as a set of model refinements, the transformations between models become first class elements of the development process. This is important because a great deal of work takes places in defining these transformations, often requiring specialized knowledge of the business domain, the technologies being used for implementation, or both. Today, these transformations are implicitly defined by skilled architects in a software project. We can improve the efficiency and quality of systems by capturing these transformations explicitly and reusing them consistently across solutions. Where the different abstract models are well-defined, we can use standard transformations. For example, between design models expressed in UML and implementations in J2EE, we can in many cases use well-understood UML-to-J2EE transformation patterns that can be consistently applied, validated, and automated.

Underlying these model representations, and supporting the transformations, the models are described in a set of metamodels. The ability to analyze, automate, and transform models requires a clear, unambiguous way to describe the semantics of the models. Hence, the models intrinsic to a modeling approach must themselves be described in a model, which we call a metamodel. So, for example, the semantics and notation of the UML are described in metamodels. Tool vendors turn to the standard metamodels of UML when they want to implement the UML in a standard way. The UML metamodel describes in precise detail the meaning of a class, the meaning of an attribute, and the meaning of the relationships between these two concepts.

The OMG recognizes the importance of metamodels and formal semantics for modeling as essential for their practical use, and they have defined a set of metamodeling levels as well as a standard language for expressing metamodels: the Meta Object Facility (MOF). A metamodel uses MOF to formally define the abstract syntax of a set of modeling constructs.

The models and the transformations between them can be specified using open standards. As an industry consortium, the OMG has championed a number of important industry standards for specifying systems and their interconnections. Through standards such as CORBA, IIOP, UML, and CWM, the software industry can take advantage of a level of system interoperability that was previously impossible, Furthermore, tool interoperation is also facilitated as a result of tool interchange standards such as MOF and XMI.

3.2 A simple example

In Fig. 3 we show a simplified example of a platform independent model (PIM) and its transformation into three different platform-specific models (PSM).

Figure 3 shows a simple PIM representing a Customer and Account. At this level of abstraction, the model describes important characteristics of the domain in terms of classes and their attributes, but without making any platform-specific choices about which technologies will be used to represent them. Specific mappings, or transformations, will be defined to create the PSMs. Three are illustrated, together with the standards that are used to express the mappings. For example, one approach is to take the PSM expressed in UML and export it in XMI format using standard definitions expressed as XML Schema Definitions (XSD) or Document Type Definitions (DTD). This can then be used as input to a code generation tool that produces interface definitions in Java for each of the classes defined in the UML. Usually, a set of rules are built into the code generation tool to perform the transformation. However, the code generation tool often allows those rules to be specifically defined as templates in a scripting language.[7]

3.3 MDA theory in a nutshell

Following a long history of the use of models to represent key ideas in both problem and solution domains, MDA provides a conceptual framework for using models and applying transformations between models as part of controlled, efficient software development process. Here are the basic assumptions and parameters governing MDA usage today:

- Models help people understand and communicate complex ideas.

[7] This includes commercial examples of MDA in action such as IBM Rational's Rose Technical Developer or Rapid Developer products (`http://www.ibm.com/rational`), and open source MDA tools applying this approach (e.g., AndroMDA (`http://www.andromda.org`) or Jamda (`http://jamda.sourceforge.net`).

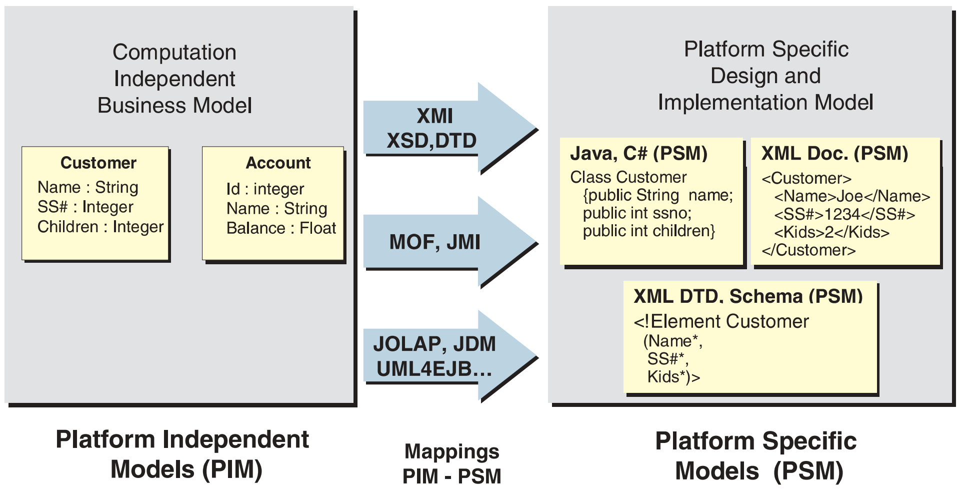


Fig. 3. A simplified example of PIM to PSM mappings

– Many different kinds of elements can be modeled depending on the context. These offer different views of the world that must be reconciled.
– We see commonality at all levels of these models – in both the problems being analyzed, and in the proposed solutions.
– Applying the ideas of different kinds of models and transforming them between representations provides a well-defined style of development, enabling the identification and reuse of common approaches.
– The OMG has provided a conceptual framework and a set of standards to express models, model relationships, and model-to-model transformations in what it calls "model driven architecture".
– Tools and technologies can help to realize this approach, and make it practical and efficient to apply.

4 MDA in practice

4.1 MDA tools

It is well known and well documented that the availability of quality tool support is a critical factor in the successful adoption of new practices. So looking at the tools market for MDA offers useful insight into current practice. There are a number of tools in the market today that advertise support for MDA. For example, at a recent OMG-sponsored meeting in at the end of 2003, over 40 vendors offered MDA tool solutions.

While there is a good deal of hype and bluster, there is also some substance to the vendors' claims of support for MDA, and it is useful to understand how the market for MDA solutions is evolving. Although there is a good deal of variety and innovation in the MDA tools market, the current offerings can be characterized as falling onto one of three primary categories:

– **Code generation based on templates**. These vendor offerings focus on generating code from higher level abstractions based on customizable patterns and templates (e.g., IBM Rational XDE, Codagen's Codagen Architect, and ioSoftware's ArcStyler). Typically, the expert architects are responsible for writing the code generation rules within the templates, which are then handed off to the rest of the team for use.
– **Modeling language extensions**. These vendor offerings enhance existing modeling approaches with richer semantics to facilitate automating the mapping to specific platforms (e.g., IBM Rational Rose Technical Developer, and Kennedy Carter's iUML,). Detailed action semantics are defined as part of the models such that models can be directly executed, or the complete system code can be generated from the models.
– **Architected RAD solutions**. These vendor offerings realize an abstract programming model that relies on an architectural framework and predefined technology patterns to generate an application for a specified target platform, or use their own proprietary runtime above the target platform (e.g., IBM Rational Rapid Developer, Versata's Versata Logic Studio, M7's Application Assembly Suite, Kabira's Kabira Design Center, and Compuware's OptimalJ).

There is a great deal of excitement around this new generation of MDA tools, and they offer a great deal of promise for automating large parts of the development of enterprise-scale solutions. All of these categories of tools are seeing success in the market, although current revenue from these products remains relatively small.

However, one recent study highlights some of the advantages of use of MDA tools. The Middleware Company performed an analysis that compared a model-driven, pattern-based development paradigm with a traditional code-centric development approach. The results showed that the MDA team, using IBM Rational's Rapid Developer product, demonstrated significant productivity and quality improvements over the traditional development approach. In summary, the MDA team developed the same application is less than 50 hours compared with almost 500 hours for the traditional team, and reported zero bugs.[8]

4.2 MDA examples

Having analyzed the range of tooling available for MDA, it is useful to briefly examine some examples of MDA in practice. Here we provide an overview of two example scenarios of MDA use. These are all drawn from real world examples of commercial application of MDA using the IBM products, but obviously greatly simplified for presentation in this paper.

4.2.1 Example 1: Connecting business and IT

One of the primary challenges to be addressed in developing enterprise-scale solutions is to connect the domain-specific requirements expressed by business analysts with the technology-specific solutions designed by IT architects. Typically, the connection between these two disparate worlds is very low bandwidth – the two communities have very different skills, use different modeling concepts and notations (if at all), and rarely understand the mapping between those concepts. The IBM Software Development Platform is intended to assist with this problem. In particular, the integration of process, assets, and deliverables is aimed at connecting these two different aspects of the system in a precise, automated way.

[8] As a single data point, this study should of course be considered illustrative rather than definitive. For further details see `http://www.middleware-company.com/casestudy/`.

An example of this in practice let's consider how we would approach the problem of redesigning an airline's flight planning system. The process begins by modeling the business process in an intuitive, easy to use notation accessible to business analysts, as illustrated in Fig. 4.

As shown in Fig. 4, a business process model captures the key business activities and workflows, in this case using the IBM WebSphere Business Integration Modeler product. This allows the current system of automated and manual steps to be understood, and potential changes to the system to be designed, simulated, and costed before any changes to the business process are committed to by the organization.

Once completed, decisions can be made about which pieces of the new business process should be automated in software. These can be automatically transformed into an initial set of use cases for the proposed system. In this illustration the activities are automatically transformed into use cases in IBM Rational Rose XDE. The mapping between business activities and use cases can initially be quite straightforward, and then the resulting use cases can be elaborated to add further detail, as illustrated in Fig. 5. This realizes the mapping from the domain-specific concepts of the business analyst into technology-specific concepts of the IT architect (the CIM to PIM mapping).

At this point an initial high level architecture of a solution should be proposed and refined. A number of patterns are available to guide the architect in choosing an architecture. In this example the "IBM Patterns for e-Business" provide one set of architectural solutions that have proved to be useful in practice [12]. The architect selects one of these patterns that matches the particular characteristics of his/her problem domain and bind the various variability points in the pattern to previously de-

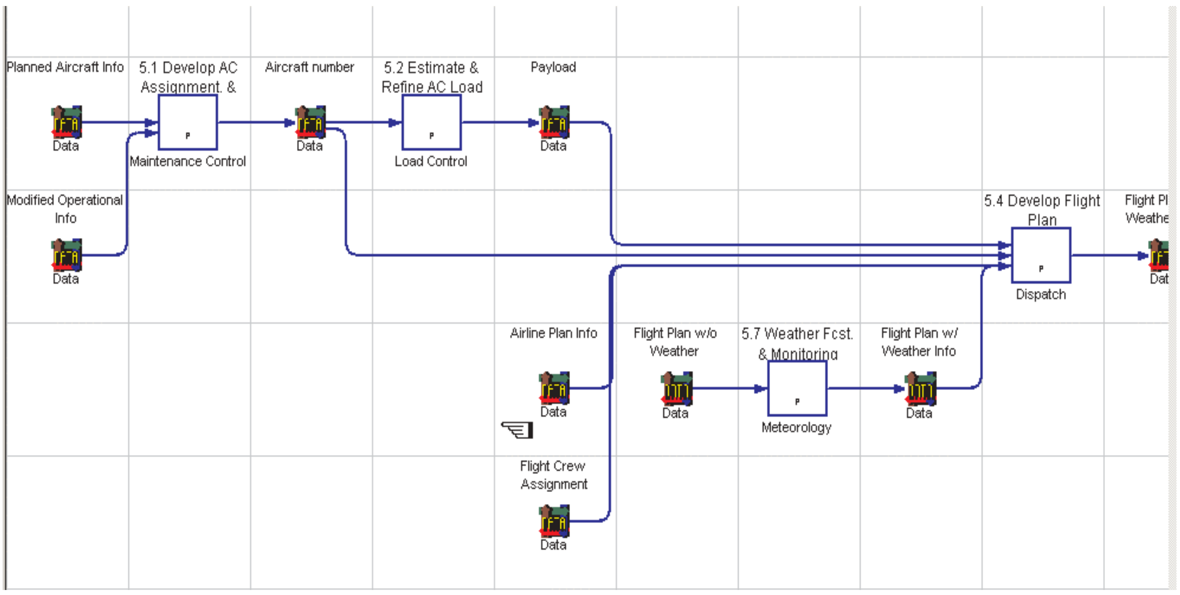


Fig. 4. The business process model (extract)

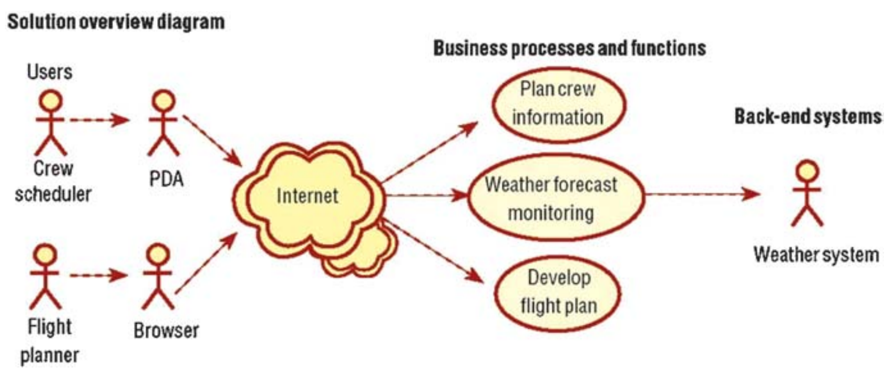


Fig. 5. A simplified solution overview diagram

fined model elements. The result, as illustrated in Fig. 6, provides the initial solution blueprint. In this example the transformation is realized by applying a predefined IBM e-business pattern in the IBM Rational Rose XDE product.

Patterns such as the "IBM Patterns for e-Business" provide strategies for refining the initial solution through the application of further patterns that transform abstract model elements into more concrete model elements. Eventually, this results in the application of a set of deployment patterns that realize the mapping of the solution to a specific physical topology, as illustrated in Fig. 7. This completes the mapping into the underlying technologies of choice (the PIM to PSM mapping).

Further refinements would then take place to add details of the particular platform products that had been chosen and the physical characteristics of those products (e.g., the particular application server, messaging infrastructure, and database management system). Reuse of domain knowledge from previous systems in flight planning and related domains would typically help to automate this refinement.

Additionally, the behavior of the system must be realized by connecting to existing services, developing new functionality, and so on. Here, traditional UML modeling approaches can be applied such as class modeling for describing the data being manipulated, behavioral modeling to describe the business logic and workflow, and user interface modeling to define the user interactions. Here again, patterns are applied to transform these models, with the final step being patterns expressed as code templates that generate code from model elements based on a predefined set of transformation rules.

An important result of this real world example is that the client organization is able to gain visibility in the process of connecting domain-specific business needs into a technology-specific solution following a repeatable, predictable process. The lessons of this example are now being applied in a family of similar solutions in the transportation domain.

4.2.2 Example 2: Platform abstraction

In creating the next generation banking system at a large multinational bank, it was recognized that many develop-

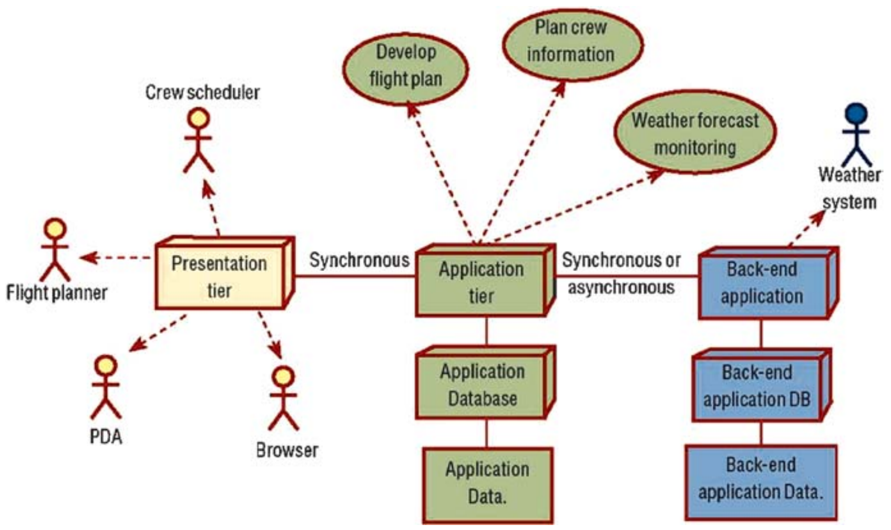


Fig. 6. An architectural model

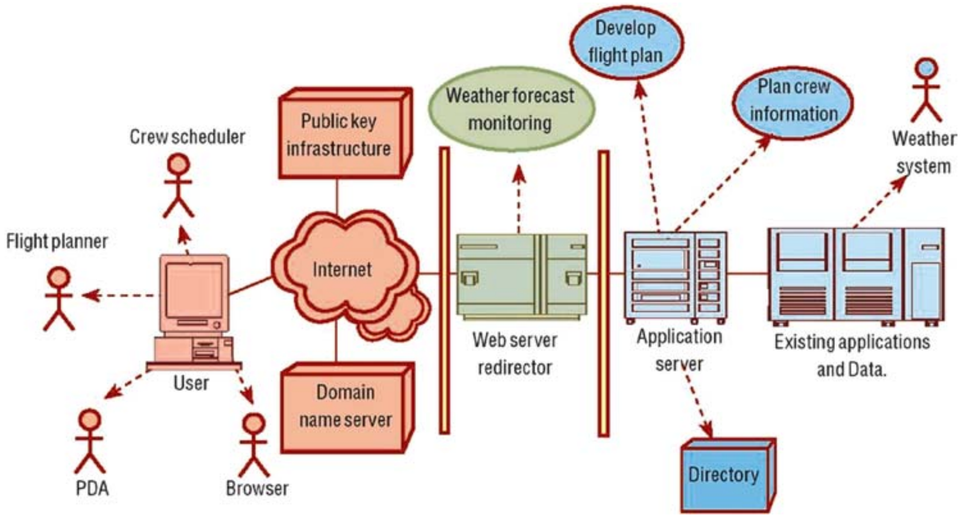


Fig. 7. A deployment model

ers in the development team were not highly skilled in the necessary J2EE and distributed architecture skills. Furthermore, the planned length of the project meant that changes in the underlying platform-specific J2EE technologies were inevitable. As a result, the organization looked to MDA as an approach that would allow them to overcome these challenges. They wanted to have their small team of skilled architects develop a platform independent model of the new system, and to create transforms that would generate large parts of the solution from these high level models. As the high level design is refined, the less J2EE-literate developers would use these transforms in realizing the final solution.

To illustrate the value this MDA approach in practice, a proof of concept was developed using the IBM Rational Rose XDE product. The first step was the development of a PIM of the system, as illustrated in Fig. 8.

The piece of the PIM shown in Fig. 8 illustrates the high level model of the solution specified in UML. A set of patterns for this PIM were developed that transform this PIM into a PSM, as illustrated in Fig. 9. These patterns mapped elements of the PIM into PSM elements based on

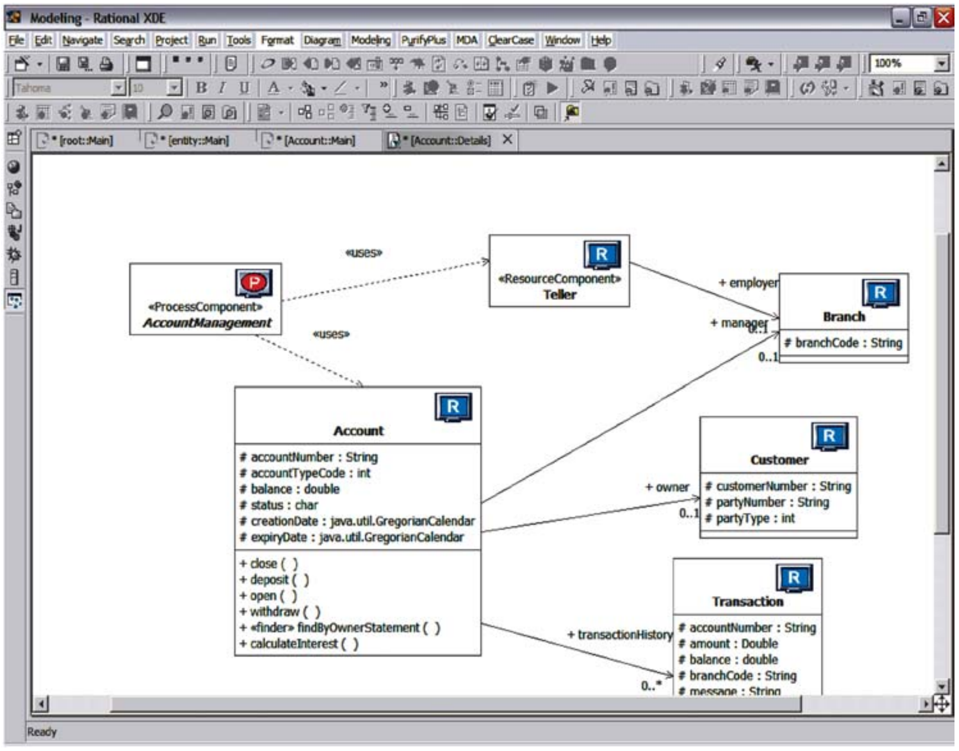


Fig. 8. The PIM (extract)

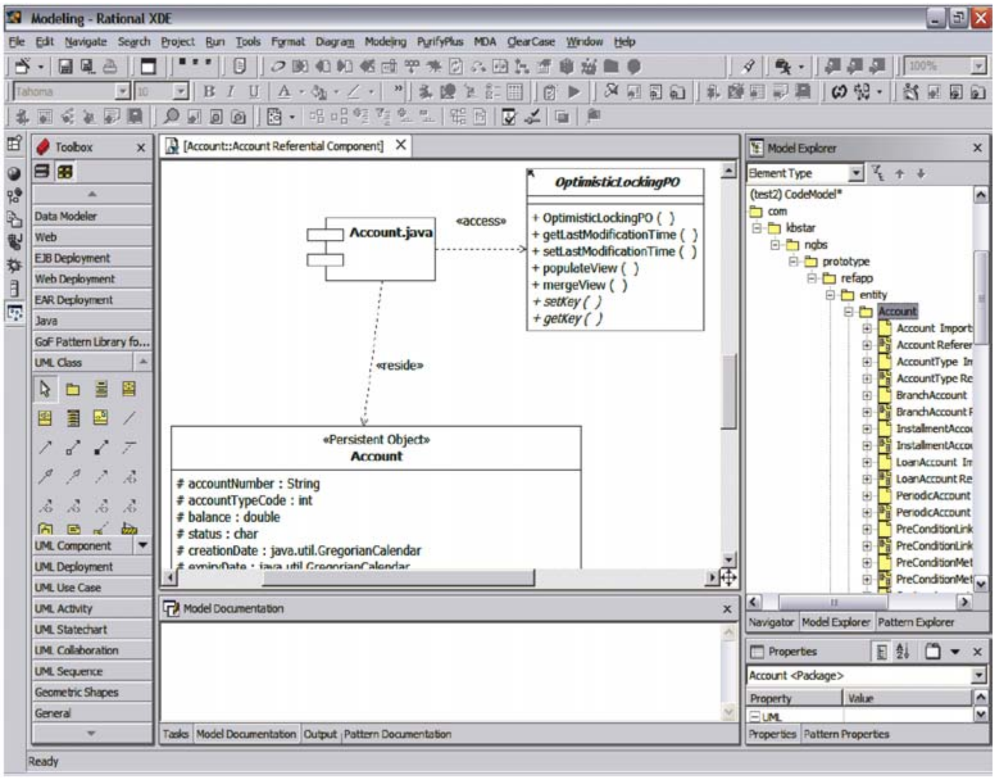
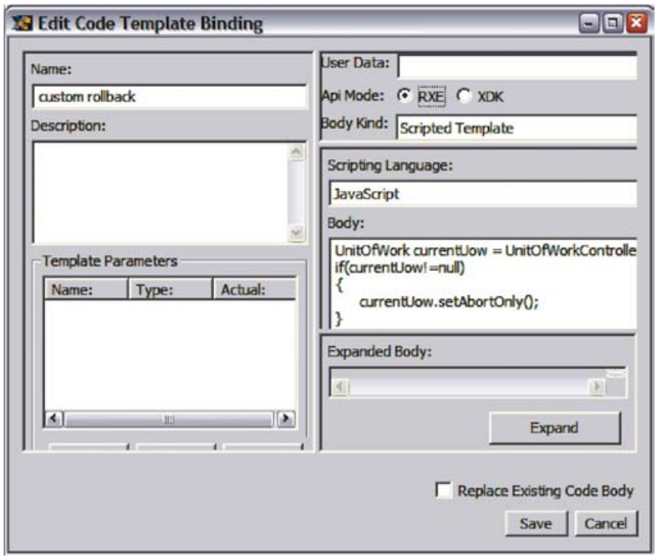


Fig. 9. The PSM (extract)

a set of guidelines provided by a proprietary J2EE application framework to be used in this system.

The PSM represents the application of a series of domain-specific patterns to the application framework, extending that framework in appropriate ways to support the required functionality. Further refinements to the PSM could then take place to complete the model.


Edit Code Template Binding

Name: custom rollback

User Data:

Api Mode: (● RXE) ○ XDK

Body Kind: Scripted Template

Scripting Language: JavaScript

Body:

```
UnitOfWork currentUow = UnitOfWorkControlle
if(currentUow!=null)
{
    currentUow.setAbortOnly();
}
```

Fig. 10. An example template binding

Finally, code templates were created specific to the application. These transform the PSM into an ISM – in this case an implementation of required functionality that uses the Jakarta Struts coding framework. An example code template is shown in Fig. 10.

This proof of concept was sufficient to demonstrate 3 key principles to the client organization:

1. An MDA solution provided a viable approach to application development for their next generation solution.
2. The separation of architect skills from developer skills would allow the project to optimize resources appropriately.
3. The selected target technologies (runtime platforms and architectural frameworks) were a suitable base for an MDA approach, and transformations from the high level models to extend this platform were viable.

5 Observations

Having considered the principles and practice of MDA, we now assess what lessons this provides for those interested in applying MDA in practice. Adoption of MDA requires a number of changes to traditional software engineering practice. What are the preparatory steps in getting ready for MDA?

Successful introduction of MDA requires that development organizations focus on 4 primary areas: Services, Architecture, Transformations, and Reuse.

5.1 Think services

Many organizations are yet to begin designing enterprise-scale solutions as collections of interacting services. This move toward service-oriented thinking is important because it introduces a number of importance concepts into an organization's way of working. Service-oriented thinking recognizes the need to separate implementation of business functionality from its specification. Services rely on the fact that a specification of what that service offers is available to other services for interrogation and interaction. It frees dependent services from needing to understand how that service realizes the behavior defined in the interface.

Services are related to, but distinct from the notion of components [13]. Two specific areas that distinguish them from components with interfaces are that services have "document-oriented" interfaces. First, services typically process or produce relatively large quantities of data that are best considered as documents, and hence frequently described in XML. This also means that services are relatively course-grained in comparison to typical components. Second, services are relatively loosely-coupled (i.e., they can be implemented in different technologies and can evolve independently) and may also be late-bound. This is quite different, for example, from CORBA objects which are relatively tightly-bound.

5.2 Architect your systems

Models of a system provide value. But only when they are well crafted, describe the system from meaningful viewpoints, are supported by appropriate documentation, and are reasonably complete. A culture that does not value and reward modeling and architectural design will typically not create models that are of value.

Hence, establishing an architecture practice within an organization is critical. Appropriate investments must be made to train architects in the notations and tools of their practice, and the creation of architectural artifacts must be seen as essential to any software-intensive project. Furthermore, we can view the task of creating an enterprise architecture as being one of the highest-value initiatives an organization can undertake, since it considers an architecture at the broadest level. This enterprise view of architecture is particularly valuable from a reuse perspective – a view which of necessity transcends individual applications.[9]

[9] Of course, this architecture has to be grounded in implementation realities. Architects who are divorced from the realities of implementing real systems are at best a distraction and at worst a menace. Just as industrial design has to take account the realities of manufacturing processes and the properties of the materials in use, so software design has to be intimately linked to the problems and costs of manipulating software artifacts.

5.3 Define transformations

Good engineers do not start from scratch every time they are asked to develop a new system. They typically rely on the wealth of previous solutions they have created as the basis for future work. The best engineers formalize the capture and reuse of that knowledge by creating libraries of common design patterns, and understand how those patterns are used to transform between models of a system. This knowledge represents a key asset to the organization, and needs to be nurtured, encouraged, and managed just like any other corporate asset.

5.4 Reuse knowledge effectively

Knowledge is only as good as your ability to search it and apply it in context. To achieve the documentation of patterns and transformation there are a number of approaches possible, ranging from highly formal to rather informal. Tooling is also available to support these approaches. Whichever approach is adopted, organizations need to have well-defined plans for how this knowledge will be managed, used, and evolved.

In applying any asset-based development approach the roles and practices in use in the organization will evolve. How much impact this will have depends on the characteristics and culture of the organization. Guidance on how to adopt and institutionalize an asset management approach is available, and should be consulted [14].

6 Directions and implications

MDA is an important approach to enterprise-scale software development that is already having significant impact in the software industry. Many organizations are now looking at the ideas of MDA as a way to organize and manage their application solutions, tool vendors are explicitly referring to their capabilities in terms of "MDA compliance", and the MDA lexicon of platform specific and platform independent models is now widely referenced in the industry. However, to be successful in practice MDA must be adopted incrementally, and a number of practical barriers to adoption addressed.

6.1 Incremental adoption of MDA

The spirit and intent of MDA is an approach that encourages use of models as the basis for software development. The ultimate vision of MDA is that models are used through the life cycle, with formal transformations between model refinements. In practice, that vision can only be realized by a very small percentage of software development organizations: the visionaries. This recognition does not, however, invalidate the importance and value of MDA. Building models of some aspect of a software domain or software solution is a useful activity; and transforming models, even manually is still in the spirit of MDA.

As a result, incremental adoption of MDA is critical to success. In fact, it is essential to overcome a number of particular problems experienced by software developers, including:

- Many of organizations build business models that are typically realized by multiple business applications (built and owned by different divisions or functional groups), and hence cannot be easily transformed in one shot (because of resource, timing, business, economic reasons). As a result, partial and incremental model transformations must be applied.
- Business process changes typically impact existing systems; and not all existing business processes and applications can readily be represented by easily discernable patterns. The combination of existing systems and lack of patterns often makes it difficult in practice to apply automatic transformations.
- Not all business domains have discernable and well-documented patterns. Hence, until patterns are documented for all domains a large portion of model transformations will have to be applied manually. Eventually, pattern languages for these domains will emerge, facilitating automation.
- Once a more abstract model has been used to produce less abstract models, all the models may evolve at different rates and independently. Not all changes to lower level models will necessitate changes to more abstract models. As a result, there may not necessarily be further forward or reverse transformations taking place.

Consequently, it can be argued that in practice most applications of MDA will not actually be able to benefit from end-to-end automated model transformations, nor require it. In fact, this is an area where current generations of technology offer tremendous value to those wanting to create models of key aspects of their domains.

6.2 Increasing adoption of MDA

There is still much to be investigated to provide the right technical foundation for MDA. Furthermore, the lessons of current MDA practice must be applied to improve MDA successes in the context of the issues of greatest importance to IT. Three areas highlight the ongoing work, and the likely future directions.

First, it has long been recognized that the majority of IT spending is on maintaining and extending existing deployed applications, not on developing new solutions. Hence, it is important for MDA to be positioned in relation to this need. In this regard, web services and service-oriented architectures (SOAs) provide an important backdrop to MDA. Distributed systems are architected to connect services offered through interfaces. The provisioning of the services can take many forms, including through exposing existing functionality as services. As a result, there is currently attention directed at how to model enterprise architectures that integrate existing systems through services. In particular, a key concern is how models of existing systems can be (semi) automatically extracted, and those models included in an MDA approach.

Second, transformations between models are typically realized through patterns and code templates. They offer the mechanism for capturing domain knowledge in reusable forms. There is a great deal of activity in the area of patterns that will be of value to MDA. Notably, a recent trend has been to define patterns at higher levels of abstraction in the development process, and to connect a series of patterns that relate the high level designs into concrete system architectures. This can be seen in the work of the “IBM Patterns for e-business” [12], and the Microsoft Enterprise Solutions Patterns [15]. Both of these essentially offer frameworks for characterizing patterns, and for applying collections of patterns in context.

Third, the core set of base standards necessary for realizing MDA is still under development. While many of the important standards are in place, others still require further development. A prime example is the Query/View/Transformation work currently on-going at OMG [16]. This work will create a common specification for expressing model-to-model transformations. Completing this set of standards will be on-going for some years to come.

Acknowledgements. The work reported in this paper has been carried out by a number of people, and it is my pleasure to acknowledge their contribution. The ideas discussed here reflect the thinking of a broad team at IBM including Ben Amba, Grady Booch, Magnus Christerson, Jim Conallen, Luan Doan-Minh, Pete Eeles, Sridhar Iyengar, Simon Johnston, Grant Larsen, Martin Nally, Jim Rumbaugh, and Bran Selic. The examples reflect the excellent work of Ben Amba, Grant Larsen, Jim Conallen, and Dave Tropeano.

References

1. Barry DK (2003) Web Services and Service Oriented Architectures. Morgan Kaufman
2. Clements P, Northrop L (2001) Software Product Lines: Practices and Patterns. Addison Wesley
3. Greenfield J, Short K (2003) Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Proceedings of OOPSLA
4. Kleppe A, Warmer J, Bast W (2003) MDA Explained: The Model Driven Architecture Practice and Promise. Addison Wesley
5. Frankel D (2003) Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley Press
6. Mellor SJ, Scott K, Uhl A, Weise D (2004) MDA Distilled: Principles of Model Driven Architecture. Addison-Wesley
7. Selic B (2003) The Pragmatics of Model-Driven Development. IEEE Software 20(5), September
8. Booch G, Jacobsen I, Rumbaugh J (1998) The Unified Modeling Language Users Guide. Addison Wesley
9. Ahmed K (2001) Developing Enterprise Java Applications with J2EE and UML. Addison Wesley
10. OMG (2004) `http://www.omg.org`
11. OMG (2003) MDA Guide v1.0.1. Available at `http://www.omg.org/docs/omg/03-06-01.pdf`, 12th June

12. IBM (2004) Patterns for e-business
`http://www.ibm.com/developerworks/patterns`
13. Manes AT (2003) Web Services: A Manager's Guide. Addison Wesley
14. Jacobsen I (1997) Software Reuse: Architecture, Process, and Organization for Business Success. Addison Wesley
15. Microsoft (2004) Patterns and Practices
`http://www.microsoft.com/architecture`
16. Gardner T, Griffin C, Koehler J, Hauser R (2003) A review of OMG MOF 2.0 Query/View/Transformation Submissions and Recommendations Towards the Final Standard. IBM White Paper submitted to the OMG, September 17