Model Driven Development for J2EE with IBM Rational Rapid Developer (RRD)

Productivity Analysis

Based on
The Middleware Company Application Server Baseline Specification
http://www.middleware-company.com/casestudy/

The Middleware Company Case Study Team
November 2003

The Middleware Company
RRD Productivity Case Study

casestudy@middleware-company.com

www.middleware-company.com

Table of Contents

TABLE OF CONTENTS 2

1 EXECUTIVE SUMMARY 4

2 INTRODUCTION 4

2.1 WHAT IS MODEL-DRIVEN DEVELOPMENT? 4
2.1.1 What is ARAD? *5*
2.2 WHAT ARE THE SUPPOSED BENEFITS OF MDD? 5
2.3 RATIONAL RAPID DEVELOPER (RRD) AND ARAD 6
2.3.1 Modeling and Code Generation *6*
2.3.2 Using the Class Model *7*
2.3.3 Building the Site Model and Style Models *7*
2.3.4 Creating Web Pages *7*
2.3.5 Adding Application Logic *8*
2.3.6 Setting the Application's Architecture *8*

3 STUDY DESCRIPTION 8

3.1 ABOUT THE SPECIFICATION 9
3.2 CHOICE OF APPLICATION 9
3.3 OVERVIEW OF THE RULES 10
3.4 OVERVIEW OF TESTING PROCESS 11
3.5 OVERVIEW OF THE TEAMS 11
3.6 OVERVIEW OF PROJECT SCHEDULE AND PROJECT MANAGEMENT APPROACH 12

4 STUDY RESULTS 12

4.1 ARCHITECTURAL ANALYSIS 12
4.1.1 UML and Code Generation *12*
4.1.2 Overall Shape of the Code *13*
4.1.2.1 Traditional Team 13
4.1.2.2 RRD Team 13
4.1.3 Security *14*
4.2 QUALITATIVE RESULTS 14
4.2.1 Traditional Team *14*
4.2.1.1 Traditional Team - Week 1 14
4.2.1.2 Traditional Team - Week 2 15
4.2.1.3 Traditional Team - Week 3 15
4.2.1.4 Traditional Team - Weeks 4 and 5 15
4.2.2 RRD Team *16*
4.2.2.1 RRD Team – Early Stages 16
4.2.2.2 RRD Team – Late Stages 16
4.3 QUANTITATIVE RESULTS 17
4.4 FACTORS THAT AFFECTED PRODUCTIVITY 17
4.4.1 WYSIWYG Page Design *17*
4.4.2 Binding Pages to Data *18*
4.4.2.1 Lists in Pages 18
4.4.2.2 Passing Objects Between Pages 18
4.4.3 Default Error Pages *18*
4.4.4 Source Control *19*
4.4.5 Team Organization *19*
4.5 ADDITIONAL TASKS 19
4.5.1 Internationalize a Page *20*

4.5.2 Add and Use a New Table *20*
4.5.3 Partition the Application *20*

5 CONCLUSION 21

6 DISCLOSURES 21

6.1 WHY ARE WE DOING THIS STUDY? WHAT IS OUR "AGENDA"? 22
6.2 DOES A "SPONSORED STUDY" ALWAYS PRODUCE RESULTS FAVORABLE TO THE SPONSOR? 22

1 Executive Summary

With the relentless drive toward ever faster creation of J2EE applications, model-driven development has drawn increasing interest. An emerging segment of model-driven tools is a class of tools known as "architected rapid application development" or ARAD.[1] These tools are associated with higher abstraction, greater productivity over traditional development and less required J2EE knowledge. Among the various products taking this model-driven, ARAD approach is IBM's Rational Rapid Developer (RRD). It promises enormous productivity gains for J2EE developers, slashing the time it takes to build server-side applications

In this case study, we test the RRD productivity claims. Two teams developed an identical application – one using RRD, the other a traditional code-centric IDE.

The result of this study is dramatic: The RRD team developed their application 10.7 times faster than the traditional team. The RRD team finished in 47.5 hours, compared to 507.5 hours for the traditional IDE team. In addition, the RRD team completed some typical enhancements to the application with comparable speed.

As a result of this case study, The Middleware Company recommends that development shops interested in increasing their productivity evaluate RRD for use in their projects.

2 Introduction

This whitepaper compares the productivity of two development teams building an identical application, which was tightly specified by a functional specification reviewed by industry experts. The specification is *The Middleware Company Application Server Platform Baseline Specification*, which is a rigorously specified version of the familiar J2EE PetStore application. One team used IBM's Rational Rapid Developer (RRD), a tool that takes a model-driven approach to J2EE development. The other team used a traditional, code-centric development approach.

We begin with an overview of model-driven development and the benefits it promises. Next we look at how RRD implements model-driven development, focusing on the key features of RRD that differ from a code-centric IDE. 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 explain how so dramatic a difference could occur.

2.1 What is Model-Driven Development?

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.

[1] For a consise, high-level discussion of ARAD, see *Architected RAD Tools Are Delivering Major Benefits,* Gartner Research Note T-19-0792, Jan 29, 2003. This note is available at this URL: http://www.viewpointpartners.com/0pdf/Gartner%20Researc%C9e%20ARAD%20-%20IO.pdf

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.

Model-driven development (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 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.

2.1.1 What is ARAD?

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. Architected rapid application development (ARAD) takes model-driven development a step further by addressing the high-level structure (architecture) of the application. In doing so, ARAD tools speed the development process significantly.

Several characteristics of ARAD 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, the code they generate is "architected" based on best practices. The architect or developer can typically choose among alternative pre-built architectures, as well as customize them or define new ones.

2.2 What are the Supposed Benefits of MDD?

Model-driven development claims to offer the following benefits:

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 MDD 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 MDD 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 MDD 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 whitepaper focuses on evaluating the benefit of faster development time. It does not address the other potential advantages of MDD.

2.3 Rational Rapid Developer (RRD) and ARAD

Rational Rapid Developer is an ARAD tool for creating server-side Java applications. RRD covers the important aspects of J2EE development: the UI, business logic, persistence, messaging, as well as integration with legacy systems and web services.

2.3.1 Modeling and Code Generation

Developing with RRD starts with a *class model*. The class model contains entities with attributes and methods. Classes within the model have relationships to one another. Typically the class model will resemble the database schema, at least initially; in fact RRD can easily generate a class model from a database schema or vice versa.

One interesting aspect of any model-driven approach is how it integrates developer-supplied implementation logic into the model. Most models cannot completely describe an application. For example, when you define an operation in a class model, the tool could generate a Java class with a stub for the corresponding method. But except for standard CRUD operations, the tool would not know how to generate the code for the method body. You would have to supply the method's implementation logic yourself.

RRD handles this issue by including the method code in the class model itself. In other words, when you define a method for a model class, you code the method within the model rather than as part of the generated Java class. RRD stores the method code as metadata in the model, then later incorporates it into the Java classes it generates.

This approach has two important consequences for the developer:

- Methods are coded "in isolation" rather than as part of a complete Java source file. The developer accustomed to scrolling through a source code file to examine other members of the same class may find this approach challenging at first.
- Ordinarily there is no reason to touch the generated Java code. In fact, from RRD's perspective the generated code is "irrelevant". This is not as radical an idea as it may sound; JSP developers rarely look at the Java servlets generated from their JSPs.

2.3.2 Using the Class Model

Once the class model is built, RRD application development centers on building constructs that interact with the classes in the model. RRD can produce many such constructs, among them web pages, messages, or generic components. Each construct has its own *ObjectSpace* (its own subset or view of the class model) that includes only the classes, attributes and methods needed by that construct. For example, to create a web page displaying the details of a single customer and related orders, we would include in the ObjectSpace for that page the Customer and Order classes, but omit the Product and Vendor classes. Drawing an analogy to database schemas: if the class model is the entire schema, the ObjectSpace is a subset of tables and fields made visible to a particular database user.

Extending the analogy, just as a database schema can include calculated fields and virtual tables, so too you can define variables and methods within an ObjectSpace; they belong only to the parent J2EE construct.

The result of this approach is that certain Java classes that derive from the class model are generated for every construct that uses those classes in its ObjectSpace. For example, if you build five different web pages that include the model Customer class in their ObjectSpaces, RRD generates a separate Java Customer class for each page. And each generated Java class includes only those Customer attributes and methods actually used in that ObjectSpace.

This approach has important consequences for the shape of the resulting Java code. We will discuss them further at various points below.

2.3.3 Building the Site Model and Style Models

If the application includes a web front end, you create a *site model*. This is a flowchart-like diagram showing the pages of the site and how a user would navigate among them. You can use this facility to initially define all the pages of the site for completion later.

The site model acts as a storyboard for the site. While it does not generate actual navigation code among pages, it does does the actual page linkages that you have coded.

You also create *style models* for the GUI. These are akin to style sheets for a text document. They contain predefined page layouts and page controls, as well as color and font schemes. Style models insure that pages in the site have a consistent look and feel. They also speed page development by taking care of most formatting choices in advance.

2.3.4 Creating Web Pages

RRD has a WYSIWYG design tool for creating web pages and other screen constructs. You build a page by placing Visual Basic-like controls in the design space, then binding the controls to objects in the ObjectSpace. For example, the data entry fields in a Customer detail page would map to the attributes of the Customer object in the ObjectSpace. RRD generates the code connecting the objects to the page. When the page is loaded at runtime, the fields are automatically populated.

RRD also simplifies the process of linking pages together. For example, if the customer detail page includes a table of the customer's orders, each line in the table could provide a link to an order detail page for that order. RRD would compose a URL for the link that includes the order ID.

RRD's approach to page design frees the developer from two tedious tasks:

- Writing explicit HTML script or JSP logic.
- Writing framework code (e.g. Struts) behind the page.

2.3.5 Adding Application Logic

Once the class and site models have been defined, you will add application logic. This takes the form of custom methods that you define and implement. These methods can belong to model classes, to individual pages or to other constructs.

2.3.6 Setting the Application's Architecture

At some point along the way you set up the application's overall architecture. This process consists of several steps:

- Choosing the deployment technology. You first choose the technology platform – J2EE or Microsoft DNA/COM – then the application server.
- Choosing the *construction patterns*, i.e. high-level patterns that will govern how the code is generated. For example, you can choose to implement persistence via EJBs or POJOs containing JDBC. Similarly you could choose to implement web pages via JSPs or explicit servlets.
- Partitioning the application for deployment. You can package the entire application in a single archive or partition it into multiple archives.

Note that these steps can take place at any point in the development process. Note, too, that you can change these choices without affecting the various models that comprise the application.

3 Study Description

This case study compares the productivity of two different approaches to developing a J2EE application: the model-driven, ARAD approach embodied by IBM's Rational Rapid Developer product and a traditional, code-centric approach. The study centers on two competing teams developing the same Java application: The Middleware Company Application Server Platform Baseline Specification (henceforth, "the specification")

It is important to note that we did not create two new teams for this study. Instead we formed one team to implement the specification using RRD. For comparison we used the results and experiences of a "traditional" team from a previous productivity study conducted earlier this year.[2] That team used a market-leading, full-featured IDE typical of the code-centric approach. We refrain from naming the IDE to focus the study on the productivity of MDD and RRD rather than make this a "vendor shoot out".

[2] *Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach -- Productivity Analysis*, The Middleware Company Case Study Team (June 2003)

Additional details about the setup and conduct of the study:

- Both teams were given files containing the static HTML for the required web pages, so they could incorporate it into the JSPs they created. In this way they could focus on J2EE development and not be sidetracked writing HTML. While the traditional used these HTML files, the RRD team chose to ignore them. Because of RRD's page design features, the team found it more productive to create the pages from scratch, guided by screenshots in the spec.
- After they finished implementing the specification, the RRD team was given a short list of enhancements to implement, "typical" of evolving application requirements. Since these additional requirements were not part of the traditional team's mandate, the RRD team's experience here stands alone as anecdotal evidence, and is not part of a quantitative comparison.

3.1 About the Specification

The functional specification used by both teams for this productivity study is *The Middleware Company Application Server Platform Baseline Specification*. It is a 46-page spec for an enterprise application that describes its requirements in detail, from database schema to web user interface. Using this specification insures that participants in this study would build comparable applications.

The Middleware Company created the specification with the help of a distinguished panel of experts, including 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)

You can download the specification from our web site, http://www.middleware-company.com/casestudy.

3.2 Choice of Application

As is well known by now, PetStore, the basis for the specification, is a simple web-based J2EE e-commerce application with the following functionality:

- **User management and security.** Users can sign into the 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 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
- User/Web session management
- Web Services
- Forms-based authentication

It should be noted 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 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 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 purchasing pets.
- 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. It describes, for example, 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.

The rules for this study are designed to factor out extraneous aspects of the comparison and focus it on the productivity of J2EE coding. In brief:

- Each team was required to use the same database schema, as spelled out in the spec.
- Each team received the same static HTML and images. The traditional team used the provided HTML in constructing their JSPs; the RRD team did not, because RRD made it easier to design the pages from scratch. They did, however, use the provided images.
- 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.)
- Each team chose their own tools for source control, logging and the like. Here are their choices: [

Traditional Team		
Tool	**Purpose**	**Comments**
Unnamed	Version control	A full-featured version control system, used 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.

RRD Team		
Tool	**Purpose**	**Comments**
Unnamed	Version control	RRD integrates well with version control, letting you check files in/out from within RRD.
Apache Axis	Web services	Apache Axis is compliant with Sun's Java API for XML Parsing (JAXP) standard.
RRD's native logger	Logging and auditing	RRD has integrated debugging and logging capabilities.

3.4 Overview of Testing Process

To ensure that the two final applications behaved similarly, we ran both through a rigorous testing process. Described in a 29-page document, this process comprised 37 testing scenarios that we performed manually on each application. The test 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 scenario was to sign into the system and edit account information. We intentionally did not perform code-level unit testing because the two teams' implementations were different, and what really mattered was that the end applications behaved similarly, so that the results of the productivity case study would be useful. That is also why the testing was manual in nature.

Both teams' applications passed the tests.

3.5 Overview of the Teams

The team using the traditional, code-centric approach consisted of three members: one senior J2EE architect from The Middleware Company (who served as team leader) and two experienced J2EE programmers. Each team member had significant experience with J2EE development on a variety of application servers.

The RRD team consisted of two members, one each provided by IBM and The Middleware Company (TMC). The IBM-provided team member is an experienced server-side developer from Technology Solution Partners LLC, a Shelton, CT based services provider that has used RRD for several years, He served as team leader. The TMC team member is an experienced J2EE developer with a solid foundation in architecture as well as development. Before this study he had experience with many development tools, including model-driven, but none with RRD.

Despite a difference in team size, we went to great lengths to ensure that the teams had roughly similar skill-sets regarding J2EE development experience generally and the specific development tool they would use. On the traditional side, the team members came in with experience in the chosen IDE. Still, they spent time reinforcing their knowledge of it and other tools they would use. On the RRD side, to get up to speed, the TMC member received three days of one-on-one RRD training before beginning the project. None of the time spent on training counted toward the total.

3.6 Overview of Project Schedule and Project Management Approach

To keep an accurate log of the experiences of each team, we held periodic conference calls separately with each team. With the traditional team we held these conferences weekly. With the RRD team, because development proceeded so much faster, we did so near the beginning of their development work and again toward the end. We took copious notes about the teams' experiences in these calls. 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 whitepaper.

4 Study Results

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

In the **Architectural Analysis** section you will learn about 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.

In the **Quantitative Results** section you will see the final productivity result numbers of each team.

The section **Factors that Affected Productivity** gives a detailed breakdown of the reasons for the difference in productivity.

Finally, **Additional Tasks** describes the RRD team's experience in performing enhancements to the application.

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 teams differed, however, in how they created these models and diagrams.

The traditional team created UML diagrams for their object model using a product named *Visual Thought*. 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 RRD team used RRD to create a class model from the database schema, then enhanced the model directly in RRD via the UML diagram it displays. They auto-generated much more of their code than the traditional team due to RRD's generation capabilities.

4.1.2 Overall Shape of the Code

The two teams produced bodies of code that are similar in many ways. For example, both applications use JSPs and EJBs. Still, the two bodies of code are different in other, important ways. How the code was produced markedly affected the shape of the result.

4.1.2.1 TRADITIONAL TEAM

For the web tier, the traditional team used a combination of servlets, JavaServer Pages (JSPs), and JSP tag libraries (taglibs). They used Apache Jakarta Struts as the framework for navigation within the 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.)

In the business tier the traditional team used EJBs, both session and entity. Session beans represented business logic while entity beans handled persistence. In addition, the team consciously used many standard J2EE design patterns:

- Session façade
- Primary Key generation in EJB components
- Business delegate
- Business interface
- Data Transfer Objects (DTOs)
- Custom DTOs
- DTO Factory
- Service locator
- JDBC for Reading via Data Access Objects (DAOs)

4.1.2.2 RRD TEAM

The RRD team's approach was different. RRD abstracts from the J2EE platform and from low-level choices about the shape of the code produced. Instead, you first choose the target platform, then make high-level choices about how the code is built. The RRD team chose one platform and set of construction patterns for development, and another for the production version.

For development, the team initially chose Apache Tomcat (which comes with RRD) as the target platform. With this platform choice, RRD produced servlets for presentation and an MVC structure of POJOs for business logic and persistence. The reason for these choices was speed and lighter weight during development.

After finishing development, the team switched to a full-featured J2EE application server. This target platform let them choose JSPs for presentation, stateless session beans for business logic and entity beans for persistence. Regenerating the code with the new settings took only minutes.

Noteworthy is how RRD replicates certain classes for each page. If five pages require the Account class from the model, each page gets its own Account.java, but one tailored to its needs. When you choose to construct EJBs, you get a separate stateless session bean for each page. The argument behind this architecture is that the page-specific classes are lighter in weight because each contains only the logic necessary for the page it serves.

As for design patterns, the RRD-constructed code used some but far fewer than the other team. More to the point, in abstracting from the J2EE platform, RRD took those decisions out of the developer's hands.

4.1.3 Security

The two teams took very different approaches in the area of security.

The traditional team used an *authentication filter*. This filter checks if the currently requested page is restricted and, if so, checks 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.

By way of comparison, the RRD team used a more distributed approach. It too used a session attribute containing login identity to indicate whether login had occurred. But each authenticated page contained its own logic to check the attribute and, if necessary, invoke the login procedure. All security-related logic was written by hand.

4.2 Qualitative results

In this section, we'll review qualitative thoughts from both teams, summarized directly from their periodic 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 their 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 happily 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.

The challenges this week include some re-factoring and re-analysis of code, such as in their data access layer, which had several flaws. The fact that their application has many layers also caused challenges when things break, 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 RRD Team

Again, impressions from the RRD team were taken more frequently because the team progressed so quickly.

4.2.2.1 RRD TEAM – EARLY STAGES

The team had made a very quick start. On Day One the team accomplished an enormous amount of work:

- Loaded the database
- Set up the application environment, including source control
- Created the class model from the database schema. This involved importing the schema to create the classes, then filling in relationships that didn't exist in the DB.
- Set up a style repository to facilitate style consistency while building the application.
- Built the "site model" (the site storyboard) by creating all the empty pages and linking them together.
- Created the login pages and logic.
- Created the shopping cart and logic.

Several things in particular contributed to productivity immediately. First, importing the initial class model from the database schema saved much time. All that remained was to specify class relationships that didn't have corresponding foreign key relationships in the schema. Second, building the site model quickly created all the blank pages and established the overall flow, or user navigation, among pages. Third, the WYSIWYG page designer proved much more efficient than direct JSP editing. And binding page controls to data objects in the class model proved a very efficient way of connecting pages to application logic.

One issue that, from the TMC member's perspective, reduced the productivity gain had to do with the style repository. RRD's GUI style handling is sophisticated and multi-layered. It lets you set up a repository, independent of the application, containing page styles and common controls. These styles and controls then become available when you build PetStore. Additionally, when you place a control in a page you can override its inherited settings in a number of ways.

The TMC member found that default settings didn't always behave as expected. For example, although he gave the default grid control the color scheme shown in the specification, when he used the control in a page it took on different colors. He had to spend additional time reformatting the control.

Additionally, the TMC member was initially concerned about the degree of abstraction RRD inserts between the developer and the J2EE platform. This concern did not hinder his productivity.

4.2.2.2 RRD TEAM – LATE STAGES

As the project progressed, the rapid development of pages continued to prove productive.

Three issues hindered productivity in the final days. The first concerned drop-down lists of states, countries, etc. RRD's native handling of such lists proved problematic, given the specification requirements. These problems forced some awkward programming. See section 4.4.2.1 below for more details.

The second concerned the default error page described in the specification. RRD placed certain obstacles in the way of using JSP's innate error trapping capabilities, forcing the team to explicitly trap application errors and pass them to the error page. This approach was cumbersome compared to how it would be done with pure JSP programming. See section 4.4.3 below for more details.

Finally, the TMC member found it challenging to find the best way to organize code in RRD. In particular, when he wanted to write some global utility functions, it was not immediately obvious where he should put them. This problem would likely disappear with experience in RRD.

4.3 Quantitative Results

Here is the final tally of development hours spent by each team to develop PetStore to spec:

Team	**Original Estimated Hours**	**Actual Number of Hours**
Traditional team	499	507.5
RRD team	40-50	47.5

As you can see, the RRD team was the clear winner from the productivity perspective in this case study. They built the PetStore application dramatically faster than the traditional team, in less than 10% of the time.

4.4 Factors that Affected Productivity

Clearly RRD's ability to generate application code from a model accounts for most, if not all, of the productivity gain. But with such a dramatic improvement it is worth digging deeper and examining the differences in detail. There were a number of significant factors contributing to RRD's greater productivity, as well as a few that detracted. While we can't quantify each one individually, we can indicate the approximate importance of each to the outcome.

4.4.1 WYSIWYG Page Design

One huge factor in RRD's favor is that it completely shielded the developers from the need to edit servlets or JSPs. Again, the traditional team used the supplied HTML. And while their IDE offered a JSP editor, they were still forced to edit JSP syntax directly. They had to convert the supplied HTML into JSPs. This can be extremely tedious, time-consuming work, even with templates and custom tag libraries. RRD's page designer, linking Visual Basic-like controls to model classes, proved a much cleaner and more productive way to build pages.

A smaller, related gain came in debugging pages. In traditional JSP development, to test the correctness of your JSP syntax you must run the application and invoke the JSP. In RRD you can choose to translate your page designs directly to servlets rather than to JSPs. When you do, the build process compiles the servlet and thus detects JSP syntax errors before you even run the application. This process is much faster. The RRD team chose this approach.

Two additional notes on RRD & page construction:

- RRD still requires or allows you to use JSP logic in specific places. For example, the text you place in a label control may include JSP expressions as well as literal strings. Still, these uses are isolated and constituted a tiny fraction of the team's time spent on page construction.
- RRD lets you view and edit the HTML/JSP logic it produces from your page design. The RRD team found a couple of places where viewing or even overriding the HTML was necessary to achieve a desired result.

4.4.2 Binding Pages to Data

RRD's paradigm of binding page controls to objects in the class model via the ObjectSpace also contributed enormously to productivity. The RRD team did not have to program to Struts or any similar low-level framework. For the typical data page, it sufficed to place labels or text fields in the page and map them to attributes of some object in the ObjectSpace. In special cases a page or an individual control required some explicit pre-load logic; RRD provides places for that logic.

There were two factors that reduced this productivity gain somewhat.

4.4.2.1 LISTS IN PAGES

The first had to do with using a list (such as a drop-down list of US states) in a page. RRD maps the user's selection from the list to an integer, even if the data to be stored is a string. In the list of US states, for example, when the user chooses Alabama, RRD considers the "value" of the list to be 1 rather than "Alabama". To bind the list to a string attribute of an Account object in the ObjectSpace required some additional programming.

Note: If the data to populate lists resides in database tables, this problem goes away. But the specification's schema did not include such tables, and the RRD team refrained from creating them because the spec barred changes to the schema.

4.4.2.2 PASSING OBJECTS BETWEEN PAGES

A second complicating factor had to do with passing data objects from one page to another via the HTTP session. Ordinarily the source page simply places the object in the session as a named attribute, and the target page retrieves the same object in similar fashion. In RRD, however, a model class translates to a separate Java class for each page that uses it. For example, if the model has a ShoppingCart class and three pages use it in their ObjectSpaces, each page gets its own generated ShoppingCart.java. The three generated classes are different because each lives in a page-specific Java package. This means the pages cannot pass actual ShoppingCart objects via the HTTP session.

RRD's preferred solution is to convert the objects to/from XML and pass the XML strings via the session. RRD does make it relatively easy to create a DTD from a model class and convert between object and XML. Nevertheless, this process does require additional programming in pages that use it. The RRD team used this technique.

4.4.3 Default Error Pages

The specification requires all application exceptions to be captured in a default error page. In the screenshot in the specification document, the error page includes the stack trace for the exception, although the spec leaves the actual content of the page to the implementer's discretion.

The JSP API makes it easy to implement the error page shown in the spec:

1. Use a page directive in each source page to point to the error page.
2. Use a page directive in the error page to mark it as such.
3. In the error page, refer to the exception via the predefined variable **exception**.

RRD, in abstracting page design from the JSP API, obscures and complicates this process, which cost the RRD team time. RRD makes it easy to designate an error page for a given source page. What's difficult is capturing and passing the exception. The team was forced to wrap every method body in a try block with a corresponding catch that place the captured exception in the session. The default error page then must retrieve the exception and display it.

4.4.4 Source Control

The use of source control probably gave the RRD team a modest advantage. In the case of the traditional team, the team leader knew the chosen product well and trained and guided the other team members in its use, but the product did not integrate with the IDE. Consequently, source control procedures always required team members to take deliberate actions outside the IDE. Also, the team had to figure out which files generated by the IDE to keep in the source control tool and which to ignore.

The RRD team chose one of several popular source control software tools that are supported in RRD. Apart from initially logging into the source control server, the team members could do all source control actions easily from within RRD. Moreover, RRD automatically knew which files to place in source control.

Offsetting this advantage to a small degree was the issue of how RRD organizes its artifacts. While the RRD team leader understood this organization very well, the RRD novice on the team faced a learning curve. Where this mattered was in choosing what items to check out for a given development task. For example, class and method definitions for the entire class model are stored in a single file, while the body of each method is stored in its own individual file. This meant, paradoxically, that two team members could simultaneously edit different methods of the same class in the model, but could not simultaneously define new methods of different classes in the model. The TMC team member had to adjust to this unusual (for him) organization of source files.

4.4.5 Team Organization

Finally, the RRD team may have gained some productivity simply by having a smaller team. Fewer team members meant less difficulty dividing up the work, fewer source control conflicts and fewer miscommunications. It also meant that tasks performed together (such as design discussions) took fewer total man-hours. The TMC team member felt that, with a moderately-sized application such as this, having a third team member with comparable skills to his own might have actually slowed development rather than accelerated it.

4.5 Additional Tasks

After completing the development of the application to specification, the RRD team received a list of enhancement / maintenance tasks to perform. These tasks were intended to represent typical follow-on tasks developers might be asked to perform on existing applications.

As noted earlier, since the traditional team did not perform these tasks, we cannot compare the two teams' experience to measure productivity gains. Still, the RRD team's experience is interesting on its own.

Here are the additional tasks performed:

- **Internationalize the account edit page.** Set up a framework for multiple locales / languages. Translate one page into a foreign language based on the choice of locale.
- **Add a new table to the database and use it.** Add a table of customer reviews that has a N:1 relationship with items. Update the class model to match the database schema. Display a list of reviews on the item detail page.
- **Partition the application.** Divide the application into separate deployment archives for presentation (WAR) and business logic (JAR).

The time spent completing these tasks was short:

Task	**Time Spent**
Internationalize the account edit page	1.1 hours
Add a new table to the database and use it	1.5 hours
Partition the application	0.3 hours

Details on these tasks follow.

4.5.1 Internationalize a Page

RRD's i18n capabilities are sophisticated. You can import locale-specific resource bundles from Excel, a database or elsewhere. Individual label controls and messages on pages are keyed to specific resources in the chosen bundle. The process is fairly straightforward; the most tedious part is translating the various pieces of text into foreign languages.

The specification added a wrinkle that complicated the task slightly: The user can choose a preferred language. So the locale must be chosen programmatically when the user logs in.

Still, it only took the RRD team a bit over one hour to generally set up internationalization for the application, translate the messages for one page into Spanish, apply the resources to the page and modify the login code to choose a locale.

4.5.2 Add and Use a New Table

The team was given SQL to create and populate a reviews table with a 1:N relationship between items and reviews. The team executed the SQL, imported the new table into the class model, set up the relationship to the items table, then modified the item detail page to list the reviews in a separate grid below the other item data. The team implemented the reviews grid to behave like others in the application, using Previous and Next buttons to handle displays beyond a certain length.

The whole process took less than 1.5 hours.

4.5.3 Partition the Application

RRD includes a "Partition Architect" that lets you partition the app. It presents a list of all constructed artifacts, along with separate lists for the different app tiers (presentation, business logic, persistence). You move individual artifacts from the first list into one of the others. You then tell RRD to build the requisite archive files. The process is comparatively simple and allows for creation of multiple partition models, with different deployment technologies and different architectures, e.g. development partition deploys locally to Tomcat as one file , production partition.deploys to WebSphere and has separate files for presentation and business logic/persistence.

The team partitioned the application into two parts: presentation and business/persistence. It took 20 minutes.

5 Conclusion

Based on the results of this case study, the productivity gains experienced using Rational Rapid Developer are impossible to ignore. The Middleware Company is impressed by the dramatic difference over the traditional approach.

Organizations wishing to improve their developer productivity should evaluate RRD for their projects. In doing so, we suggest considering these factors:

- **RRD's strong emphasis on modeling and therefore on architecture.** We at The Middleware Company consider modeling and architecture good things, but they do require a change in thinking for traditional, code-centric developers. Additionally one might consider how RRD and its models would interact with other model-based tools already in use. RRD provides class model import and synchronization with IBM's UML modeling tools Rational Rose and Rational XDE, as well as XMI import.

- **RRD's abstraction from the J2EE platform.** Again, RRD shifts your development activities almost completely away from infrastructure code and towards modeling and business logic. Certain low-level platform issues such as package structure and use of code patterns are taken from the developer's hands. This may be a challenge to experienced J2EE developers, just as some C++ programmers find the absence of pointers and direct memory management in Java a challenge. However, it is the primary reason why developers less experienced in J2EE can quickly become productive in RRD without understanding the platform complexities.

- **The developer's background.** The TMC member of the RRD team felt that RRD is particularly well suited for developers just coming to J2EE from elsewhere, that it hides the complexities of J2EE development much like PowerBuilder and Visual Basic hide the complexities of OO development.

- **RRD's ability to handle existing applications.** In this study the two teams built a new application from scratch. They did not evaluate the ease of pulling an existing application into the tool. Shops that have more maintenance than new development should do so when considering RRD. Additionally, RRD claims broad legacy integration functionality, although these capabilities were outside the scope of this case study.

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

6 Disclosures

This study was commissioned by IBM.

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

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

IBM 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.

6.1 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 case 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)

6.2 Does a "sponsored study" 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 study if they feel it would be 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.