RESEARCH REPORT

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

Maintainability Analysis

http://www.MiddlewareRESEARCH.com

The Middleware Company
Research Team
January 2004

research@middleware-company.com

1 DISCLOSURES

1.1 Code of Conduct

The Middleware Company offers the world's leading knowledge network for middleware professionals. The Middleware Company operates communities, sells consulting and conducts research. As a research organization, The Middleware Company is dedicated to producing independent intelligence about techniques, technologies, products and practices in the middleware industry. Our goal is to provide practical information to aid technical decision making.

- Our research is credible. We publish only what we believe and can stand behind.
- Our research is honest. To the greatest extent allowable by law we publish the parameters, methodology and artifacts of a research endeavor. Where the research adheres to a specification, we publish that specification. Where the research produces source code, we publish the code for inspection. Where it produces quantitative results, we fully explain how they were produced and calculated.
- Our research is community-based. Where possible, we engage the community and relevant experts for participation, feedback, and validation.

If the research is sponsored, we give the sponsor the opportunity to prevent publication if they deem that publishing the results would harm them. This policy allows us to preserve our research integrity, and simultaneously creates incentives for organizations to sponsor creative experiments as opposed to scenarios they can "win."

This Code of Conduct applies to all research conducted and authored by The Middleware Company, and is reproduced in all our research reports. It does not apply to research products conducted by other organizations that we may publish or mention because we consider them of interest to the community.

1.2 Disclosure

This study was commissioned by Compuware.

The Middleware Company has in the past done other business with Compuware.

Moreover, The Middleware Company is an independently operating but wholly owned subsidiary of VERITAS Software (www.veritas.com, NASDAQ:VRTS). VERITAS and Compuware have a number of business relationships in certain technology areas, and compete directly against each other in other technology areas.

Compuware commissioned The Middleware Company to perform this study on the expectation that we would remain vendor-neutral and therefore unbiased in the outcome. The Middleware Company stands behind the results of this study and pledges its impartiality in conducting this study.

1.3 Why are we doing this study? What is our "agenda"?

We are compelled to answer questions such as this one, due to controversy that sponsored studies occasionally create.

First, what our agenda is *not*: It is not to demonstrate that a particular company, product, technology, or approach is "better" than others.

Simple words such as "better" or "faster" are gross and ultimately useless generalizations. Life, especially when it involves critical enterprise applications, is more complicated. We do our best

to openly discuss the meaning (or lack of meaning) of our results and go to great lengths to point out the several cases in which the result cannot and should not be generalized.

Our agenda is to provide useful, reliable and profitable research and consulting services to our clients and to the community at large.

To help our clients in the future, we believe we need to be experienced in and be proficient in a number of platforms, tools, and technologies. We conduct serious experiments such as this one because they are great learning experiences, and because we feel that every technology consulting firm should conduct some learning experiments to provide their clients with the best value.

If we go one step further and ask technology vendors to sponsor the studies (with both expertise and expenses), if we involve the community and known experts, and if we document and disclose what we're doing, then we can:

- Lower our cost of doing these studies
- Do bigger studies
- Do more studies
- Make sure we don't do anything silly in these studies and reach the wrong conclusions
- Make the studies learning experiences for the entire community (not just us)

1.4 Does "sponsored research" always produce results favorable to the sponsor?

No.

Our arrangement with sponsors is that we will write only what we believe, and only what we can stand behind, but we allow them the option to prevent us from publishing the report if they feel it would result in harmful publicity. We refuse to be influenced by the sponsor in the writing of this report. Sponsorship fees are not contingent upon the results. We make these constraints clear to sponsors up front and urge them to consider the constraints carefully before they commission us to perform a study.

2 TABLE OF CONTENTS

3 EXECUTIVE SUMMARY

In the spring of 2003 The Middleware Company performed and published a report on research testing whether development tools taking a Model-Driven Architecture (MDA) approach increased the productivity of developers building a new J2EE application. That report found that MDA increased productivity 35% over a traditional, code-centric approach.

This study extends the examination to the realm of application maintenance. Two teams performed a set of typical and diverse enhancements to an existing application. One used an MDA-based tool, while the other team used a code-centric approach with a traditional enterprise-caliber integrated development environment (IDE).

The team taking the MDA approach completed the five enhancements 37% faster than the traditional team, in 165 hours versus 260. These results are well in line with those of the first study.

As a result of this study, The Middleware Company reinforces its recommendation that development shops interested in increasing their productivity evaluate MDA-based development tools for use in their projects.

4 INTRODUCTION

This report compares the productivity of two development teams maintaining and upgrading an identical J2EE application. One team used a Model-Driven Architecture (MDA) approach to J2EE development. The other team used a traditional, code-centric development approach. The original application was a version of the familiar J2EE PetStore application, defined by a rigorous functional specification. The upgrades, which represented a range of typical enhancements, were defined by another rigorous specification. Both specs were reviewed by industry experts.

We begin with an overview of MDA, its origins, its specifics and the benefits it promises. Next we describe the study itself: the specification and application used, the nature of the two teams and the terms of the study. Then we examine the results of our study – the structure and quality of the code produced by both teams, the qualitative experience of the developers, and of course the quantitative results. Finally, we drill deeper to see where the differences lie and what explains them.

4.1 What is Model-Driven Architecture?

4.1.1 Model-Driven Development

To understand MDA, it helps to first look at the broader concept of *Model-driven development* (MDD)

A model is simply an abstract representation of some part of an application or system. We may model something as specific as the classes that make up the user interface, something as broad as the distribution of data and functionality across the entire network, or anything in between. And we can build models with any degree of sophistication we choose: from hand-drawn boxes on a whiteboard to complex UML diagrams produced by a modeling tool.

While models have long figured into J2EE development, too often they remain strangely disconnected from the implementation. The model serves as a guide or blueprint, but developers still have to write all the implementation code by hand. As the application evolves, developers often find adherence to the model confining rather than useful. Maintaining the model becomes a chore rather than a help.

MDD is a paradigm that connects the model more closely to the implementation. With MDD, the model not only encapsulates the application design, but is used to generate the implementation code. MDD typically consists of four steps:

1. Create a class model. This includes defining data entities and business operations.
2. Generate Java code from the model. The tool produces code for any J2EE constructs – servlets, JSPs, EJBs, SOAP objects – that you specify.
3. Supply implementation code for the defined operations. Typically the tool knows how to implement CRUD[1] operations. But custom operations such as **placeOrder( )** or **validateCreditCard( )** would require custom code from the developer. (Note that this step may come before step 2, depending on the tool and the modeling approach.)
4. Package the application for deployment to a particular platform.

4.1.2 What is MDA?

Despite generating code from the model, basic model-driven tools may still require developers to manually write a great deal of infrastructure code to complete the application. MDA takes model-driven development two steps further by (1) addressing the high-level structure (architecture) of the application and (2) doing so in a standardized way. In this way MDA tools can speed the development process significantly.

Several characteristics of MDA tools make this possible:

- First, they raise the model's abstraction to a higher level. A class in the top level model simply represents a domain entity, abstracted from how that entity will be used in the application. The entity might be implemented as a business object, a message, a web service, a JSP or all of these, but those choices are made independent of the basic model.
- Second, they generate infrastructure code, including non-Java artifacts such as deployment descriptors. As a result they produce a much higher portion of the application's total code than do basic model-driven tools, allowing the developer to focus on defining the application components.
- Finally, they generate code with an architecture based on best practices. The architect or developer may choose among alternative pre-built architectures, as well as customize them or define new ones.

4.1.3 Where does MDA Originate?

The 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.

1 Create, retrieve, update, delete

4.2 What are the Stated Benefits of MDA?

Model-Driven Architecture claims to offer the following benefits over a traditional, code-centric development approach:

Faster development time. Code generation can save you the "grunt work" required to hand-write the same files over and over again. With the traditional approach, an entity bean, for example, requires 3 or 4 Java classes and one or more XML files. A clever MDA tool can automate most of this.

Architectural advantages. Modeling at the domain level can force you to actually *think* about the architecture and object model behind your system, rather than letting you simply dive into coding (which many developers still do). It's well accepted that greater attention to modeling up front reduces architectural flaws down the line.

Improved code consistency and maintainability. Most organizations have problems keeping code consistent in their projects. Some developers use well-accepted design patterns, while others do not. Using an MDA tool to generate your code with a consistent algorithm, rather than writing it by hand, can force all developers to use the same underlying design patterns, since the code is generated in the same way each time. This can become a huge advantage from the maintenance perspective. Furthermore, developers may be more likely to understand each other's code more easily, given that they're all speaking the same design language.

Increased portability across architectures, middleware vendors and platforms. Models, by definition, abstract to one degree or another from the code they generate. An MDA tool could be configured to produce a body of code with a particular configuration and characteristics. For example:

- Data entities in the model could generate entity beans, JDO classes, or plain old Java objects (POJOs) with JDBC.
- Screen designs in the model could generate JSPs, explicit servlets, or a Swing GUI.
- Deployment settings in the model could generate descriptors for any specified application server, be it WebLogic, WebSphere, JBoss or another.

Additionally, a model could even abstract beyond J2EE itself, giving you the ability to generate J2EE, .NET, or CORBA code from the same model.

This report focuses on evaluating the benefit of faster development time as it pertains to application maintenance. It does not address the other potential advantages of MDA.

4.3 MDA in Detail

MDA defines three layers of models and the relationships among them.

- The *Platform-Independent Model* (PIM) is the highest and most abstract model. It defines the entities and operations of the application domain in a highly abstract way.
- The *Platform-Specific Model* (PSM) is the next level down. This model has the J2EE-specific metadata for the various layers of an application: database, EJB, web and integration. Here, for example, one might define custom finder methods for entity EJBs.
- The *code* model has the actual generated source code: Java classes, JavaServer Pages and deployment descriptors.

The generation of PSM and code model is based on templates or patterns:

- *Technology patterns* translate the PIM into the PSM. This version of OptimalJ exposes the technology patterns to tailoring.

- *Implementation patterns* translate the PSM into the code model; in other words, they generate the source code. This translation will be based on established J2EE design patterns, and the template should be editable.

4.4 OptimalJ and MDA

In this study, the MDA team used Compuware's OptimalJ product (Architecture Edition v3.0), a full-featured implementation of the MDA standard. It presents three layers of models:

- The *domain model*, or PIM. This model breaks down into two areas: *classes* (entities) and *services* (processes).
- The *application model*, or PSM. This model breaks down to the various system tiers: dbms, ejb, web, integration.
- The *code model*.

OptimalJ exposes the translation templates to editing:

- Exposure of the *technology patterns* that translate the domain model into the application model is new to this version of OptimalJ.
- Exposure of the *implementation patterns* that translate the application model into the code model existed in prior versions. But out of the box, OptimalJ's implementation patterns use many established J2EE design patterns, such as model-view-controller (MVC), business façade and data transfer object (DTO).

In addition to the expected code, OptimalJ also generates a default application from the application model. This is a complete web application with links and pages to invoke the CRUD operations for every domain class and the custom operations for every domain service. As we will describe below, the MDA team found this default application very useful for testing purposes.

5 STUDY DESCRIPTION

This study, commissioned by Compuware Corporation, follows on a previous study we performed for them[2]. Both studies measured the productivity benefits of using an MDA approach to J2EE development over a traditional, code-centric approach. While the first study addressed development of a *new* J2EE application, this one focuses on maintaining and enhancing an *existing* application.

As we will explain below, this study follows on the previous one in several respects: It uses the same methodology. Not only do the participants work from a formal application specification, but that specification builds on the one used in the previous round. The application completed in the previous study is the starting application for this one. The IDEs used by the two teams are newer versions of the same two used in the previous study. And there is continuity among the team members from last round to this.

5.1 About the specification

The functional specification used by both teams for this productivity study is *The Middleware Company Maintainability Specification*. It builds upon the specification used in the previous study[3], which described a complete application. The *Maintainability Specification* spells out the requirements for a series of specific enhancements to the baseline application. Those

[2] That study, *Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach -- Productivity Analysis*, can be found at http://www.middlewareresearch.com.

[3] *The Middleware Company Application Server Platform Baseline Specification*

enhancements, described in detail below, cover a range of typical maintenance tasks, from adding user features to augmenting business logic to integrating with legacy systems and web services.

The Middleware Company created the specification with the help of a distinguished panel of experts. The expert group includes:

Rob Castaneda (Author; CEO, CustomWare Asia Pacific), Rod Johnson (Author, *Expert 1-on-1: J2EE Design & Development*), Salil Deshpande (CEO, The Middleware Company) , William Edwards (Practice Director and Senior Enterprise Architect, The Middleware Company), Tom Murphy (J2EE/.NET Analyst, META Group), Cameron Purdy (CEO, Tangosol), Bola Rotibi (Senior Analyst, Ovum), Andrew Watson (Vice President & Technical Director, Object Management Group)

You can download the specification from our web site, http://www.middlewareresearch.com.

5.2 Choice of application

As in the previous round, the basis for the specification used in this study is the well known PetStore application[4], a simple web-based J2EE e-commerce application. The baseline PetStore has the following functionality:

- User management and security. Users can sign into the system and manage their account.
- A Product catalog. Users can brose a catalog of pets on the web site (such as birds, fish or reptiles).

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

- We have a specification for the PetStore, rather than merely an implementation.
- The "modern" PetStore implementations, which conform to our base 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 purchasing pets.
- Neither the original nor the follow-on specification mandates any particular architectural approach, giving teams the freedom to architect their applications as they see fit.
- Shopping cart functionality. Users can add pets to their shopping cart and manage their shopping cart in the usual ways.
- 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, 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

[4] As noted in the past, we recognize that "PetStore" evokes mixed emotions in some, because Sun Microsystems never intended the original PetStore sample application to be used as the basis of a 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-architectured application.

- User/Web session management
- Web Services
- Forms-based authentication

5.3 Specific Enhancements to PetStore

Our Maintainability Specification describes requirements for five major enhancements to the baseline application. Each focuses on a different combination of additions or changes, from new user features to business logic changes to integration with other applications.

The teams in this study implemented all of these enhancements:

1. Tracking pet maturation
2. Calculating shipping costs
3. Soliciting supplier bids for new inventory
4. Integrating with Amazon.com via its web service
5. Integrating with a mainframe application to track shipments

The requirements for each enhancement are summarized in this table and described further in the following sections. For full details, see the *Maintainability specification*.

	Changes to:			
Enhancement	**User Interface**	**Business Logic**	**Database Schema**	**Integration**
(1) Pet Maturation	Significant changes to several pages	Calculate pet age, discount factor, price. Handle individual pet instances in cart.	Extensive: new tables, fields, relationships	None
(2) Shipping Costs	New page showing shipping options; changes to other pages	Calculate cost for each shipping option	1 new table, several new fields	Use web service to get distance
(3) Supplier Bids for New Inventory	None; need simple JMS consumer to monitor messages	Send bid solicitation JMS message when inventory falls	1 new table	Use JMS to solicit and receive bids
(4) Amazon Web Service	New page showing books with links to Amazon	Send query for books on pets to Amazon, get and process results	None	Use web service API to Amazon.com
(5) Shipment Tracking System Integration	New link on home; new page showing tracking info	Connect to CICS app, send tracking query, process results	1 new field	Use mainframe (CICS) app via JCA

5.3.1 Pet Maturation

The baseline specification treats all pets in a specific item grouping (such as Labrador puppy) as identical. A customer could buy any desired number by manually entering the quantity in the shopping cart.

The new specification recognizes that a pet's value changes as it ages. Each individual animal ("pet instance") is displayed, priced and selected separately. Pet pricing is calculated from a base price for the item and a discount based on the animal's age. For each category of pet (dog, cat, etc.) there is a series of "life stages", age categories with associated discount factors. When a pet is purchased, its current age determines what life stage category it belongs to and thus by how much the price is discounted.

This enhancement requires substantial additions to the database schema to account for individual pet instances, manage life stage categories and track pet instances in the shopping cart. It requires additions to the business logic to calculate a pet's age and price discount as the pet gets older.

It also requires changes to the shopping cart portion of the UI, to display and manipulate individual pet instances in a cart line item. For example, if a customer buys four Labrador puppies from the same item grouping with the same base price, they are listed in the cart as one line item with a quantity of four. That line item includes a drop-down list containing the IDs of the individual puppies. The **Remove Item** button for that line item then applies only to the selected puppy, not to the entire line item.

5.3.2 Shipping Costs

The baseline application does not calculate shipping costs as part of a purchase. This enhancement defines a formula for shipping charges based on shipment weight and distance shipped. After completing shopping, the customer sees a table of shipping options with the shipping cost calculated for each. The customer picks the desired option, after which the shipping cost is added to the purchase prices of the pets.

This enhancement requires some changes to the database schema, business logic and user interface. It also requires integration with a web service used to calculate the distance between two zip codes.

5.3.3 Supplier Bids for New Inventory

The baseline specification includes a suppliers table in the database but does not use it in the application. Nor does it require any action when a pet is sold out. This enhancement requires that when inventory of an item falls to a critical level, the application sends to suppliers of that item a JMS message soliciting bids for new inventory.

This enhancement requires changes to the database schema to indicate what suppliers sell what products and to track purchase history. There are no changes to the UI. The bulk of the work involves the business logic and JMS mechanics of sending messages. Note that, for purposes of this study, the specification does not require implementation of logic to receive and process a supplier's bid. For testing purposes the two teams set up a simple JMS consumer to receive and display bid solicitation messages.

5.3.4 Amazon Web Service

This enhancement adds a completely new feature to the original application: When the user finishes shopping and proceeds to checkout, the application displays a selection of books from Amazon.com related to the categories of pets in the shopping cart. Clicking on the link for one of the books takes the user to the main page at Amazon for that book.

This enhancement requires use of Amazon's published web service API to retrieve book information. It does not require any further interaction with the web service beyond a simple book query. Since the application does not store book information, no changes to the database are necessary. Some business logic is required to retrieve a selection[5] of books for each pet category represented in the shopping cart, combine them into a single list and eliminate duplicates. And the spec requires a new web page to display the selected books with links to Amazon's web site.

5.3.5 Shipment Tracking System Integration

This enhancement adds legacy integration to PetStore. It uses a CICS application that receives a shipment tracking number and returns a history for that shipment.

This enhancement requires connection to the mainframe application via the Java Connector Architecture (JCA). The logic to send a query and process the results was the most challenging aspect of this enhancement. Other changes are fairly simple: adding a new link to the home page and creating a new page for the query and results.

Both teams received the JCA adapter and appropriate details about the target application's API, including a COBOL file containing the record structure. However this enhancement presented a special circumstance: OptimalJ has the capability to parse a COBOL file and generate Java stub code, much the way it can generate web service stub code from a WSDL document. The traditional team's IDE has no such wizard. This capability, while powerful, is not inherent to the MDA approach. Therefore, to keep the study's focus on the development approaches rather than on the tools, the traditional team was given the same stub code.

5.4 Overview of the Rules

The rules for this study were much like those of the first MDA study. The *Maintainability Specification* follows the example of the *Baseline Specification* in spelling out rules for enhancing the application. Where appropriate, it 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.

As before, the two teams were required to use the same database schema as spelled out in the spec, although not the same database engine. Each was given DDL for the chosen engine that modified the original schema and loaded sample data. The DDL was organized incrementally by enhancement so that the teams could complete each one independently of those down the road.

Other rules for this study remained the same as in the previous one to factor out extraneous aspects of the comparison and focus it on the productivity of J2EE coding:

- The spec did not mandate the implementation details of the J2EE code, including choice of J2EE implementation patterns – those decisions were left up to each team. What mattered was that the two resulting applications behave similarly.
- Each team was given their choice of J2EE-compliant application server, so that forcing the use of an unfamiliar one would not hinder their productivity. (As before, we refrain from mentioning the specific application servers used.)
- The teams chose their own tools for source control. Both teams used the same full-featured source control system.
- The teams also chose their own tools for web services and logging. Both teams used the same open source tools, namely Apache Axis for web services and Apache Log4J for logging.

[5] It proves impractical to display *all* books for a pet category. For example, a query for books with keyword "CATS" returns over 75,000 titles, including such favorites as *Cat's Cradle*, *Old Possum's Book of Practical Cats* and the ever popular *The Cat in the Hat* (not to mention the lesser known but equally gripping *Cattus Petasatus: The Cat in the Hat in Latin*).

One key difference is that, unlike the last round, this time the teams received no HTML or image files. Most of the UI work consisted of changes to the appearance and functionality of existing JSPs. Any new pages were to be designed according to the spec and created from scratch using the IDE.

5.5 Overview of Testing Process

As in the last study, each team performed unit testing prior to submitting their final code bases. Furthermore, to ensure similar application behavior, both applications were subjected to a set of more than 50 manual tests described in a test specification. 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. Both teams' applications passed the tests.

5.6 Overview of the Teams

The team using the traditional, code-centric approach consisted of two members: one senior J2EE architect from The Middleware Company (who served as team leader and held the same position in the previous study) and one experienced J2EE programmer. Both team members had significant experience with J2EE development on a variety of application servers.

The MDA team consisted of two members, one each provided by Compuware and The Middleware Company (TMC). The Compuware-provided team member is a technical architect experienced with OptimalJ. He served as team leader. The TMC team member is an experienced J2EE developer with a solid foundation in architecture as well as development. Both team members participated on the MDA team in the prior study.

For further continuity with the previous study, the teams used newer versions of the same IDEs used in the previous round: OptimalJ 3.0 for the MDA team; a market-leading, full-featured J2EE development environment for the traditional team.

And, as before, we made sure that the teams had roughly similar skill-sets regarding J2EE development experience generally and the specific development tool they would use. All four team members had previous experience with the IDE they would use. Three of the members, having participated in the prior study, were already familiar with the existing body of PetStore code; the fourth took time to get to know it before starting the study.

5.7 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. During these calls we took copious notes about the teams' experiences. The teams would answer the following questions:

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

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

6 STUDY RESULTS

In this section, we discuss the results of the study. The results are organized into the following sections:

The **Architectural Analysis** section examines the architecture and J2EE patterns used by each team.

The **Qualitative Results** section summarizes each team's qualitative thoughts on the development approach they chose – the issues each team encountered and how they resolved those issues.

The **Quantitative Results** section presents and discusses the productivity result numbers of each team.

Finally, the section **Factors that Affected Productivity** summarizes the reasons for the difference in productivity.

6.1 Architectural analysis

Since the development work performed in this study built on that of the original one, many of the critical design choices were already in place. As the previous report describes in detail, both teams used J2EE design patterns extensively in their original work. On the MDA side, these patterns resulted from the implementation templates that translate the Platform-Specific Model (PSM) into code. The MDA tool automatically generated high-quality pattern code from the model. On the traditional side, the patterns stemmed from deliberate design choices and (for the most part) explicit coding. In fact, the previous study found that the traditional team actually used several more patterns than the MDA team.

Some of the design patterns used by both sides in the original application include:

- Session Façade
- Business Delegate
- Data Transfer Object (DTO)
- Model-View-Controller (MVC)

Additionally, the traditional team used these patterns in its original coding:

- Service Locator
- Data Access Objects (DAO)
- JDBC for Reading

In the current round, both teams expanded their use of patterns. On the MDA side, the newer version of OptimalJ incorporated patterns that the previous study's version did not, particularly Service Locator and DAO. Since these additions resulted from changes to the implementation patterns, they stem from the MDA approach rather than this particular MDA tool. In fact the MDA team in the previous round could have added those patterns had they edited the templates themselves. On the other side, the traditional team used the EJB Command pattern in the legacy integration enhancement. All in all, in this round both teams used approximately the same set of design patterns.

In terms of overall design quality, the code produced this time compared favorably with that of the previous round. For new work that extended previous functionality, the two teams adhered to the same basic designs. The MDA models generated new code that was architecturally equivalent to the existing code. And the traditional team continued their deliberate use of appropriate design patterns.

In terms of work that was substantially new – such as JMS, web service or legacy integration – the MDA approach continued to help keep architectural quality high. Two examples:

- When a web service or JCA integration was added to the model, the model generated not only the basic client stub classes, but also a stateless session bean and business façade wrapper to the stub.
- For writing a JMS producer, the MDA approach made it very easy to define a message data structure and tie it to the message-producing component, thus minimizing the amount of custom business logic actually required.

On the traditional side, the team had to consciously design and build the wrapper code for the various integration components. They did so with a level of quality comparable to that of the original application.

6.2 Qualitative results

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

6.2.1 Traditional Team

6.2.1.1 Traditional Team – Pet Maturation

After initially discussing the changes asked for in the provided specification, the traditional team adopted a pair-programming approach as they began to refactor the existing code base. Pair programming proved beneficial on many occasions as one team member would offer advice or explanation while the other coded. The team spent several hours going over the existing solution and deciding how best to implement the changes for Enhancement One (Pet Maturation). Pair programming helped set a level of understanding of the existing code. It provided a particular benefit with respect to source code control, as there was never a moment when the team's code fell out of sync.

After considering the use of CMP entity beans, the team decided to continue using the Data Access Object (DAO) and JDBC for Reading patterns for displaying information. An initial slowdown occurred while the team worked on the SQL query necessary to load the new item instances. The IDE did not help during this task.

Once they worked out the query, the team spent time chasing down references to **item** objects in the original code. The IDE was effective in quickly finding all of these references and helped clarify the depth and breadth of the impact the Pet Maturation enhancement would make on the existing code.

In addressing the new requirements for displaying individual pet instances, the team investigated using inheritance to solve the problem of loading two similar kinds of data in response to user's requests. This course of action proved faulty; after a day of frustrating work, they abandoned it for what seemed a simpler, faster solution. The team decided to leave existing working code in place and write additional code to accommodate the retrieval and display of the new item-instances.

The team spent a day ripping out some of the previous changes and developing the new code. They adjusted the entity beans to accommodate the new schema and modified the **ShoppingCart** session (as well as the business interfaces it implemented) to accommodate the additional methods now required.

The next morning found the team working on the **OrderManagerSession** and attempting several new deployments and tests of the new methods. They spent the afternoon writing helper methods to manipulate dates, discussing the web-tier code changes and particularly focusing on the **productDetails** page and the changes required there.

The team now spent most of its time making required adjustments to the web tier; they found the chosen IDE helpful when coding the new JSPs and deploying each new version of the web application. Initially, these deployments caused significant delays because the IDE was set to recompile every JSP. This setting was later adjusted, speeding the deployment time considerably.

Day Five started with a morning meeting discussing some of the memory requirements of the IDE and comparing notes regarding what behavior should be expected from it. Much of that day was then spent adjusting some awkward code left over from the original code base and fixing some newly written array manipulation code that was failing during tests. The new code was then tested.

As work progressed into the second week, the team focused on the web tier and struggled with some user-interface aspects of Pet Maturation. Complex manipulation of collections and code changes necessary to display item instances held within them as specified proved a good challenge to the team.

Their IDE was only nominally helpful in resolving the problems that came up. It assisted them when they found they had to back-track and refactor previous attempts and helped by offering a preview of the look and feel of the user-interface they produced using JSPs. There were, however, almost no wizards or shortcuts available that actually sped up the team as they sought to complete these chores. One exception was the ability to save an existing JSP with a new name and then drag and drop it to a new file location from within the IDE.

One task in particular stymied the team: The removal of an individual pet instance from a shopping cart line-item made up of several instances. The spec required the line item to have a drop-down list of instances, from which the user would select the one to remove. The team eventually utilized a bit of JavaScript to resolve it. Their IDE did not help in understanding the syntax of this language, but did offer a tree-view of the various tags in the JSPs, allowing more rapid navigation through a large page. The preview feature was not very useful when solving this problem as, without runtime data, the JavaScript caused errors to appear in the preview. This proved somewhat distracting.

In the second week the team found it harder to synchronize their development efforts. During this week the team worked more independently and, as a result, had to check their contributions in and out more often. This exposed them to further difficulties and a disappointing lack of assistance from the IDE. After several attempts taking more than three hours, they discovered that a file specific to the IDE and its management of EJB files had not been included in the updates to source control. The IDE did not offer a warning, or at any time help to resolve the resulting synch-up errors that slowed the team down. Additionally, twice this week team members found they had to delete their local copy of the project and reload it completely from the server to get back in synch.

The team devoted an entire day to debugging Pet Maturation; at one point a member expressed frustration at having to deal with so many files.

6.2.1.2 Traditional Team – Shipping Costs

By the time the team began the Shipping Costs enhancement, they had become increasingly confident of the synching process and working apart, and so became much more productive.

They created a new project to house this enhancement and, after modifying the data, benefited from the IDE's wizards as they refactored the existing entity beans and added a new one. Losing a connection with source control during this process caused a minor slowdown.

A startup class implemented as a servlet for portability caused some slowdown in productivity as the team faced an unfamiliar security error while attempting to update the status and inventory of the stored items in the database. The team discovered that, due to the need to

access a protected resource (a datasource) , the default initialization of a JNDI InitialContext would not suffice. They eventually resolved the problem through programmatic use of JNDI.

A notable benefit of using the IDE was the ease with which it generated the required web service client using Axis. As a result of this ease, the team also made some small effort to build the stub for the Amazon web service required in Enhancement Four.

While one member moved on to the next enhancements, the other continued debugging and refining Shipping Costs. He spent several hours on entity bean refactoring and dealing with an error in the selected relationships between the **Order** and **ShipmentMethod** beans. While human error was at fault initially, the team faced difficulties wherein their IDE manufactured additional fields and forced a mapping between these in its relationship wizard. This construction of additional fields caused great confusion and frustration as it diverted attention away from the entity relationship to correcting what appeared to be a bug.

The original Shipment JSP was discovered to be missing functionality and an associated struts action. It cost some time to go back and fix the problem.

6.2.1.3 Traditional Team – Supplier Bids

The third week was one of debugging and integration. The team members spent most of their time working independently of one another, which allowed them to work on several tasks in parallel, but also presented large challenges when merging the resultant code.

The team spent much time refactoring what proved to be naïve initial attempts at satisfying the requirements: The JMS client had to be reworked so that it did not eat up 100% of CPU time due to a runaway thread. The class originally responsible for sending the message was refactored and split into a well-defined interface, its implementation, and a helper class.

6.2.1.4 Traditional Team – Amazon Books

Obtaining the pet category for the Amazon web service search proved more work than expected as it was not previously propagated along with the order or selected items. Making it available required modifying several files, DTOs as well as their DAO (DTOFactory) classes.

However, the team used a Java **TreeSet** to good effect to process the results from the Amazon web service query (remove duplicates and sort by two fields as required by the spec).

6.2.1.5 Traditional Team – Legacy Integration

Receiving the stub classes for talking to the CICS application simplified the team's task enormously. Without them, the team would have had to write that code by hand; lacking experience in COBOL programming or CICS connectivity, they undoubtedly would have found this experience arduous, time consuming and frustrating.

One problem hindering productivity had to do with the distribution of classes for deployment. Using a JCA adapter and accompanying RAR file encouraged the use of an EAR file for deployment. This caused several slowdowns as the team discovered that all non-EJB classes needed to be placed in a separate JAR file from the EJB JAR and the WAR file so that they could be safely shared and correctly loaded by the class loader. This took considerable time to detect as crucial and then configure, however once accomplished, the deployment of changes was much swifter (< 20 seconds) as neither the WAR nor EJB JAR files needed to be rebuilt for most re-deployments.

6.2.2 MDA Team

6.2.2.1 MDA Team – Setup

The MDA team began the first week getting set up to work. This included configuring source control properly. OptimalJ generates many different types of files, including model metadata and IDE-specific files as well as the expected Java and JSP source. Setting up version control to handle only those files that must be shared proved to be a critical initial task. Even though they took great care in doing so, the team later encountered synchronization problems. A small, seemingly isolated change to the model affected a number of other files. More than once the team had to either back out a change or do explicit version comparisons (which, fortunately, the source control software made relatively easy) to get back in synch.

6.2.2.2 MDA Team – Pet Maturation

Once properly set up, the team commenced the Pet Maturation enhancement. They began by modifying the domain model to reflect the required changes to the database schema. While most of the model changes were straightforward, the mapping between a shopping cart line item and a pet instance posed an issue. The spec describes a one-to-many (1:N) relationship from line item to pet instance via an intermediate join table. More precisely, it defines a 1:N relationship from line item to the join table, then a one-to-one (1:1) from there to pet instance. To model this situation in OptimalJ's domain model, the team would have to choose between two imperfect alternatives:

- Define a domain class for the join table, then model the two relationships to it exactly as in the database.
- Model a many-to-many (N:M) relationship from line item to pet instance, letting the tool create a table for the join. Then, if necessary, go to the DBMS tier of the application model and point the tool to the join table already in the database.

The team chose the latter alternative for two reasons, one philosophical and one practical. Philosophically, since the join table was present in the database only to facilitate a relationship between two other tables, it does not stand for a real domain entity and therefore should not be represented by a class in the domain model. Practically, this alternative would cause the tool to generate complete code to move between line item and pet instance, reducing the amount of navigation logic the team would have to write.

This experience demonstrated to the team that while many types of entity relationships can be easily modeled, certain complex relationships may require workarounds.

Once the domain (platform-independent) model was updated to reflect the schema changes, the team had to update the application (platform-specific) model, then regenerate the code. In doing so they ran into two obstacles. The first concerned newly created relationships to pre-existing classes; for example, the newly created **PetType** class has a relationship to the preexisting **Product** class. The team found that OptimalJ the was not properly updating the application model to reflect the new relationships. The culprit turned out to be a property of certain application model elements that controls under what circumstances they are regenerated. The property had been previously set to a value that prevented proper updating of the application model. Using OptimalJ's model checking utility, the team discovered the cause of the problem. Changing the property to a less restrictive setting and trying again solved it.

The second obstacle had to do with the new version of OptimalJ. The initial PetStore application was created in an earlier version (2.2), then ported to this version (3.0). The new version exposes the technology patterns for editing; it does so by adding various nodes to the domain and application models. One such node resides under each domain service that has a "domain view", a particular combination domain classes that the service uses. (For example,

the initial PetStore application had an **AccountManager** domain service that used a domain view comprising an **Account** object and its related **SignOn** and **Profile** objects.) When a domain view is affected because the related domain classes have changed, this new "technology node" under the service is used to update the domain view.

The problem was that when porting PetStore to OptimalJ 3.0, these nodes were not created. The model had one service whose domain view had to be updated to reflect changes in domain classes. As a result, for that service the team had to delete and redefine its domain view, then regenerate all associated code. Diagnosing and solving that problem cost several hours.

With the models updated and the code regenerated, the team tested the new class model using OptimalJ's default application. As described above, this is a complete web application that exposes every CRUD or custom operation in the domain model. OptimalJ generates it completely; it requires no coding. The default application proved extremely handy throughout the project. Whenever a team member changed the domain class model in any way or implemented a domain service operation, he could immediately test the change without having to write test code. In the current case, the team tested the new domain classes by browsing, retrieving and updating them in the default application. This guaranteed that they were reaching the new tables in the database and that the plumbing code was working properly end to end.

All in all, despite the complexities of the various layers of modeling, the team felt that the schema changes and related code changes proceeded much more quickly and easily than they would have had direct coding been necessary.

As work progressed on the Pet Maturation piece, other tasks proved easy:

- The spec required that, at application startup, all pet instances with status selected would have their status reset to **ready**. MDA made it very easy to create a special startup component to invoke that logic.
- Handling pet instances by status required a custom finder for the instance entity EJB. It took no time to define the finder in the application model, after which code regeneration automatically updated not only the EJB code but the business façade as well. The new finder could be easily tested in the default application.

6.2.2.3 MDA Team – Shipping Costs

Some of the issues encountered in the first enhancement reappeared here:

- Resetting the regenerate property of certain model elements so that changes to the domain model would propagate correctly down through the application and code models.
- Synchronizing code between team members when model changes were made.

The most significant part of this enhancement proved to be the use of a web service to calculate the distance between two zip codes. The process of creating a web service client in OptimalJ took longer than it otherwise might have because the TMC team member had not done it before and found it initially confusing. On the one hand, the product easily processed the WSDL document into the necessary Java classes. But on the other hand, artifacts of the new web service appeared in two separate places in both the application and code models; optionally it could also have been added to the domain model.

A related problem had to do with the JAR file OptimalJ created for the web service client code. One team member's version of the generated JAR contained all the necessary code, while the other's did not. The discrepancy lay in the build script used to create the JAR; the two members' versions were at odds. After experimenting with recreating the JAR manually, the team eventually got past the obstacle by synchronizing with the correct files; but they had spent several hours looking for the cause of the discrepancy, without success. This incident

highlighted again the hidden complexity of the MDA approach, which most of the time simplified the team's work but occasionally presented confusing problems.

On the other hand, when it came time to use the new web service, the TMC team member found the process easy. He had defined a new domain service to manage shipping costs; one of its operations was a simple wrapper to the web service. Since OptimalJ had already wrapped the web service in a stateless session bean with a corresponding business façade, implementing the new operation was extremely easy. Moreover, since the web service and custom service methods automatically appeared in OptimalJ's default application, testing the new logic was equally simple.

6.2.2.4 MDA Team – Supplier Bids

This enhancement introduced schema changes that created new relationships to existing entities. As a result the team faced the same model update challenge as in the Pet Maturation piece. Fortunately they had learned from the previous experience. Other domain model changes proved very straightforward.

The JMS portion was fairly easy. For message production, the team did the following:

- In the domain model, they defined a new data structure for supplier bid solicitation messages and a domain service to handle message production.
- In the application model, they defined a JMS message and tied it to the data structure, then tied that message definition to the stateless session EJB component that derived from the new domain service.

To create a test JMS consumer, the team defined a message-driven bean (MDB) in the application model, then auto-generated the Java code for it.

In this way, the team had to write virtually no JMS code to implement the spec requirements for this enhancement. In fact, the only manual code necessary was in the **onMessage** method of the MDB to handle message receipt.

6.2.2.5 MDA Team – Amazon Books

Having already once been through the exercise of creating a web service client, the team thought that doing so for the Amazon web service would go smoothly. However, they encountered a temporary obstacle. OptimalJ's model checker reported many errors when processing the Amazon WSDL document. They stemmed from the fact the WSDL uses **locale** and **key** XML element names, which are reserved words in OptimalJ. After some research and experimentation, the discovered that the errors did not prevent them from going forward, so they were simply ignored.

6.2.2.6 MDA Team – Legacy Integration

Connecting to a mainframe application through a JCA adapter proved remarkably easy. The teams were given an adapter for CICS applications and a piece of COBOL code containing the required record structure, as well as basic information about the target application and some rudimentary documentation. After the JCA adapter was installed (a simple process), setting up the JCA client was much like defining a web service client:

- OptimalJ parsed the COBOL file much like a WSDL document to create the correct data structure (a custom DTO). This process also created the appropriated elements in the integration tier of the application model.
- The team had the product create corresponding data structures in the domain and application models.
- Generating the code model (EJB and web code) from the application model was straightforward.

- The default application again proved handy for testing.

Overall, the process was extremely quick and did not require any knowledge of COBOL.

6.3 Quantitative results

To complete all five PetStore enhancements took the MDA team 164.8 hours of development time, compared with 260.5 hours for the traditional team. This comes to a 37% improvement in productivity for the team using MDA.

This result is well in line with those of the previous study, which found a 35% productivity improvement for the MDA approach in building the PetStore application to the original baseline specification. The new result indicates that the MDA approach can offer similar productivity gains with respect to maintaining / enhancing an existing application.

A closer look at the teams' experience with each individual enhancement can shed additional light on the benefits of the MDA approach.

6.3.1 Pet Maturation

Of all the enhancements, this one most closely resembled the original PetStore development experience in terms of its mix of tasks – designing / developing an entity model; writing business logic; creating web pages and the request-handling logic behind them. This enhancement required no integration tasks. Not surprisingly, the results are closely in line with those of the original study.

6.3.2 Shipping Costs

Here the traditional team worked slightly more productively than the MDA team. A closer look at the details reveals some interesting factors:

- In terms of schema changes and new or improved business logic, this enhancement is much simpler than the first one. The traditional team spent half as much time on design and got right to coding, whereas the MDA team modified their model and generated new or modified code. The time savings from code generation over the time spent modeling seems to be proportional to the size and complexity of the application or enhancement. In other words, for a smaller piece of work such as this one, MDA did not pay off as handsomely.
- This enhancement was the first to require use of a web service. The IDEs used by both teams easily generated the basic Java stub code from a WSDL document, so there was little difference on that count. However, the process of adding a web service to the MDA model was more complex than adding one to a traditional project. The TMC member of the MDA team faced a learning curve in this process that cost the team some time.
- Partially offsetting that cost, however, was the fact that the model also generated a wrapper to that web service in the form of a stateless session bean and corresponding business façade. These components slightly simplified the business logic that the MDA team had to write.
- At the same time, the MDA team ran into a mysterious problem with the JAR file generated for the web service. For unexplained reasons, OptimalJ did not properly build the JAR containing the web service classes. The team lost two hours diagnosing the problem and re-synchronizing before getting back on track.

We believe that, barring product-specific problems, the two teams would have completed this enhancement in about the same time.

6.3.3 Supplier Bids

While the results for this enhancement were much like those for the first one (Pet Maturation), the nature of the work was very different. This enhancement had no UI requirements, but added a JMS piece. The MDA team spent longer on design and modeling, but more than made up for it in time savings on integration and business logic.

The MDA team was able to implement the JMS logic with little or no explicit JMS coding.

- For message production, the MDA team modeled a component and a message data structure.
- For message consumption (for testing only, as allowed by the spec), the MDA team modeled and generated a message-driven bean.

The traditional team, by contrast, manually wrote and debugged JMS client code for both production and consumption.

6.3.4 Amazon Web Services

Here, as in the previous enhancement, integration played a key role; in this case, it was the Amazon web service. Again, the MDA team benefited from having friendly wrapper code generated along with the web service stub; they spent less time on integration and business logic than the traditional team. In terms of UI work, the two teams spent about the same time. But because there were no schema changes, the MDA team spent no time on modeling (beyond the web service integration). This savings led to a greater margin than for Enhancements One and Three.

6.3.5 Legacy Integration

The challenge in this enhancement was much like the previous one: very little change to the data structures, some UI work, some business logic, but mostly integration. And the result was a comparably large difference between the two teams.

Here the challenge centered almost exclusively on using JCA to send a query to a CICS application and get a result. The MDA team was able to add the JCA integration to their MDA model in almost exactly the same way as for a web service. This meant they could auto-generate the stub code without having to know anything about COBOL. And, as with the web services, they also auto-generated wrapper components. The entire effort was remarkably easy.

The traditional team, by contrast, did not have the benefit of a wizard to generate their stub code, so to keep the study's focus on the differing approaches the team was given those classes. The remainder of their work centered on installing the JCA adapter and integrating the provided code into the application.

6.4 Factors that Affected Productivity

All in all, what factors accounted for increased (or possibly decreased) productivity from using an MDA approach? To summarize the qualitative and quantitative results presented above, here are some factors that affected productivity generally:

- **Model-generated code.** MDA clearly saved time in letting developers model complex data relationships, then generating all the necessary plumbing code and components according to J2EE patterns. The added time cost of modeling was more than repaid by reduced coding time.
- For example, adding a property to a data structure required updating a DTO and the code that populates it. At one point the traditional team expressed their frustration over having to update so many files to make a change. Somewhat offsetting this frustration,

they also noted that in many cases "having the DAO/DTO mechanism in place saved significant time as complex queries and one-off use-cases were fairly trivial to implement".

- **Greater benefit from larger tasks.** That said, the MDA benefit is greater for more complex modeling tasks. As we noted regarding the Shipping Costs enhancement, for small structural changes the MDA payoff is smaller, perhaps even.
- **The learning curve of MDA modeling.** One small setback to productivity on the MDA side was the complexity of the models and the relationships among their elements. The tasks in this study involved MDA features not used in the first round. For example, we noted how at several places the MDA team had to adjust properties of certain model elements to get the proper results, or how defining a web service created an unexpectedly large number of model artifacts. Whether this complexity is endemic to MDA or specific to OptimalJ is unclear, but more experience with the MDA approach would clearly reduce or eliminate this factor. If the MDA team had to make another, comparable round of enhancements they could do so more quickly.
- **Default application.** The MDA team noted more than once the benefit of using OptimalJ's default application for testing. While this default application may be specific to OptimalJ, it is generated by the standard MDA mechanism. Any MDA tool could produce such an application, given the proper implementation template.
- **Packaging the application.** While the MDA team did have to understand how their tool packaged the resulting application into archive files (WARs and JARs) – and at one point even tried recreating a JAR manually – they never had to adjust that packaging. By contrast the traditional team, at least for the legacy integration enhancement, had to manually readjust the packaging of their application, moving certain classes into a new JAR to solve a deployment problem.

Factors that seemed to be roughly comparable on both sides and thus did not confer any productivity gain:

- **Editing JSPs.** The two teams' IDEs were about equal in terms of editing JSPs. Both teams edited their pages manually and used Struts tag libraries.
- **Source control** and team synchronization proved challenging on both sides. While the MDA approach generated many metadata files that needed to be synchronized, the traditional team had comparable challenges when certain key files were not properly synchronized in source control.

Looking specifically at integration tasks:

- The MDA team benefited from easier integration of web services. The tool not only generated stub code, but convenient wrapper components for using the client stub.
- The MDA team benefited in the same way when it came to legacy integration. The traditional team, even after receiving the stub code, still had to write the wrapper code by hand.
- The MDA team was able to build the JMS piece without writing any explicit JMS code. The traditional team spent time writing and debugging explicit JMS client code.

7 CONCLUSION

Based on the results of this study, The Middleware Company continues to find Model-Driven Architecture a significant and important technology for improving J2EE developer productivity. The results of this study are in line with those of the previous one, indicating that MDA confers productivity benefits to developers maintaining existing applications as well as building new ones.

We again encourage organizations wishing to improve their developer productivity to evaluate MDA-based development tools for their projects. In doing so, we suggest they keep these factors in mind:

- MDA seems particularly well suited for enterprise-class applications. The productivity benefits are greatest with more complex applications and data structures.
- MDA also seems particularly well suited for handling integration technologies, particularly JMS, web services and JCA.
- The learning curve for MDA can be significant. While MDA spares you from writing or editing plumbing code and simplifies many design decisions, it still exposes the details of J2EE technology. More importantly, it adds the intricacies of multiple layers of models. Its power goes hand in hand with complexity.[6] Architects practicing it must learn MDA on top of (rather than instead of) J2EE. On the other hand, to the extent that a particular MDA tool can segregate modeling from coding, thereby solidifying the division of labor between architects and developers, the developers will face a much shallower learning curve.

Please write us at research@middleware-company.com to share your impressions and experiences with us.

[6] To borrow a quote from one member of the MDA team in the first study: "It makes brain surgeons better brain surgeons, but it won't make janitors into brain surgeons."