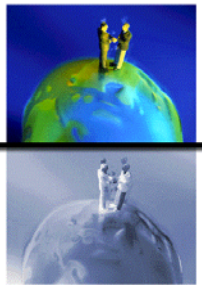

Java, J2EE, and XML Web Services Expertise

Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach

Productivity Analysis

June 2003

Table of Contents

1 Executive Summary

Recently the Object Management Group (OMG) released their Model Driven Architecture (MDA), a new paradigm in server-side development. MDA differs from traditional development in that with MDA you first build your object model using the Unified Modeling Language (UML), then *generate* your code from that UML model using a code generation tool that uses a pattern repository. The OMG believes MDA has many benefits associated with it—perhaps one of the most exciting is increased developer productivity.

The purpose of this case study is to prove or disprove the claims of increased development productivity stemming from MDA-based tools. Two teams developed the same J2EE PetStore application—one team used an MDA-based tool, while the other team used a modified code-centric approach with a traditional enterprise-caliber integrated development environment (IDE).

The result of this study is the MDA team developed their application 35% faster than the traditional team. The MDA team finished in 330 hours, compared to 507.5 hours for the traditional IDE team. As a result of this case study, The Middleware Company is recommending that development shops interested in increasing their productivity evaluate MDA-based development tools for use in their projects.

2 Introduction

By reading this case study, you will learn an overview of the MDA paradigm and see what productivity gains, if any, a team might experience by embracing it. If you are looking for maximum developer productivity within your projects, then this case study is a must-read.

2.1 What is the Model-Driven Architecture (MDA)?

The Model-Driven Architecture is a development paradigm that aims to insulate business and application logic from technology evolution. It helps you build code quickly, in a middleware-independent, well-architected, consistent and maintainable fashion.

The crux of the MDA paradigm is a development process with the following steps:

1. Secure business requirements for an application.

2. Develop UML diagrams for the domain model, independent of any particular technology (J2EE, Microsoft .NET, CORBA, etc.). This UML model represents the core business services and components. It will have elements such as a pricing engine, a shopping cart or an order. This UML model is called a *Platform-Independent Model (PIM)* because it is completely technology-independent—this UML model would be the same regardless of whether you decided to use J2EE or .NET. You develop this UML model using the UML modeling capabilities of an MDA-specific modeling tool.

3. Build UML diagrams for the application, specific to a particular technology (such as J2EE, for example). This UML model will have elements that are technology-specific, such as specific J2EE design patterns. This UML model is called the *Platform-Specific Model (PSM)*. You can build this manually, or you can generate much of it using an MDA tool and hand-tune only the pieces of it that require customization.

4. Finally, *generate* the application code using an MDA tool. That is to say, instead of writing the application by hand based on the UML model, you *generate* the majority of it from the UML diagrams. In the case of J2EE, the MDA tool would generate most of the servlets, JSPs and EJBs. You would then be left to fill in any details that could not be modeled using UML, such as business logic.

2.2 Who is behind MDA?

Object Management Group (OMG) created the Model-Driven Architecture. OMG is an industry standards body represented by several hundred member organizations drawn from both the IT user and vendor communities. OMG is the home of several widely used standards, including Unified Modeling Language (UML) and the Common Object Request Broker Architecture (CORBA). Because it was developed using OMG's open process, MDA is a vendor-neutral approach; any vendor can create an MDA tool that assists with the MDA process. For more information about MDA, please refer to OMG's white papers on the subject, available from http://www.omg.org.

2.3 How is MDA different from traditional development?

We have mentioned several times that MDA lets you generate code from UML models. That is true, but it was also true pre-MDA. Rational Rose, for example, can generate Java classes from a UML model. The key advancement of MDA is that it lets you go from a platform independent, high-level design all the way to platform specific code that is fairly complete. There are several particular points to note:

- MDA starts from a higher level of abstraction than other design processes. The top-level model (PIM) is very abstract; just entities and services.

- The PSM is a complete description of the application in the form of *metadata*. At that level you can enhance the design with technology specific features (e.g. custom finders for EJB entity beans) without touching Java code.

- The code generated from the PSM is close to a complete application. Many tools generate code from some kind of model (such as *Middlegen* or *XDoclet*), but they give you pieces of an application. They are not comprehensive because they do not start from a complete model of the application.

- The algorithms that generate PSM from PIM, and code from PSM, are intended to be configurable by the architect.

2.4 What are the suggested benefits of MDA?

The organizations that have collaborated to produce MDA believe that it will result in the following benefits:

Faster development time. By generating code rather than handwriting each file, you are saving the "busy work" required to write the same files over and over again. For example, in the J2EE world, you sometimes need to write six or more files to create just one EJB component. Most of this can be automated with a clever code generation tool.

Architectural advantages. When using MDA, you model your system using UML—not just by modeling Java classes, but high-level domain entities as well. This procedure forces you to actually *think* about the architecture and object model behind your system, rather than simply diving into coding, which many developers still do. Software engineering principles have proven that by designing your system first,

you'll reduce the possibility of introducing architectural flaws into your system later on in the development life cycle.

Improves code consistency and maintainability. Most organizations have problems keeping their application architectures and application code consistent in their projects. Some developers will use well-accepted design patterns, while others will not. By using an MDA tool to generate your code with a consistent algorithm, rather than writing it by hand, you give all developers the ability to use the same underlying design patterns, since the code is generated in the same way each time. This is a significant advantage from the maintenance perspective. For example, developers at organizations that subscribe to an MDA approach to development will all be able to understand each other's code more easily because they will be leveraging the same design paradigm and language.

Increased portability across middleware vendors. If you need to switch between middleware platforms (for example, switching between J2EE, .NET or CORBA), the Platform-Independent UML Model (PIM) is reusable. From the PIM, one should be able to regenerate the code for the new target platform. While not all of the code can be regenerated automatically, the ability to regenerate a large proportion of one application certainly would save time over having to rewrite it all from scratch. (Note: The Middleware Company believes organizations probably won't take advantage of CORBA, J2EE and .NET neutrality. However, we can see how this may be useful, for example, to independent software vendors (ISVs) selling products that need to support a variety of J2EE application servers so that they can sell their software to any J2EE customer).

In this case study, we focused on evaluating the benefit of *faster development time.* We will not be evaluating some of the other suggested advantages of MDA, such as easier maintenance and improved quality, although we plan to do so in future case studies.

3 Study Description

To evaluate the claim of faster development time using MDA, Compuware Corporation commissioned The Middleware Company to perform a case study where two teams would develop the same application—one team using MDA, the other not. The Middleware Company was chosen to perform this case study primarily because it is vendor-neutral and therefore unbiased in the outcome of the study. The Middleware Company pledges to you that it has conducted itself in a fair and impartial manner in this case study.

In this study, we will not be mentioning the names of the tools being used by team members, although to ensure fair representation of the code-centric approach, we can verify that one of the market's leading IDEs was used. We want this study to be an educational evaluation of the productivity gains that may be obtained from tools that apply the MDA approach, as compared to traditional, code-centric environments. We don't want this study to turn into a "vendor shoot-out."

3.1 About the specification

When performing a case study such as this one, there is the danger that the teams would diverge and build non-comparable applications. To avoid this problem, we wrote a 46-page specification for the J2EE PetStore that described in detail the requirements for the application, from the database to the web user interface. You can download the specification from our web site, http://www.middleware-company.com/casestudy.

The functional specification used for this productivity case study is called *The Middleware Company Application Server Platform Baseline Specification*. This is a specification that was independently created by The Middleware Company over a period of two to three months with the help of a distinguished panel of experts. The panel of experts included book authors, open source contributors, CTOs and VPs of Engineering, representatives from two of the top three IT research firms in the United States, and interested critics of prior case studies conducted by The Middleware Company. The expert group members include:

Assaf Arkin (Chief Architect, Intalio), Clinton Begin (Author, Open Source JPetStore), Rob Castaneda (Author, CEO, CustomWare Asia Pacific), Salil Deshpande (The Middleware Company), William Edwards (The Middleware Company), Marc-Antoine Garrigue (OCTO, lecturer ENSTA), John Goodson (VP, DataDirect), Erik Hatcher (Author, Java Development with Ant), Rod Johnson (Author, Expert 1-on-1: J2EE Design & Development), Anne Thomas Manes (Analyst, CEO, Bowlight), Vince Massol (Author, JUnit in Action), John Meyer (J2EE/.NET Analyst, Giga Information Group, now Forrester Research), Tom Murphy (J2EE/.NET Analyst, META Group), Cameron Purdy (CEO, Tangosol), Roger Sessions (.NET Guru, Founder, ObjectWatch), Vivek Singhal (CTO & VP Engr, Persistence Software), Bruce Tate (Author, Bitter Java), Bruno Vincent (OCTO), Andrew Watson (Vice President & Technical Director, Object Management Group), Wayne Williams (CTO, Embarcadero Software), Joe Zuffoletto (Author, BEA WebLogic Server Bible)

The specification was created to make possible not just productivity case studies, but also several other kinds of studies: architecture and design, performance, interoperability, and more.

3.2 Choice of application

We decided to use the familiar J2EE PetStore application for this comparison. While we acknowledge that the PetStore has its advantages and disadvantages, we chose it primarily because it is a simple application that is familiar to the majority of the community. That familiarity is important to us because we wanted the community to easily understand the case study.

For those of you who may be unfamiliar with the PetStore, it is a simple web-based J2EE e-commerce system. The PetStore has the following functionality:

User management and security. Users can sign into a system and manage their account.

A Product catalog. Users can browse a catalog of pets on the web site (such as birds, fish or reptiles).

Shopping cart functionality. Users can add pets to their shopping cart and manage their shopping cart.

Order functionality. Users can place an order for the contents of their shopping carts.

Web services. Users can query orders via a web service. We extended the PetStore to include this technology, since web services are an emerging area of interest.

From a technology perspective, the PetStore includes the following:

- A thin client HTML UI layer
- JSPs to generate HTML on the server
- JDBC SQL-based data access
- EJB middle tier components
- Ad-hoc database searching
- Database transactions
- Data caching
- User/Web session management

- Web Services
- Forms-based authentication

It should be noted that "PetStore" evokes mixed emotions in some people, because Sun Microsystems never intended the original PetStore sample application to be used as a basis of a case study. After all, PetStore was originally merely a sample application for J2EE, not a fully blown specification. Furthermore, the original PetStore did not represent a well-architected application.

We believe we have addressed these challenges in the following ways:

- We now have a specification for the PetStore, rather than merely an implementation.
- The "modern" PetStore implementations, which conform to our specification, have departed substantially from Sun's original implementation. Practically, what the specification has most in common with Sun's original PetStore is that the application domain involves pets being purchased.
- The specification does not mandate any particular architectural approach to designing the PetStore, giving teams the freedom to architect their applications as they see fit.

3.3 Overview of the rules

The specification describes in detail the rules for building the application. For example, the specification describes what data can be cached, database exclusivity rules, requirements for forms-based authentication and session state rules. It also describes in great detail the required experience of using the application.

We gave each team the same HTML and images. Furthermore, each team was required to use the same database schema. We decided to do this because we wanted to measure the productivity of J2EE coding, not database creation or image generation. We also wanted to make sure that the user interfaces were as close to identical as possible.

We did mandate that the teams use EJB in their code bases. However, beyond this, we did not mandate the details of the implementation of the J2EE code, including choice of J2EE implementation patterns—those decisions were left up to each team. What mattered to us was that the resulting application behaved similarly for each team.

Furthermore, each team was given their choice of using any J2EE-compliant application server. This was also intentional because we wanted each team to use the application server most familiar to them, to maximize their productivity. After all, this is a productivity case study.

We will not be mentioning specific product names of application servers used. Mentioning application server names wouldn't be fair to those vendors, since they were not invited to participate in this case study and assist with the process.

Each team could also choose their own source control tool, as well as miscellaneous J2EE helper tools, such as logging tools or documentation tools. Again, given that this is a productivity study, we felt it most appropriate to give the teams freedom to choose the tools that would make them most productive based on their experience. Both teams chose the following tools:

Tool	Purpose	Comments
StarTeam	Version control	StarTeam is recognized as a leading version control system. We use it internally at The Middleware Company for many purposes.
Apache Axis	Web services	Apache Axis is compliant with Sun's Java API for XML Parsing (JAXP) standard.
Apache Log4J	Logging and auditing	Apache Log4J is quickly becoming a standard logging package for use in J2EE projects.

Note that we did mandate the MDA team to utilize a tool that maintains adherence to MDA. For the traditional development team, we mandated they utilize a leading traditional J2EE EIDE. We did this because the point of the study was to compare different development paradigms, and therefore each team should use the tool appropriate for that paradigm.

3.4 Overview of testing process

Each team performed unit testing prior to submitting their final code bases. Furthermore, to ensure that the final applications behaved similarly, we ran each application through a rigorous testing process. This process comprised 37 testing scenarios that we performed manually on each application. The test scenarios measured whether the applications performed as described in the original specification. The test scenarios are functional in nature, in that they do not perform unit testing of code, but rather describe a process for a tester to interact with the application using the web interface. For example, one of the test scenarios was to sign into the system and edit account information. Both teams' applications passed the tests.

We also tested both teams' code for acyclic dependencies using a package structure analysis tool (available for download at http://javacentral.compuware.com/pasta/). Both teams scored 90% on their acyclic dependency tests, which is excellent.

Finally, we visually inspected the code generated by both teams, since many code-generation tools are known to produce "bad code." Based on our inspection we believe the generated code to be of good quality.

3.5 Overview of the teams

We went to great lengths to ensure that the teams had roughly similar skill-sets, so that neither team would have an inherent advantage over the other. Each team member had significant experience with J2EE development on a variety of application servers. Furthermore, to help factor out differences in skill sets, time was spent orienting the developers on the tools and technologies they would use. The teams also worked out minor team formation issues. This time was not part of our final tally of productivity.

Each team consisted of 3 members:

One senior J2EE architect. This individual had detailed knowledge of the respective development environment. He had responsibility for development, architecture and identifying the appropriate work assignments for completing the specification.

Two experienced J2EE programmers. These individuals each had at least 3 years of experience developing J2EE applications.

3.6 Overview of project schedule and project management approach

To keep an accurate log of the experiences of each team, we held weekly conference calls separately with each team. We took copious notes about the teams' week-by-week experiences in these calls. The teams would answer the following questions:

- What did your team work on this week?
- What was good this week from the productivity perspective?
- What was challenging this week from the productivity perspective?

Summaries of these notes are presented in the next section of this case study.

At the onset of the project, we held a project kickoff event where each team estimated how long it would take them to complete their project. Additionally, each team submitted weekly timesheets detailing their activities from the prior week. These timesheets included estimated time verses actual time spent on tasks.

4 Study Results

In this section, we will discuss the results of the case study. The results are broken up into the following upcoming sections:

In the architectural analysis section you will learn about the architecture and J2EE patterns used by each team. We have created a special *Design Patterns Glossary* section later in this document that explains each pattern briefly. The glossary should be an interesting read for those of you who are aspiring J2EE architects. If you are not familiar with J2EE patterns, you may want to read that glossary before reading the architectural analysis.

In the qualitative results section you will hear each team's qualitative thoughts on the development approach they chose. You'll understand the issues each team encountered, and how they resolved those issues.

In the quantitative results section you will see the final productivity result numbers of each team.

4.1 Architectural analysis

4.1.1 UML and Code Generation

Both teams created UML diagrams for their object models. In fact, they each created very similar object models. They each had abstractions for:

- User Accounts
- User Profile Information
- Products
- Product Categories
- Suppliers
- Shopping Carts
- Orders
- Line Items

The traditional team created UML diagrams for their object model using an open-source tool. They made their UML strictly for design and communication purposes—they did not auto-generate J2EE code from the UML. They did, however, use their IDE's wizards to generate JavaBean accessor/mutator methods, EJB components, struts code, exception handling code and stub-code that sped the implementation of interfaces.

The MDA team created a UML Platform-Independent Model (PIM) as well as Platform-Specific Model (PSM) using their MDA tool. They auto-generated more of their code than the traditional team due to the UML code generation capabilities of the MDA approach. This generated code included JSPs, EJB components, Struts code, exception handling code and interfaces, as well as numerous design patterns and scaffolding code that you need to write when building an application. In addition to the scaffolding code, the MDA tool used the UML object models and generated a working application based on the tool's code-generation patterns.

4.1.2 The Web Tier

Both teams used a combination of servlets, JavaServer Pages (JSPs), and JSP tag libraries (taglibs) in their web tiers. Both teams used *Apache Jakarta Struts* as the framework for navigation within their web tier. Struts is a popular open source framework for building J2EE-based web applications. It encourages use of the well-accepted Model-View-Controller (MVC) design paradigm. Both teams leveraged automated code construction for the MVC.

4.1.3 The EJB Tier

From the architectural perspective, the material differences between the two groups' EJB tier code are the patterns they chose to use.

Pattern	Traditional Team	MDA Team
Session-entity wrapper	Yes	Yes
Primary Key generation in EJB components	Yes	Yes
Business delegate	Yes	Yes
Data Transfer Objects (DTOs)	Yes	Yes
Custom DTOs	Yes	Yes
DTO Factory	Yes	No
Service locator	Yes	No
JDBC for Reading via Data Access Objects (DAOs)	Yes	No
Business interface	Yes	No
Model driven architecture	No	Yes

As you can see from the table above, both teams used a good deal of J2EE design patterns. Although this might be seen as "overkill" for a PetStore application, we wanted to use patterns that a real development shop would use for an actual project, so that we could really put MDA productivity to the test.

Also of note, the traditional team used more patterns than the MDA team. When we first put together this comparison table, this concerned us. Did the traditional team's use of additional patterns invalidate this productivity case study? After all, one might think that the traditional team's code base would naturally take longer to develop given that more patterns were used.

To answer this question, we analyzed each of the patterns used in detail. We also interviewed members of both teams. After conducting this research, our consensus on this matter is that even with the additional patterns used by the traditional team, this comparison is still perfectly valid. To help you understand why this is the case, let's take a look at each pattern the traditional team used and the MDA team did not use:

The service locator pattern did not cause the traditional team extra work. In fact, it saved them time, because by using this pattern, it simplified the code written by clients to the EJB tier including EJB

components calling other EJB components. It reduced the necessary lines of code as it was implemented using a single class with static methods. All EJB access was accomplished using those methods.

The JDBC for Reading via DAOs pattern did take the traditional team time to implement; however, had they not used this pattern, they would have had to write entity beans instead of the data access objects (DAOs). Furthermore, the MDA team had to write some code to sort data manually, which the traditional team achieved through the JDBC for Reading pattern. Therefore this did not cause the traditional team any material increase in work.

The business interface pattern did not cause the traditional team any material amount of work. According to them, it took roughly 10 minutes per interface. Furthermore, the MDA team also had to create business interfaces – just at a higher level, since they had to create the business object model using the MDA-recommended platform-independent model (PIM).

The DTO Factory pattern took the traditional team a small amount of time to implement and test. However, using this pattern actually saved them time since this pattern reduces the amount of code you need to write in your application.

It should also be noted that the traditional team had one of the world's top J2EE patterns experts (Owen Taylor, author of our J2EE Patterns course). This expert knowledge of patterns is not common amongst most IT development organizations. It is therefore understandable that the traditional team used more patterns than the MDA team. Since the patterns chosen by the traditional team did not hamper their time to develop their code base, our conclusion is that there is no evidence in comparing the architectures that would invalidate a productivity comparison.

4.1.4 Security

For the most part, the two applications built by the teams were very similar. However, one interesting area where the two teams diverged was in the area of security.

To address security, the traditional team used an *authentication filter*. This filter checks to see if the currently requested page is restricted and, if it is, checks to see if the user is signed in (sign-in information is stored in the HTTP Session). The team wrote this filter from scratch. The filter implements the `javax.servlet.Filter` interface, supplied by Sun Microsystems.

By way of comparison, the MDA team used their MDA tool to assist with development of the security system. Their MDA tool provided a security framework allowing them to select from a variety of authentication methods, including simple authentication, digest-based, form-based, and programmatic. The team chose to use the programmatic approach. The framework also provided login and logout JSP pages that the team customized for the PetStore requirements.

4.2 Qualitative results

In this section, we'll review qualitative thoughts from both teams, summarized directly from their weekly status updates.

4.2.1 Traditional Team

4.2.1.1 TRADITIONAL TEAM - WEEK 1

The traditional team adopted an iterative prototyping style development process. In the first week, they built a simple prototype that allowed them to architect their design patterns basis for their project. It was a fairly involved process for them to establish a comfortable development environment, and they ran into a few challenges getting StarTeam version control to collaborate effectively with their IDE. They used workarounds to sidestep this issue, which was largely resolved by week 2. They also spent a good amount

of time during week 1 architecting the object model for their system, deciding on a package structure, and other environment-related issues.

From the productivity perspective, they were impressed by the capabilities of their IDE that first week. They found the IDE to integrate well with their chosen application server, as well as have good code generation capabilities. It helped that each of the team members had experience with the IDE in the past.

Their productivity challenges this week related to sharing files, communication issues, performing code reviews, and maintaining a consistent package structure.

4.2.1.2 TRADITIONAL TEAM - WEEK 2

In week two, the traditional development team began to specialize in their various development roles. One team member owned the model (EJB) layer, another team member owned the view (web) layer, while a third team member added value through providing helper classes and interfaces between layers to keep the team members productive.

The team decided to build their system in a use-case fashion, by focusing on each use-case in turn. This week they worked on user account maintenance use-cases, such as sign-in, sign-out, user creation and user preference maintenance.

From the productivity perspective, what they found good this week was leveraging the Struts framework. They found Struts coding to be very efficient. Furthermore, they decided to use a strategy of decoupling their web tier and business tier by allowing "stub" implementation code to be plugged in for code that was not fully built yet. They switched between "stub" and "real" code through properties files and the business delegate design pattern described in the appendix.

The productivity challenges this week were the typical ones a development team first encounters, such as consistent use of source control. They also had some challenges with their IDE, in that if they tried to generate J2EE components from their IDE, and needed to modify them later, the components did not round-trip back into the IDE very easily.

4.2.1.3 TRADITIONAL TEAM - WEEK 3

This week, the traditional team continued their development by working on product catalog browsing, as well as some shopping cart functionality.

From the productivity perspective, what they found good this week was that they had resolved their version control collaboration issues. They also reduced build-time dependencies by building libraries of interfaces that team members could use. At this stage in development, each team member was very productive and the project was well partitioned so that each team member had his own area of development. Furthermore, their IDE had been providing them value in code generation capabilities.

The only challenges this week were clarifications required for the PetStore specification, and some minor refactoring they needed to perform on their shopping cart.

4.2.1.4 TRADITIONAL TEAM - WEEKS 4 AND 5

In their final weeks, the traditional development team wrapped up their application by coding the final use cases, performing testing, and debugging. Also, they integrated their team members' code together, abandoning the stub implementations that had served their purpose.

Productivity gains these weeks included several items. They received a continual benefit from the stub implementation architecture they chose. Very little team communication was required given the interfaces they had authored that allowed them to collaborate efficiently. Also, their web service implementation was built very quickly, as their IDE provided capabilities enabling them to do this—specifically, code generation capabilities in the form of automated WSDL generation and SOAP server deployment.

The challenges this week included some re-factoring and re-analysis of code, such as in their data access layer, which had several flaws. The fact that their application had many layers also caused challenges when things broke, since it was not always clear where the issue was until they delved into the layers. They also had some other minor issues related to Struts and property files.

4.2.2 MDA Team

4.2.2.1 MDA TEAM - WEEK 1

In the first week, the MDA team performed many of the typical steps project teams go through at the onset of their project. They created a detailed design of their object model using their MDA tool's UML modeling capabilities. They set up their source control system, and began to partition the project tasks so that each team member could remain efficient. They also set up a project directory structure and began some initial development.

What was good this week from the productivity perspective was they noticed that they didn't need to build very many components, because the MDA tool was capable of generating the code for them. In fact, they noticed the tool could generate not only the components, but also much of the 'scaffolding' code that links the components together, generating the application end-to-end. The team anticipated 50 to 90 percent of their application could be generated, including web pages, Struts actions, EJB entity beans, EJB session bean facades, J2EE design pattern scaffolding code and database tables.

The key challenge this week was to grasp mentally the new paradigm that the MDA approach represents. The approach was a departure from things they had been doing traditionally, and the tool they used provided many layers and structure that required ramp-up time.

4.2.2.2 MDA TEAM - WEEK 2

In week 2, the MDA team was knee-deep in development. They had completed many parts to their site, including their UML object model, their framework for components, the site navigation system, JSP templates (made in Dreamweaver) that have some code common to all pages, their home page, a good portion of the EJB shopping cart functionality and minor user account management functionality.

What had been good this week from the productivity perspective was that they were able to generate EJB session/entity bean code, web tier code, and a DDL model from their UML object model. It was also very easy to create a test data set. Finally, the end-to-end application framework provided by their MDA tool allowed them to integrate their components together quickly.

The challenges this week were similar to those of the traditional team. They had environment challenges, including getting used to their source control system in a collaborative development environment. They spent quite a bit of time getting their product configured. They also underestimated the learning curve required to embrace their MDA tool, and had to continually learn throughout the week to get used to the paradigm shift from traditional development. Finally, they realized that the code generation capabilities of MDA are sometimes overkill, in that it generated heavier code than they needed. They did want to note, however, that it was possible to modify the algorithms used by the tool to generate code, and in a real project that would be done to alleviate this problem, however this fell outside the scope of this case study.

4.2.2.3 MDA TEAM - WEEK 3

During week 3, the MDA team finished several more use-cases for their system, including shopping cart, order processing, account management, and a sign-in module.

From the productivity perspective, several good things happened this week. Their security system was built very easily since the MDA tool assisted them in this code generation. Their idea of using JSP templates also was beneficial, since their JSPs in essence "inherited" common code from other JSPs, so common code could be updated in a centralized place.

One developer was starting to see the benefits of the MDA process this week, and felt compelled to make the following statement:

The value of MDA is analogous to the value of OO in general. It requires more of you in the design phase, and the payoff comes in the implementation phase. And once you've built one or two apps, you really start to get the benefit from it.

They also had several challenges this week. They ran into some Struts nuances that bogged them down temporarily. They also had a minor issue relating to using session identifiers and cookies with web browsers. Their final challenge this week was MDA-related. They noticed that with the MDA process, you may run into development potholes due to a mismatch between the code that you *thought* would be generated, and the code that was *actually* generated. Their MDA tool sometimes generated code they didn't need, or too much code, which did not help productivity. What the team realized was that their own inexperience with MDA and code generation was the cause of this problem, since they were inexperienced in anticipating what code would be generated. They believe this would be solved with experience and time with using their MDA tool, and the more they used it, the more productive they became. But the short-term unfamiliarity with their MDA tool did cost them a few minor delays.

4.2.2.4 MDA TEAM - WEEK 4

In week 4, the MDA team finished their development, testing and debugging.

From the productivity perspective, their positive gains this week were due to MDA-based code generation, as well as the value of the Struts framework and authentication framework that came with their MDA tool. Also, the integration process they used ran very smoothly.

The challenge this week was that *sometimes* the code that was generated was a liability. They would either have to materially modify it, or abandon it. They did note, however, that the code they wrote manually was patterned after what the MDA tool generated, and that gave them a leg up from the implementation perspective. They also had challenges with source control integration of the MDA tool. The takeaway point they wanted to note is that when you're using a complex product like an MDA tool, you need to understand what's going on and be careful about how you mesh it with version control. First-time users may stumble a couple of times before getting it right.

4.3 Quantitative results

Here is the final tally of development hours spent by each team:

Team	Original Estimated Hours	Actual Number of Hours
Traditional team	499	507.5
MDA team	442	330

As you can see, the MDA team was the clear winner from the productivity perspective in this case study. They came in well ahead of their estimate. Of note, the MDA team did not experience this productivity benefit right away. Their productivity seemed to get better and better as the project duration increased. They attribute this to the fact that an MDA tool does have a ramp-up time associated with it, since it is a new paradigm for most developers. Indeed, one of the programmers expected that the team would have been 10-20% faster were it not for the learning curve they experienced, as this was their first project using the tool.

Also of note, we did not perform formal "bug tracking" in this project; however, we did perform the manual scenario tests described earlier in this document. One interesting point to note was that several bugs were found during the testing process for the traditional IDE team, and none for the MDA tool team. We believe this is due to the consistency of the code generated using the MDA paradigm.

5 Conclusion

Based on the results of this case study, The Middleware Company is impressed by the productivity gains our MDA team experienced using the Model-Driven Architecture. We encourage organizations that wish to improve their developer productivity to evaluate MDA-based development tools for their projects, especially those involving enterprise-class applications and web services. While a short introduction to the MDA approach and tools might be necessary for development teams, the productivity benefits gained from the approach—especially for work on subsequent applications—make the effort significantly worthwhile.

We believe MDA has other values as well. For example, the Platform-Independent Model (PIM) has a very long lifespan. As a development shop, it behooves you to create a business object model such as this that survives technology and lasts for many years. This is an important tool to have in communicating between developers who may be using different tools and technologies.

Another benefit we see to MDA is that it enables the knowledgeable architects within an organization to ensure that the average developer use consistent J2EE design patterns. The architect can achieve this by using the MDA tool to automatically generate pattern code tuned to the architect's wishes. In this manner, the use of patterns becomes a natural part of developers' coding.

One of our team members made the following comment about MDA: "It makes brain surgeons better brain surgeons, but it won't make janitors into brain surgeons." What he means by this is MDA is best utilized with capable, experienced architects at the helm of a project. You still need to have staff members be knowledgeable about J2EE patterns and best practices, object-oriented development, and architectural tradeoffs.

Finally, in terms of code and application quality, a significant observation was that the bugs found using hand-coding methods required remediation and re-testing, which further impacts overall development productivity. The MDA team using automatic code generation via patterns did not require these additional steps to construct their application.

Note that there are other interesting aspects to MDA that we had not evaluated in this case study, such as application performance and maintainability. We did do some basic performance testing to determine that performance between the applications was comparable. However, to obtain the necessary comprehensive and objective results, we feel that this merits further investigation. Also, in the context of maintainability, we think it would be very interesting to see how easy it is to change code in a system built using MDA, compared to a system built using a traditional approach. After all, when you refactor an MDA-based system, you modify the original UML and re-generate code from that UML.

Therefore, to further analyze the differences between handwritten and generated code, we are likely to perform a second study that addresses MDA in the contexts of performance, maintainability and other relevant application aspects. Until then, please share your experiences with us by emailing us at casestudy@middleware-company.com.

Thank you for reading. We hope this case study was useful to you, and we wish you good luck in your development!

6 Appendix: Design Patterns Glossary

In this section, we will explain in more detail each of the design patterns used by the teams. This section is useful for those who may need to brush up on J2EE patterns. It is fairly technical and requires an understanding of J2EE programming. For more information about these patterns, download our book, *EJB Design Patterns*, for free from http://www.theserverside.com/.

6.1.1 Session-Entity Wrapper

When client code (such as JSP code or servlet code) accesses an EJB, it's important to reduce the number of network round trips and transactions to a bare minimum. You can achieve this by wrapping entity beans with a session bean wrapper. The client code calls the session beans, which then delegate to entity beans. Thus the session beans serve as a network façade and a transactional façade on behalf of the entity beans. This is also a natural fit, since a session bean models an action while an entity bean models data. Actions use data.

6.1.2 Primary Key Generation in EJB Components

A primary key is a unique identifier for a database record. Traditionally primary keys have been generated at the database-level by using database features (such as Oracle's sequences). This approach is perfectly valid, however it is not portable across databases. This is problematic for independent software vendors (ISVs) and other groups that need to support a wide variety of databases.

You can generate primary keys in a portable fashion by using EJB components. There are several strategies you can use to achieve this. For example, you can use an algorithmic primary key generator to generate keys in-memory (note that the precision of such primary keys is limited due to limitations in the Java Virtual Machine). You can also create a database table dedicated to creating primary keys, and increment a counter that is cached in-memory via EJB components for performance reasons. These strategies are discussed in more detail in the *EJB Design Patterns* book or our *J2EE Patterns* course.

6.1.3 Business Delegate

One of the challenges with EJB development is coordinating a team to be productive. Often times team members are waiting on each other for code, twiddling their thumbs.

The business delegate design pattern solves this problem. The idea behind business delegates is to create a layer of plain Java classes that hide EJB API complexity by encapsulating code required to call EJB components. The client code (in this case, the web layer) calls the business delegates, which then access EJB components on their behalf. This allows you to "plug in" stub implementations for code that is not yet developed, keeping team members productive.

Business delegates have other minor benefits as well. They effectively shield your client code from EJB completely, allowing you to use another technology at a later date if you see fit. They can also cache data locally, and retry failed transactions in a transparent manner to client code.

6.1.4 Data Transfer Objects (DTOs)

Data Transfer Objects (DTOs), also known as Value Objects (VOs), are classes that hold data that is marshaled across tiers. They are useful for aggregating large amounts of data that would normally take many network roundtrips to retrieve. By using DTOs you can reduce the number of network roundtrips required to your EJB layer. Rather than calling 10 entity bean getter methods, you would retrieve a single DTO that contained all 10 fields. Generally speaking, a DTO maps to an entity bean.

6.1.5 Custom Data Transfer Objects

Custom DTOs are a specialization of the generic DTO design pattern. The best way to understand the need for Custom DTOs is to think about an entity bean that has hundreds of attributes. Most of the time, client code will only want to retrieve a small fraction of this data from the entity bean. Thus it doesn't make sense to populate and marshal a DTO that embodies the complete entity bean. A Custom DTO can represent a subset of an entity bean, thus solving this problem.

6.1.6 DTO Factory

A typical J2EE application will require a plethora of DTOs. DTOs may be needed for each use-case of the system, and DTOs may need change as the domain model changes. A DTO Factory is responsible for creating and consuming DTOs. It can create a DTO and populate it based on a primary key of an entity bean. It can look up an entity bean, make local calls to retrieve data, populate a DTO, and return that DTO to a session bean. This saves from having to rewrite this basic population code over and over again in the session beans. The purpose of the DTO factory is to shield the application from the high velocity of change DTOs experience. It can increase maintainability, reuse, and even performance.

6.1.7 Service Locator

The Service Locator pattern was invented by John Crupi's team at Sun Microsystems who wrote the *J2EE Patterns* book. It is a Java class with static methods that EJB client code uses to obtain references to EJB homes via the Java Naming and Directory Interface (JNDI). The Service Locator pattern enhances performance since it can cache JNDI homes, and reduces the need to write redundant JNDI code in clients. Client code doesn't need to know JNDI, deal with EJB exceptions, or narrow references. Think of the Service Locator pattern as a helper class that contains common JNDI code needed to be written many times.

6.1.8 Business Interface

One of the best-practices established in EJB development is that an enterprise bean class should not implement its own remote interface, since a remote interface contains methods that a bean should not implement, such as those defined in `javax.ejb.EJBObject`. However, the idea of a class implementing an interface is a nifty way to enforce compile-time checking that the method signatures for an implementation match those of an interface, and we would still like to retain this value.

A solution to this is the *business interface* pattern for EJB components. A business interface contains your EJB business methods. The remote interface extends this interface, and the enterprise bean class implements this interface.

6.1.9 JDBC for Reading via Data Access Objects (DAOs)

The JDBC for Reading pattern encourages you to use JDBC to read database data, rather than go through an entity bean layer. There are many reasons for this, such as inflexibility in the EJB query language, the ability to hand-tune SQL, and the granularity constraints that you have when performing SQL with entity beans.

Data Access Objects (DAOs) are lightweight persistent Java classes that are a valid substitute for entity beans. In this project, the traditional team used them as an implementation of the JDBC for Reading pattern, in that they used them purely for reading purposes. They had entity beans as well, for writing purposes, primarily because entity beans were easily generated using the IDE.

DAOs for reading perform well since they can release database connections quickly. Also DAOs allow you to perform order-by operations on queries, which you can't do with entity beans in the current EJB specification's query language. This ordering helps in sorting data to be presented in the user interface.

By way of comparison, the MDA team used entity beans to go directly to the database and didn't use the "JDBC for reading" pattern or DAOs. They sorted data by pushing them into special Java classes called Form Beans and then wrote a routine to sort them. Thus, both teams achieved the same result in different ways.