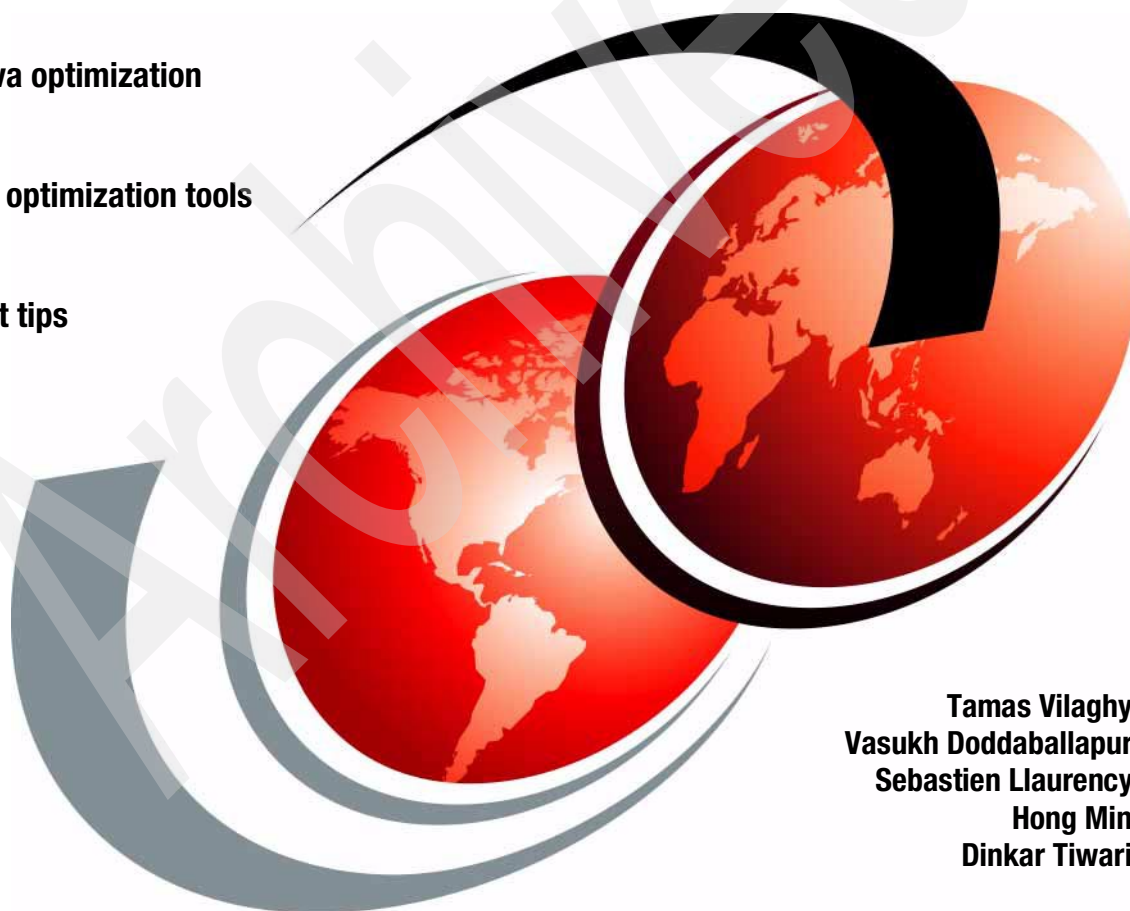


Writing Optimized Java Applications for z/OS

Tips for Java optimization

How to use optimization tools

Deployment tips



Tamas Vilaghy
Vasukh Doddaballapur
Sebastien Llaurency
Hong Min
Dinkar Tiwari



International Technical Support Organization

Writing Optimized Java Applications for z/OS

October 2002

Archived

Take Note! Before using this information and the product it supports, be sure to read the general information in “Notices” on page v.

First Edition (October 2002)

This edition applies to WebSphere for z/OS Version 4.0.1 for use with the z/OS Operating System Version 1.1 and above.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. HYJ Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 2002. All rights reserved.

Note to U.S Government Users – Documentation related to restricted rights – Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	v
Trademarks	vi
Preface	vii
The team that wrote this redbook	vii
Notice	ix
Comments welcome	ix
Chapter 1. Introduction	1
1.1 Overview	2
1.2 Performance metrics	3
1.3 Environment	3
Chapter 2. The performance of an architecture	5
2.1 Overview	6
2.2 J2EE architecture	8
2.2.1 Model View Controller architecture	9
2.2.2 J2EE components and performance	11
2.2.3 World of patterns	17
2.3 Number of tiers	19
2.3.1 Logical tiers	19
2.3.2 Physical tiers	20
2.4 Scalability	25
Chapter 3. Tools	29
3.1 Performance monitoring	30
3.2 Introduction to tools	32
3.3 Introscope	34
3.4 JInsight	44
3.5 JProbe	49
Chapter 4. Programming techniques	57
4.1 Applying design patterns	58
4.1.1 Creational patterns	58
4.1.2 Structural patterns	62
4.1.3 Behavioral patterns	66
4.2 Code optimization techniques	69
4.2.1 Java programming techniques	69
4.2.2 JDBC and SQLJ	81

4.2.3 Servlet and JSP performance hints and tips	103
4.2.4 EJB programming hints and tips	112
Chapter 5. Data access	117
5.1 The sample application: OnlineBuying	118
5.2 Cache data to reduce remote calls	119
5.3 CMP versus BMP	123
5.4 BMP with SQLJ	124
5.5 Stateful vs. stateless session EJBs	128
Chapter 6. Deployment and execution	133
6.1 Isolation levels	134
6.2 Read-only methods	137
6.3 Transaction attribute	139
6.4 Load at server startup	142
6.5 JNDI lookup	143
Related publications	147
IBM Redbooks	147
Other resources	147
Referenced Web sites	147
How to get IBM Redbooks	148
IBM Redbooks collections	148
Index	149

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

CICS®
DB2®
DRDA®
IBM®
IBM eServer™
IMS™

MVS™
NetVista™
OS/390®
Perform™
Redbooks (logo)™ 
S/390®

SP™
Tivoli®
VisualAge®
WebSphere®
z/OS™
zSeries™

The following terms are trademarks of other companies:

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.

Preface

This IBM Redbook is designed to help you write optimized Java applications for z/OS. It addresses different aspects of performance, and discusses architectures, design patterns, coding techniques, deployment, and data access issues. It applies to WebSphere for z/OS Version 4.0.1 for use with the z/OS Operating System Version 1.1 and above.

Using tips on writing Java-based solutions that have been collected from many different sources, we show how applying many of these tips and measuring performance can help you tune your Java application to perform better on z/OS.

We describe how to use the optimization tools Introscope, JInsight, and JProbe, and provide recommendations regarding application architecture and Java coding practices. Ultimately, we illustrate how deploying the application on the z/OS WebSphere platform will provide enhanced workload management and high availability, as well as improved scalability and security.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Poughkeepsie Center.

Tamas Vilaghy is a project leader at the International Technical Support Organization, Poughkeepsie Center. He leads redbook projects dealing with e-business on zSeries servers. Before joining the ITSO, he worked in System Sales Unit and Global Services departments of IBM Hungary. Tamas spent two years in Poughkeepsie, from 1998 to 2000, dealing with zSeries marketing and competitive analysis.



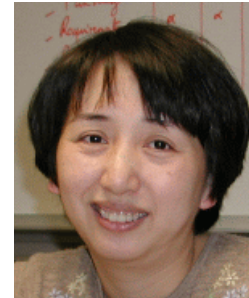
Vasukh Doddaballapur is a Java developer and applications analyst at PentaFour, USA. He has five years of experience in e-business and OLTP application development. He holds a MS degree in management from Bombay University, India. His areas of expertise include Java, WebSphere, OS/390, DB2 and CICS.



Sebastien Llaurency is a Java developer and WebSphere specialist in France at the Design Center for e-Transaction Processing. He has two years of experience in prototype development for EMEA customers. His areas of expertise include Java development tools and WebSphere Application Server.



Hong Min is a software engineer at the IBM Design Center for e-Transaction Processing, Poughkeepsie, USA. She has four years of experience in S/390 e-business technical support, customer application design, and prototyping. She holds an MS degree in Computer Science from Drexel University, Philadelphia, PA. Her areas of expertise include OS/390, Java and WebSphere.



Dinkar Tiwari is an e-business IT Consultant in USA. He has five years of experience in application design and development and systems and storage networking sales/marketing. He holds a BS degree in Computer Science from Northern Illinois University, DeKalb, IL. His areas of expertise include C/C++, Java and WebSphere.



Thanks to the following people for their contributions to this project:

Rich Conway
International Technical Support Organization, Poughkeepsie Center

Bob St. John, Mary Ellen Kerr, David Cohen, Ivan Joslin
IBM WebSphere development, Poughkeepsie

Ravi Kalidindi
Owner of the PreciseJava Web site, with lots of useful additions to Chapter 4

Notice

This publication is intended to help application developers who code applications to be deployed on WebSphere for z/OS. The information in this publication is not intended as the specification of any programming interfaces that are provided by WebSphere for z/OS. See the PUBLICATIONS section of the IBM Programming Announcement for WebSphere for z/OS for more information about what publications are considered to be product documentation.

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- ▶ Use the online **Contact us** review redbook form found at:
ibm.com/redbooks
- ▶ Send your comments in an Internet note to:
redbook@us.ibm.com
- ▶ Mail your comments to the address on page ii.

Archived

Introduction

This chapter gives an introduction to the performance work we did, why performance is important, and how it relates to other aspects of development. It also describes the performance metrics and the environment we used.

There are important caveats to performance measurements and evaluations. First, a performance evaluation should not rely on a single performance data element. There are many aspects to performance. There are also many aspects to the system being evaluated, like the functional requirements of a particular application and non-functional requirements such as security, availability, etc. So when you evaluate a performance measurement, make sure you take these aspects into consideration.

Second, performance data is always a point-in-time snapshot. A system or an application always evolves, and this is especially true with WebSphere on z/OS. Users demand security, functionality, availability and performance, so WebSphere on z/OS changes continuously and you always have to apply the latest service levels to get the most out of it.

In this book we gather some useful hints regarding application performance and also report on some measurements. The latter are not official, not done in a controlled environment—they are just measurements the project team made to verify some thoughts we had.

1.1 Overview

What is application performance? What is the effect of poorly performing applications? How do we measure it? And once we measure it, how can we write applications that perform well?

Application performance depends on both the system and the application architecture and design, and each affects the other in the following manner:

- ▶ An application's architecture and physical implementation affects the consumption of a system's resources.
- ▶ A system's architecture and physical implementation affects the runtime of an application.

Typically, system specialists focus on new technology to update systems as necessary, and thus lose familiarity with applications. Similarly, application developers focus on understanding and implementing business processes and requirements, and thus may fail to understand the systems their applications run on. Both areas influence performance without having an explicit focus on it.

As a result, when performance problems surface, no one accepts responsibility because no one knows where the problem lies. Often, the fixing of performance problems is a reactive process and adds unanticipated costs. In fact, the only solution often is to upgrade the systems to faster and bigger machines.

Applications developed with no regard to performance result in unnecessary processing. This leads to expensive upgrades due to high CPU utilization, poor response times, and high processing times. Improving these factors has become a fundamental requirement for businesses, especially for Web applications. Tuning of applications has become a highly sought-after skill. Companies either establish teams comprised of performance specialists or pull together people from different teams as needed to focus on performance-related issues.

Knowing the effects of poorly performing applications allows us to place much needed emphasis on the design and implementation of applications. However, we still need to learn how to design and code an application to get the best end-to-end performance possible.

These are the issues we discuss in this redbook. We focus on the application architecture, design, coding, and deployment. For system tuning, refer to *z/OS for e-business: Introduction to System Tuning*, SG24-6542.

This redbook is not intended as an exhaustive work on performance, because we did not create a fully controlled performance measurement environment—we just made some basic measurements and give recommendations based on these.

Another aspect of all performance material is time. The residency related to this project ran in December of 2001. We ran the then current version of WebSphere for z/OS. Since then many PTFs have been issued which deal with performance problems. For the latest PTFs available, check the WebSphere for z/OS support page at:

http://www-3.ibm.com/software/webservers/appserv/zos_os390/support.html

We also gathered some recommendations related to Java coding. The Internet has many recommendations on how to write better Java code. We believe that this redbook will help you get faster access to this information and also help bring together the latest Java coding practices with the traditional strengths of z/OS, such as scalability and security.

1.2 Performance metrics

In this redbook, we make recommendations on a variety of topics including application architecture, programming practices, and deployment. To do so, we ran some tests with Java code and measured the following performance-related metrics:

- CPU utilization
- Throughput (requests per second)
- Response time
- Memory usage

1.3 Environment

Our testing environment included the following hardware and software.

► Development

- IBM NetVista Workstation
 - Pentium III 864MHz
 - 512MB RAM
- Microsoft Windows 2000 Professional operating system
- VisualAge for Java Enterprise Edition v4.0
- WebSphere Studio v4.0

Deployment

- IBM eserver zSeries 900
- z/OS 1.1 operating system

- IBM Java Developer Kit for OS/390 v1.3.0
- WebSphere Application Server for z/OS and OS/390 Application Assembly Tool v4.00.023
- WebSphere Application Server for z/OS and OS/390 Administration v4.0.1.006, also known as System Management End User Interface (SMEUI)
- WebSphere Application Server for z/OS and OS/390 v4.0.1 PTF level 14
- DB2 v7.1
- JDBC Type 2 driver

Many tools exist in the market for performance monitoring, and each tool has its own niche. Some are development tools allowing performance monitoring during the development process to help the development team write optimized code. Others are production tools allowing real-time performance monitoring on a production machine. We used the following performance monitoring tools to measure required metrics:

- ▶ IBM JInsight v2.1
- ▶ Sitraka JProbe Profiler with Memory Debugger Serverside Suite Edition v3.0.1
- ▶ Wily Technologies Introscope v2.6.1



The performance of an architecture

This chapter discusses architectural issues to be considered when designing a Java application.

2.1 Overview

Many factors affect end-to-end performance and scalability of an application. Here are some:

- ▶ Application architecture and network topology
 - Physical tiers
 - Logical tiers
 - Granularity of application components
 - Security
 - Network complexity and delays
- ▶ Business and technology requirements
 - Complexity of business logic
 - Hardware technology used
 - Software technology used
- ▶ Java runtime
- ▶ Web application server runtime
- ▶ Connectors to Enterprise Information Systems
- ▶ Code generated by tools

In the above list, certain factors are under the control of application architects and programmers, mainly the factors under application architecture. Hardware and software decisions are often a combination of business and technical requirements. When possible, they should be chosen so they are best suited for the application.

Most vendors have ongoing efforts to improve the performance of their products. IBM has been continuously working towards improving the performance of its Java runtime, WebSphere Application Server, connectors, development tools, and other pertinent factors, with a high degree of success. Although we may not control the performance metrics of products we use, we should decide upon the right products to use. Using a certain connector or tool may be crucial for an efficient application.

As applications are the main area we can control, they should be architected with business requirements and performance considerations in mind. Occasionally, performance may have to be put on a lower priority in order to achieve business requirements. However, where possible, performance should be a major factor in application architecture.

Figure 2-1 on page 7 shows possible topologies with WebSphere for z/OS.

Following are some general performance guidelines regarding WebSphere for z/OS:

- ▶ Using the HTTP Transport Handler provides the best performance when connecting to WebSphere on z/OS.
- ▶ Use of a processor with IEEE floating point support is recommended because it is widely used by Java.
- ▶ WebSphere on z/OS always uses security credentials, which affects performance. Carefully design and implement security to avoid too many security checks. Activate EJBROLES only if you really use it. Carefully choose the authentication mechanism to be used for checking users.

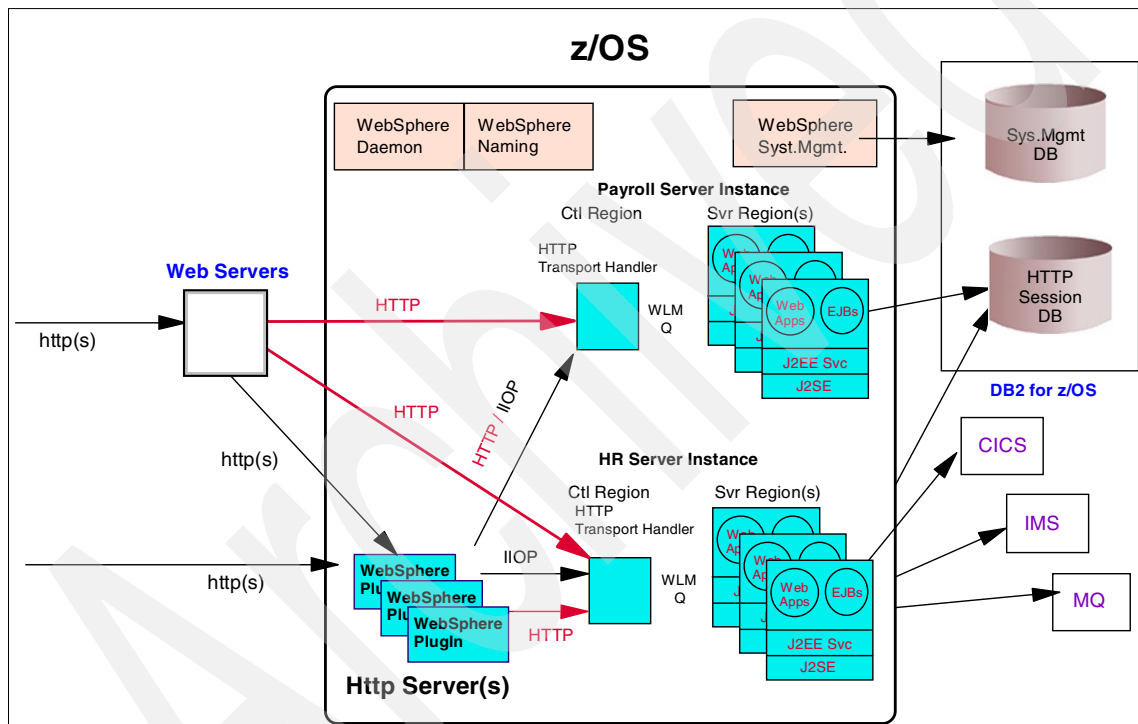


Figure 2-1 WebSphere for z/OS HTTP configuration options

The number of topologies shows the number of ways you can approach application development. Often the type and nature of an application influences the topology used. For Web applications, we know that the possible components are the browser, Web server, application server, databases, connectors, and application development and assembly tools. We need to make decisions concerning the programming models (servlet, J2EE), topology of the application (logical and physical tiers of the application), and actual application architecture.

In the remaining sections, we address some factors that have architecture performance impact.

2.2 J2EE architecture

How is the J2EE platform linked to performance? How can you take advantage of all its components?

Here are some benefits of the J2EE platform:

- ▶ Simplified architecture and development
- ▶ Scalability to meet demand variations
- ▶ Integration with existing information systems
- ▶ Choices of servers, tools, and components
- ▶ Flexible security model

We can split the design into three levels of architecture (Figure 2-2 on page 9):

- ▶ Use the application logic model as early as the conceptual design phase. Once the basic logic of the application and the key IT components (that is, servers, network, database) are known, you can then create a model representing the application and use it to perform what-if scenarios. This process helps you examine application performance under various conditions.
- ▶ The IT component model explores in detail the application's effect on a particular IT component, such as a database or an application server. Using the application logic model identifies the application's "hot spots" (critical points of high demand). An understanding of these points identifies the specific components that should be modeled.
- ▶ Toward the end of the development cycle, the enterprise model uses what you now know about the overall target environment, and helps you understand how this application performs with shared IT components and resources. The model results show the effect of other applications, as well as different load volume stresses, on the developed application.

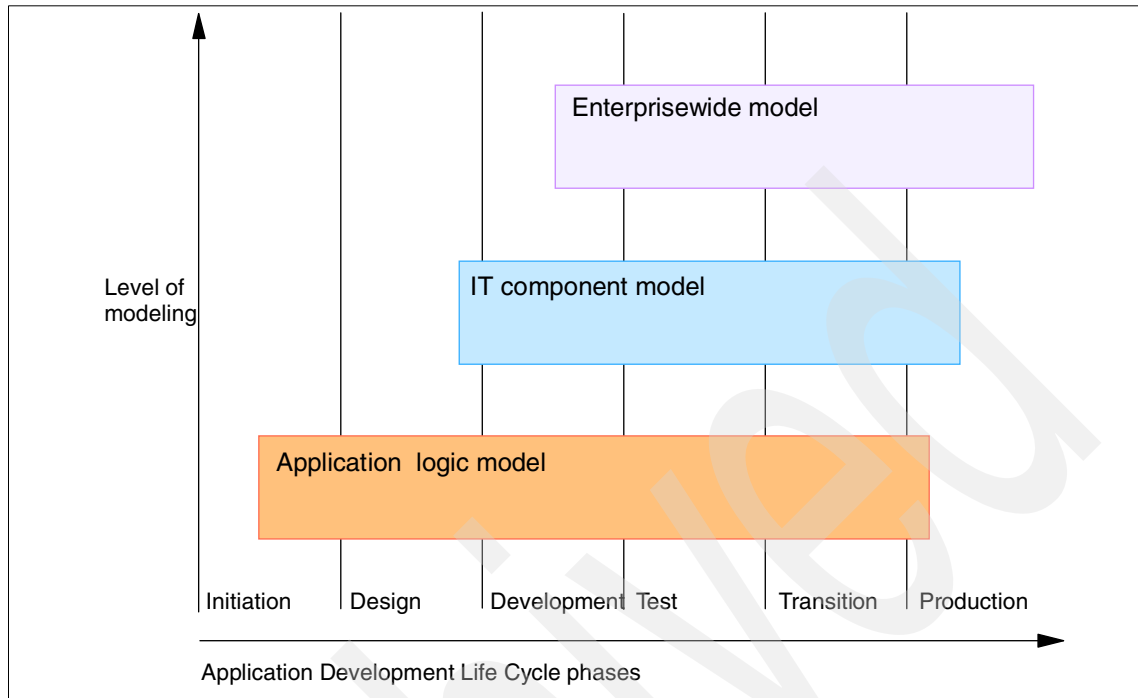


Figure 2-2 Levels of modeling

2.2.1 Model View Controller architecture

The Model View Controller (MVC) application architecture helps a lot in design and analysis. Its architecture is a way to divide functionality among objects involved in maintaining and presenting data so as to minimize the degree of coupling between the objects. The MVC architecture was originally developed to map the traditional input, processing, and output tasks to the graphical user interaction model. However, it is straightforward to map these concepts into the domain of multitier Web-based enterprise applications. The separation of presentation logic and business logic is defined by three parts:

- Controller

Manages and controls all interactions between the user and the application. Usually the controller is a servlet that receives the user request and passes the input parameters to the model that does the work. Also, when the business process ends, it controls the creation of the View.

- Model

Encapsulates the business logic, rules, and data, and does the business processing, usually implemented by JavaBeans or EJBs.

► View

Uses the results of the business process and constructs the response presented to the user, usually implemented through JavaServer Pages (JSPs).

Figure 2-3 shows the main components and functions of MVC.

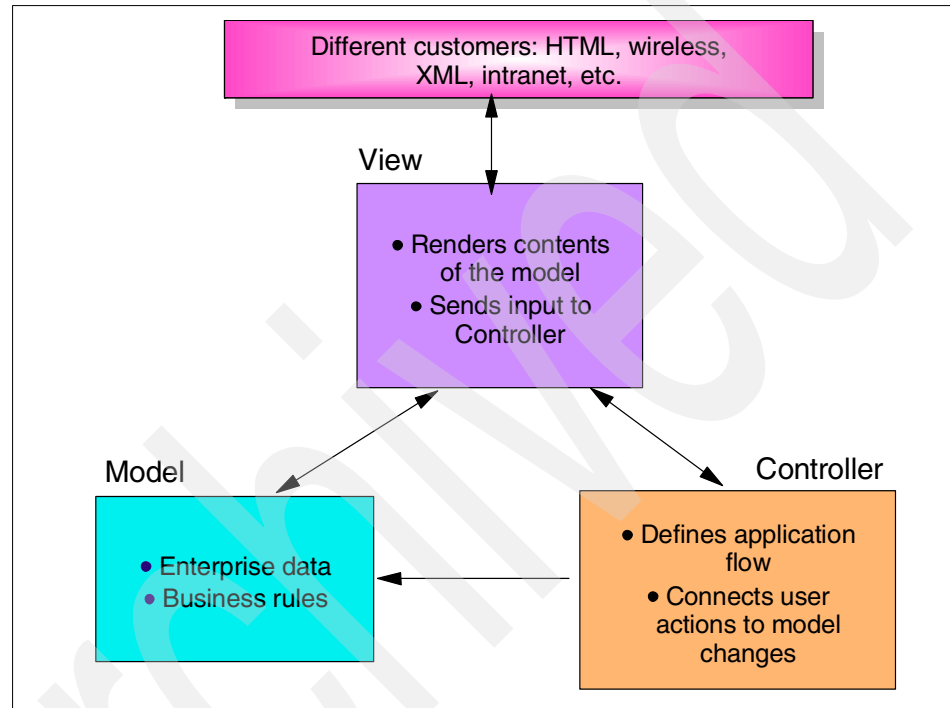


Figure 2-3 The Model-View-Controller architecture

Also important is the role separation in a development team, where the work of a page designer can smoothly integrate with the separate work of a business logic developer.

Based on the Model-View-Controller (MVC) architecture, you can, for example, use a session bean as a facade to an entity bean. The facade session bean increases performance by reducing the number of network calls. Besides performance improvements, the facade session bean hides the complexity of EJBs, leading to separation of presentation logic and business logic.

The MVC architecture enforces the separation of presentation logic and business logic in an application. The MVC model also offers the benefits of applying specific security models to the presentation logic and business logic. In fact, the separation of the logic, the business and the role allows you to have the flexibility to choose where you want to perform some security check. For instance, according to the MVC model, we have the EJB security model shown in Figure 2-4. We can control the security and allow the client to access the EJB facade or not, but we can go deeper than that by giving the client the right to execute only a specific method.

The J2EE security model is composed of several layers. Figure 2-4 shows a client request coming to a session bean where a level of security can be set (using an EJBROLE). Then we can protect each method of the EJB to allow only a defined user to execute it. This control can be achieved in a declarative way (based on an xml file: deployment descriptor) or a programmatic way (using J2EE methods: `isUserInRole`, `getCallerPrincipal`).

Once the client reaches the persistence layer, an authentication mechanism can be used to validate whether any action can be completed.

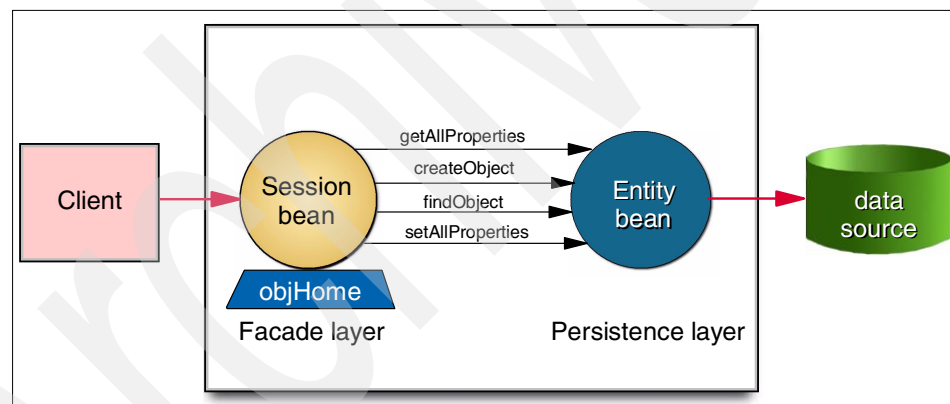


Figure 2-4 J2EE security levels

Attention: Note that using EJBROLES is expensive. The more you use it the more it affects performance, so use it only when you really need it.

2.2.2 J2EE components and performance

The J2EE architecture can be defined by the three main parts: the components, the containers, and the connectors. These are shown in Figure 2-5 on page 12.

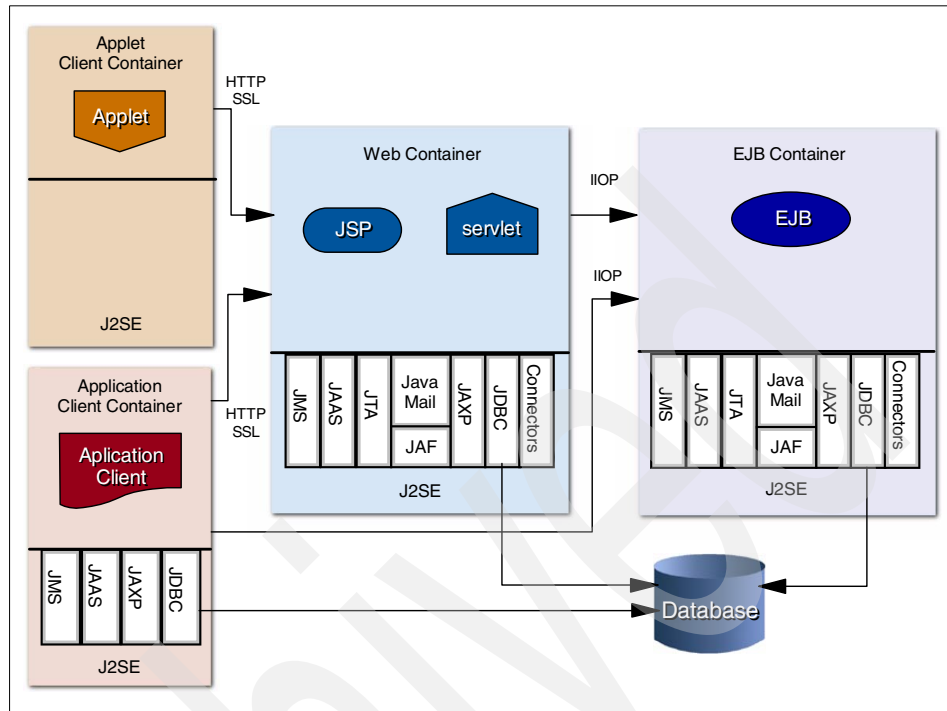


Figure 2-5 J2EE architecture

Components are the key focus of application developers. JSPs, servlets, and EJB components provide separation of presentation logic, controller logic, and business logic. Containers and connectors conceal complexity and promote portability.

Containers provide services to components transparently, including transaction support and resource pooling. Many component behaviors can be specified at deployment time, rather than in program code. Components inherit the qualities of service of the underlying platform container.

Connectors define a portable service API to plug into existing enterprise information systems, for example CICS and IMS transaction servers. Connectors promote flexibility by enabling a variety of implementations of specific services.

There are some general guidelines regarding performance that you should take into consideration during application design, for example, remote calls are more expensive than local calls.

Calling application components on other server regions or server instances or other systems is always more costly than using local calls with local components. Partition your application wisely to avoid those delays. Networking delay vs. more CPU usage is always a trade-off you have to make. Certainly there is a trade-off between application portability and performance, too. Using all components locally means less overhead, but this means you cannot put the components of the application onto any server you may want or you have to cope with the additional network delay and additional CPU cost of the remote calls.

Servlets and JSPs

For applications that require complex modeling on the Web application server node, it is not easy to define the granularity of servlets and how servlets interact. But without a good design for the servlets and JSPs it is hard to maintain the application.

Problems with performance

The granularity of artifacts on the server can impact the response time for a client request. This can be due to a business object with a lot of logic that needs to perform several calls to get its reply on the server. Then this “big” object needs, for instance, to analyze, format and send back the reply to the client. This can also be done through multiple calls. The granularity of the objects transmitted through each call can consume a significant amount of system resources.

One extreme design approach is to have only one servlet-per-use case. The servlet then acts as the central event handler and processes all requests from the client. It executes the necessary action for that event and forwards the request to one (of many) JavaServer Page for displaying the result. By using this solution, it may be difficult to develop that servlet. Because it is responsible for a whole use case, we may have to implement a lot of logic in that servlet.

This can cause efficiency concerns and make programming difficult.

The other extreme is to have as many servlets as JavaServer Pages and to “chain” them. This means that a servlet gets a request, executes the proper action, and calls the JavaServer Page specific to that servlet to display the result. A request from that JSP then gets to another servlet and so on.

This approach is hard to maintain since having many servlets and JSPs can get confusing when trying to understand the flow of the application.

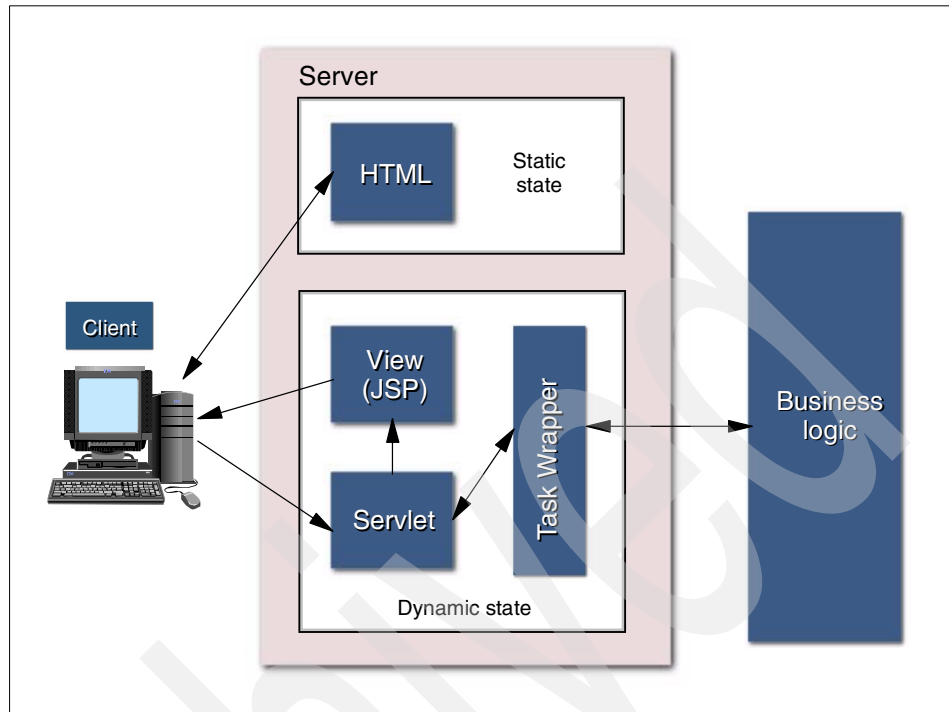


Figure 2-6 Structure of the servlet/JSP pattern

A JavaBean wrapper is designed to allow either a session or entity enterprise bean to be used like a standard JavaBean and it hides the enterprise bean home and remote interfaces from the developer (this is an access bean). This wrapper is running in the EJB container, so some local calls (“within the EJB container”) are done.

It is a good idea to encapsulate that session bean with a Java class, so that it is easier for the servlet programmer to access the session bean. Therefore, we need a Java class with the same methods as in the session bean. If we use a class with static methods for accessing the session bean, it is easy to cache the InitialContext, rarely changing read-only data within one JVM and so get a performance benefit.

Entity EJBs

If an entity bean is the target of a container-managed relationship, then it must have local interfaces. The direction of the relationship determines whether or not a bean is the target. In Figure 2-7 on page 15, for example, the ProductEJB bean is the target of a unidirectional relationship with the LineItemEJB bean. Because the LineItemEJB accesses the ProductEJB locally, the ProductEJB must have

the local interfaces. The `LineItemEJB` also needs local interfaces—not because of its relationship with the `ProductEJB`, but because it is the target of a relationship with the `OrderEJB`. And because the relationship between the `LineItemEJB` and `OrderEJB` is bidirectional, both beans must have local interfaces.

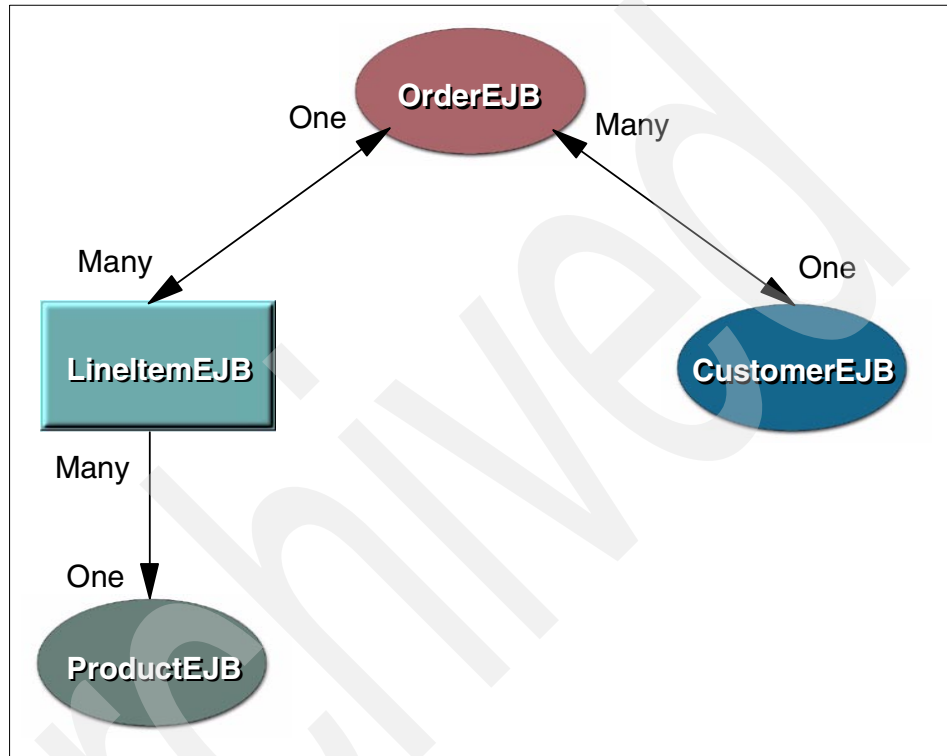


Figure 2-7 EJB relationships

Because they require local access, entity beans that participate in a container-managed relationship reside in the same EJB jar file. The primary benefit of this locality is increased performance because local calls are faster than remote calls.

Performance and access

Because of factors such as network latency, remote calls are slower than local calls. On the other hand, if you distribute components among different servers, you might improve an application's overall performance or scalability. Both of these statements are generalizations; actual performance can vary in different operational environments. Nevertheless, you should keep in mind how your application design might impact performance.

Granularity of accessed data

Because remote calls are likely to be slower than local calls, the parameters in remote methods should be relatively coarse-grained. Since a coarse-grained object contains more data than a fine-grained one, fewer access calls are required.

For example, suppose that a CustomerEJB is accessed remotely. This bean would have a single getter method that returns a CustomerDetails object, which encapsulates all of the customer's information. But if the CustomerEJB is to be accessed locally, it could have a getter method for each instance variable: getFirstName, getLastName, getPhoneNumber, and so forth. Since local calls are fast, the multiple calls to these finer-grained getter methods would not significantly degrade performance.

Session EJBs

A stateless session bean does not maintain a conversational state for a particular client. When a client invokes the method of a stateless bean, the bean's instance variables may contain a state, but only for the duration of the invocation. When the method is finished, the state is no longer retained. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client.

Because stateless session beans can support multiple clients, they can offer better scalability for applications that require large numbers of clients. Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients. At times, the EJB container may write a stateful session bean out to secondary storage. However, stateless session beans are never written out to secondary storage. Therefore, stateless beans offer better performance than stateful beans.

To improve performance, you might choose a stateless session bean if it has any of these characteristics:

- ▶ The bean's state has no data for a specific client.
- ▶ In a single method invocation, the bean performs a generic task for all clients. For example, you might use a stateless session bean to send an e-mail that confirms an online order.
- ▶ The bean fetches from a database a set of read-only data that is often used by clients. Such a bean, for example, could retrieve the table rows that represent the products that are on sale this month.

Access beans

Access beans adapt an Enterprise JavaBean to the JavaBean's programming model by hiding the home and remote interfaces from the users. You do not have to manipulate those interfaces directly. In addition, an access bean provides fast access to enterprise bean data, because it maintains a local cache of attributes. Access beans have been designed specifically to support servlet and JavaServer Page programming, but they can also be used to access an enterprise bean from another enterprise bean or any application.

Note that access beans do not contain any proprietary code, so they can be used with any application server. There are three types of access beans:

- ▶ Java bean wrapper

Using this access bean, an enterprise bean can easily be consumed by a visual builder tool. This is the only access bean available for session beans.

- ▶ Copy helpers

Access beans may employ copy helper objects that are basically caches of user-selected entity bean attributes that are stored inside the access bean. A servlet or JavaServer Page component does not need to make remote calls to obtain the attribute values. The *getter* and *setter* methods for these attributes deal directly with the local cache rather than calling straight through to the remote getter and setter call. Methods are provided to flush the cache to the actual enterprise bean database and to refresh the cache from the actual enterprise bean. This can improve performance significantly for entity enterprise beans that have a large number of attributes, where issuing one remote call to get and set a large number of attributes is faster than issuing a single remote call for each attribute. A copy helper access bean also has all of the characteristics of a Java bean wrapper.

- ▶ Row-set

Row-set access beans contain multiple copy helpers. They are used to manipulate a collection of access beans.

2.2.3 World of patterns

Patterns for e-business are a group of reusable assets that can help speed the process of developing Web-based applications. Patterns break these reusable assets down into the following elements:

- ▶ Business patterns identify the interaction between users, businesses, and data. Business patterns are used to create simple, end-to-end e-business applications.

- ▶ Integration patterns connect other Business patterns together to create applications with advanced functionality. Integration patterns are used to combine Business patterns in advanced e-business applications.
- ▶ Composite patterns are combinations of Business patterns and Integration patterns that have themselves become commonly used types of e-business applications. Composite patterns are advanced e-business applications.
- ▶ Application and Runtime patterns are driven by the customer's requirements and describe the shape of applications and the supporting runtime needed to build the e-business application.
- ▶ Product mappings to populate the solution. The product mappings are based on proven implementations.
- ▶ Guidelines for the design, development, deployment, and management of e-business applications.

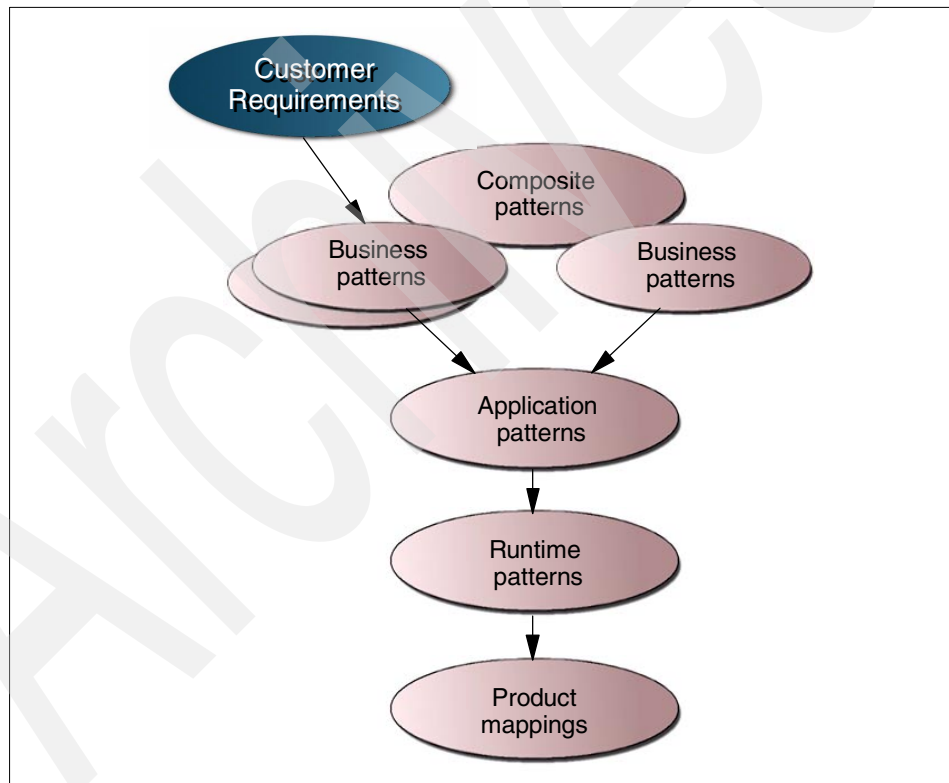


Figure 2-8 From requirements to products

The patterns leverage the experience of IBM architects to create solutions quickly, whether for a small local business or a large multinational enterprise. As shown in Figure 2-8 on page 18, customer requirements are quickly translated through the different levels of pattern assets to identify a final solution design and product mapping appropriate for the application being developed. You may also see 4.1, “Applying design patterns” on page 58 regarding Design patterns.

For more information on this subject, refer to the following Web site:

<http://www-106.ibm.com/developerworks/patterns/>

2.3 Number of tiers

The number of tiers often depends on the complexity of existing applications and their proprietary interfaces. More complexity and proprietariness of the existing application leads to more middleware required for integration.

2.3.1 Logical tiers

In the previous section, you read about the J2EE application model and its benefits. Additionally, the model-view-controller (MVC) model is gaining popularity these days. The MVC model separates an application into three parts:

- ▶ **Presentation logic (View)**
Constructs the user interface or page/document to be returned to the user.
- ▶ **Controller logic (Controller)**
Controls the model and view.
- ▶ **Business logic (Model)**
Implements the business model.

This creates three independent logical tiers and allows a separation of graphical, programming, and enterprise skills, resulting in assignment of personnel into the areas they are best suited for.

J2EE accounts for all three of the components in the MVC model through its specifications for Java Server Page (JSP), servlet, and Enterprise Java Bean (EJB), as shown in Figure 2-9.

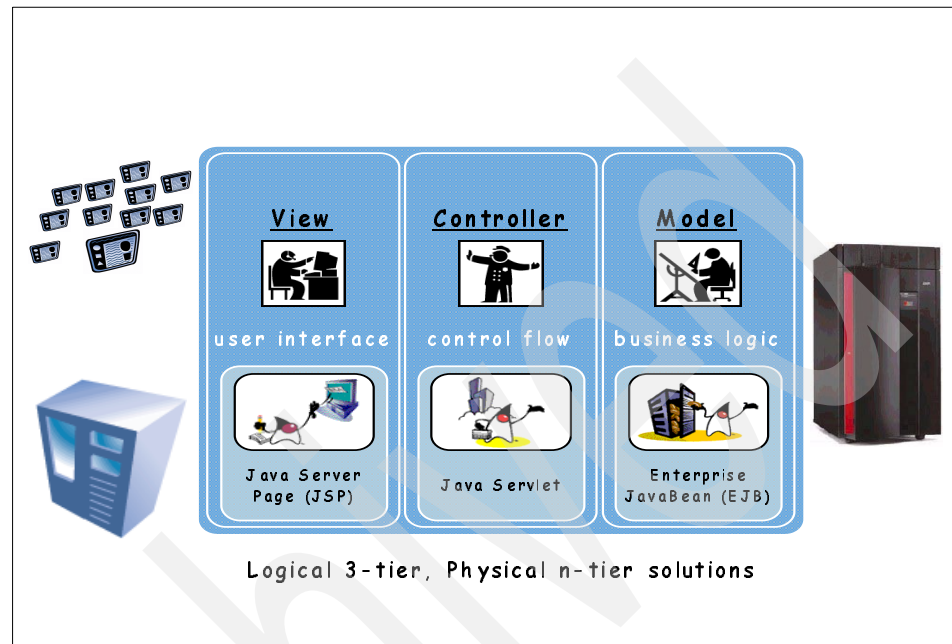


Figure 2-9 MVC model and the J2EE counterparts

This allows us to separate skills in the following manner:

- ▶ Employees with graphical skills can be assigned to develop JSPs to handle the user interface and the presentation of the application.
- ▶ Similarly, programming skills may be applied to developing servlets, which control the presentation and the business logic.
- ▶ Finally, enterprise skills can be focused towards the core business logic to develop EJBs.

We will be adhering to this MVC model throughout this book.

2.3.2 Physical tiers

In many cases security requirements increase the number of physical tiers implemented. Increasing the number of physical tiers generally increases the path length. And increased path length leads to latency resulting in poor performance. But, again, the number of tiers often depends on the existing environment and the type and nature of the application.

Let's look at four different topologies to get an idea of advantages and disadvantages of each one.

Figure 2-10 shows a Web-enabled client/server solution using a thin client and a thick middle tier connecting to the back end.

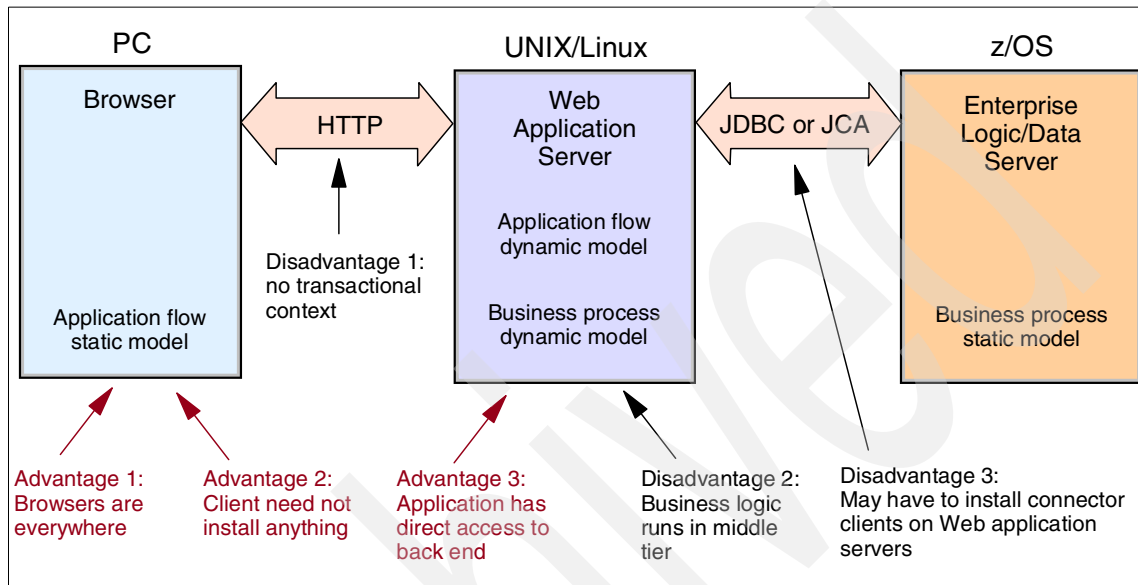


Figure 2-10 Topology 1: three-tier Web-enabled thin client, thick middle tier

The thin clients (browsers) provide an advantage since no special code needs to be implemented on the client. Another performance advantage is that the middle tier can access the data directly. However, this also creates a disadvantage because there is no transactional context, no separation between presentation/controller and business logic.

Figure 2-11 shows a distributed-objects solution using a thick client and a distributed object server on the middle tier.

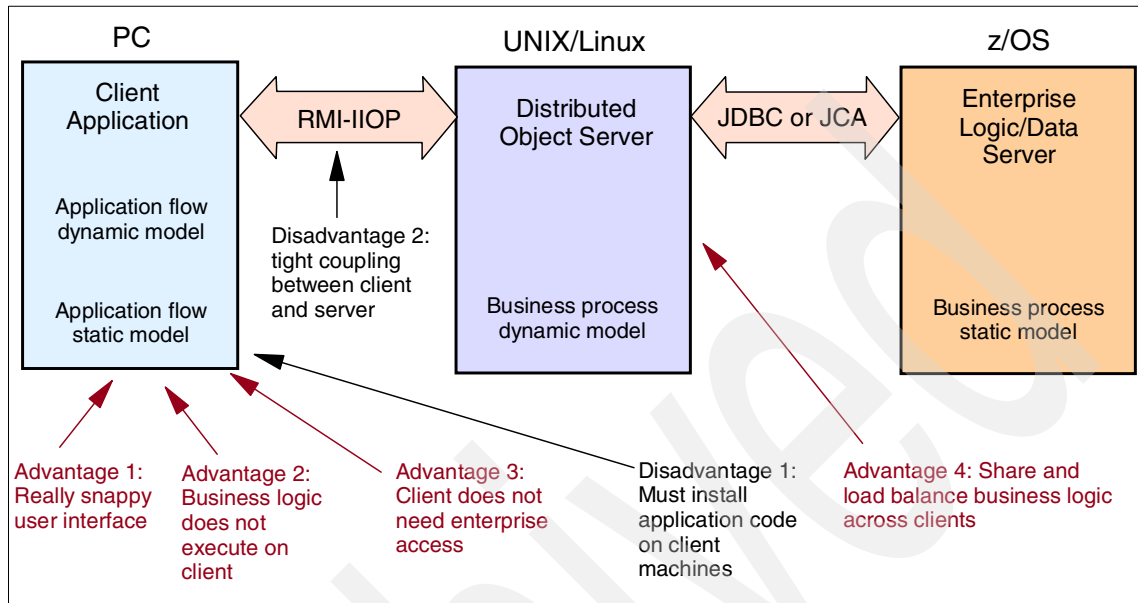


Figure 2-11 Topology 2: Distributed objects using three tiers, thick client

The middle tier implements a distributed object server, which provides the main advantage in its ability to share and load balance business logic across clients. However, this scenario creates a major disadvantage due to the need of installing some application code on the client machines. When the number of client machines increases, so will the management and maintenance costs.

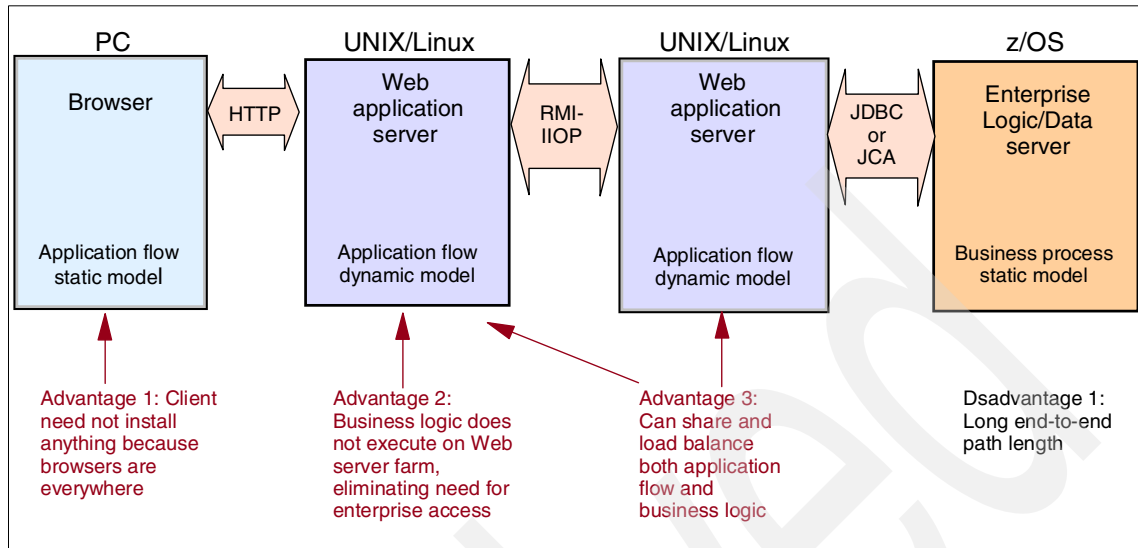


Figure 2-12 Topology 3: Web-enabled distributed objects

Again, thin clients (browsers) provide an advantage because no special code needs to be implemented on the client. Other advantages include the separation of presentation/controller and business logic and the load balancing of the application flow and business logic. The biggest disadvantage is the long end-to-end path length, resulting in higher latency and poor response time.

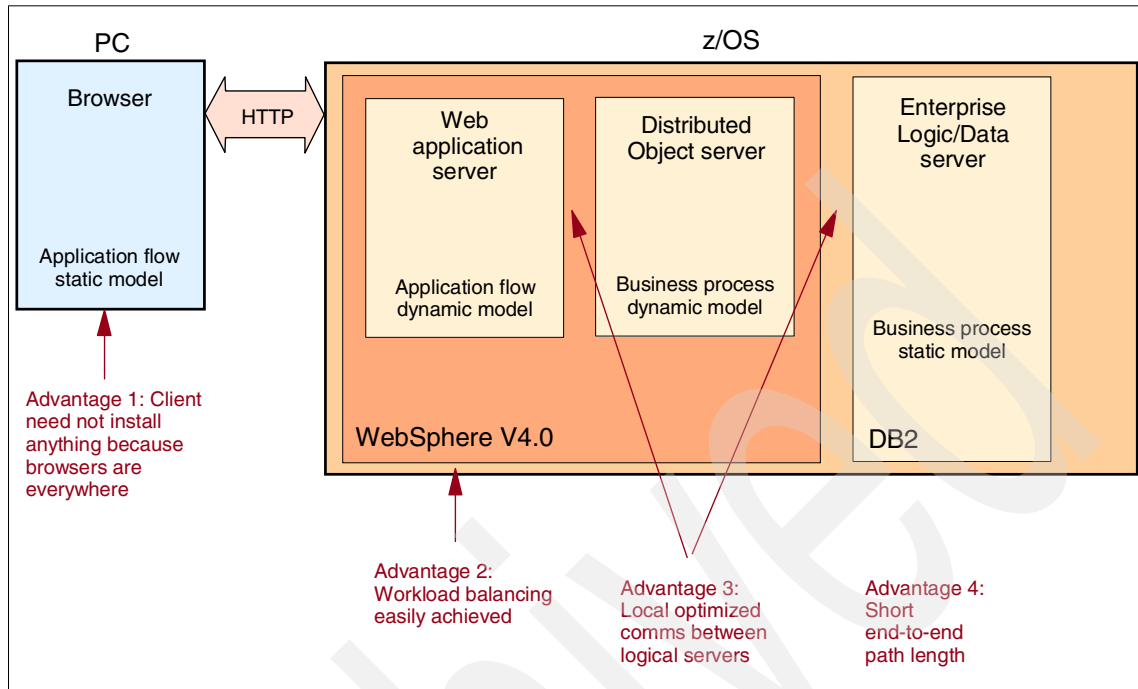


Figure 2-13 Topology 4: Web-enabled two-tier

This topology has all of the advantages of the previous topology. However, the physically long end-to-end path length has been removed by implementing the logical tiers on one server, each logical tier in its own logical partition (LPAR). This results in lower latency and better response time, mainly due to HiperSockets. The zSeries HiperSockets, also known as internal queued direct input/output (IQDIO), is a microcode function supported by z/OS V1R2. HiperSockets operate with minimal system and network overhead, and eliminate the need to utilize I/O subsystem operations and to traverse an external network connection to communicate between LPARs.

Table 2-1 summarizes the pros and cons of each topology mentioned.

Table 2-1 Comparison of topologies

	Advantages	Disadvantages
Topology 1 Web-enabled 3-tier	Thin clients (browsers) readily available. Application has direct access to back end.	No transactional context. Merging of presentation, controller, and business logic.
Topology 2 Distributed objects 3-tier	Share and load balance business logic across clients.	Poor manageability and maintenance.
Topology 3 Web-enabled distributed objects	Thin clients (browsers) readily available. Separation of presentation, controller, and business logic. Share and load balance application flow and business logic.	Long physical end-to-end path length.
Topology 4 Web-enabled 2-tier	All advantages of Topology 3. Short physical end-to-end path length.	

2.4 Scalability

Scalability is inseparable from performance. How does the application perform when the number of requests increases, can it keep up with the performance with higher workload? Of course, a scalable application must meet performance criteria before scaling. When designing applications, we not only take performance into consideration, but should also keep scalability in mind.

From a system point of view, we can study the scalability of each component in the architecture to identify potential bottlenecks. Hardware components include CPU speed, a cluster machine's multi-processor effect, memory, cache, network, storage, I/O devices, etc. Software components include operating system, database, security server, network software, application server or transaction server, connectors, and the application itself.

For Java applications, there are some important factors that affect scalability, like threading, garbage collection, session handling, etc. But the most important element is the JVM itself.

Performance and scalability are key requirements for Java on OS/390 and z/OS. For the last several years, OS/390 JVM has improved performance significantly over releases, which can be seen in Figure 2-14 and it also demonstrates that late releases of OS/390 JDK scales very well as the number of threads increases.

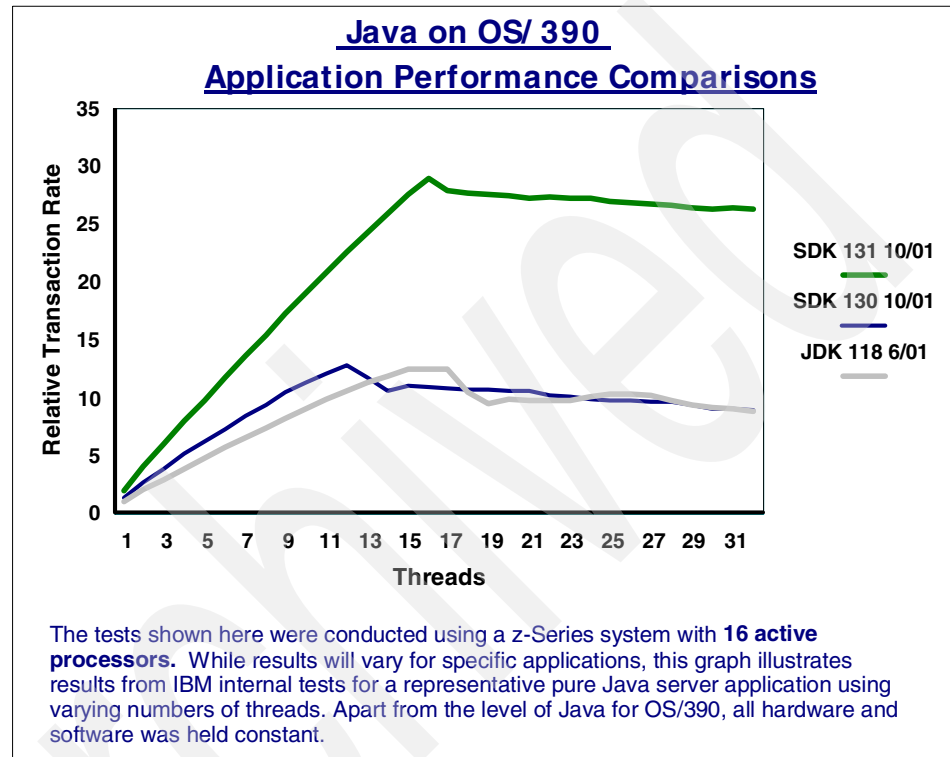


Figure 2-14 Java for OS/390 application performance comparisons

In most cases, a scalable application is a multithreading application. That does not mean the application itself has to always create and manage threads. The evolving Java technology has made application development much easier.

For example, servlet technology and EJB technology move the job of multitasking application execution to the Web container and EJB container in the application server, and multiple instances of a servlet are created by the Web container to handle concurrent requests. So selecting an application server that provides good performance and scalability is crucial to the success of e-business applications. Threading is just one example out of many that can affect application server scalability.

In the multithreaded execution environment, trouble situations like race condition, deadlock, lack of resources, and other problems associated with concurrency are common and difficult to debug. Although servlet developers do not have to manage threads in the programs, they need to be aware of the multithreaded execution environment during development to develop thread-safe applications.

Synchronization can be used to solve some concurrency issues. In Java, methods and code blocks may be marked with the *synchronized* keyword. Multiple threads cannot simultaneously execute with any of the synchronized methods on the same class instance, and a synchronized code block is treated like a synchronized method on the object that is provided as an argument.

But synchronization should be used sparingly for two main reasons. First is the overhead of the synchronization mechanism itself. Somewhere a lock must be acquired and released before and after the synchronization code. A more important issue with careless use of synchronization is the impact of thread contention on performance. Synchronization on large code segments causes long wait on the lock and potentially impacts scalability. Many lockings that are shorter force frequent context switching on the thread if the lock acquiring fails.

Another example of impact on scalability are session beans.

A stateful session bean retains conversational data between method calls and across transactions. It is kept in the container for a specific client as long as the client holds onto the session or until time-out. A stateless session bean does not have the data that needs to be maintained beyond the scope of a single method call. So any free instance of stateless session bean can be picked by the container to respond to a client method call.

From a scalability point of view, the container needs to create more stateful session bean instances to keep up with client requests. The number of stateless session bean instances can be smaller and aid garbage collection. Application designers and programmers should make the right design choice of stateful and stateless session beans based on function and performance requirements. If stateful session beans are used, they need to be disassociated from clients as soon as possible and the time-out period should not be too long.

Tools

In this chapter we introduce the performance monitoring and profiling tools. We deal with Introscope from Wily Technology, JInsight from IBM, and JProbe from Sitraka. We offer a comparison of the tools and discuss the main capabilities.

Important: Many of JInsight's features are now being made available as part of the WebSphere Studio Application Developer profiling tool. Check the Application Developer documentation for the latest features of that profiling tool.

JInsight is a prototype technology and so there is no support from IBM.

3.1 Performance monitoring

The complexity of a production Web environment is increasing with:

- ▶ HTTP Web Server
- ▶ Web Application Server
- ▶ Database Server
- ▶ Transaction Server
- ▶ Security Server
- ▶ Messaging Server, and more....

You need to know what is happening in the WebSphere Application Server, and how the applications are performing.

The WebSphere Application Server executes critical components of e-business applications such as servlets, JSPs, EJBs, etc. These in turn interact with several enterprise resources. e-business application performance and response depend on the critical components that execute in the WebSphere Application Server and the resources they depend on. Therefore, it is necessary to have visibility inside the WebSphere Application Server to monitor performance and response time of these components and identify any bottlenecks.

Figure 3-1 on page 31 shows a typical scenario.

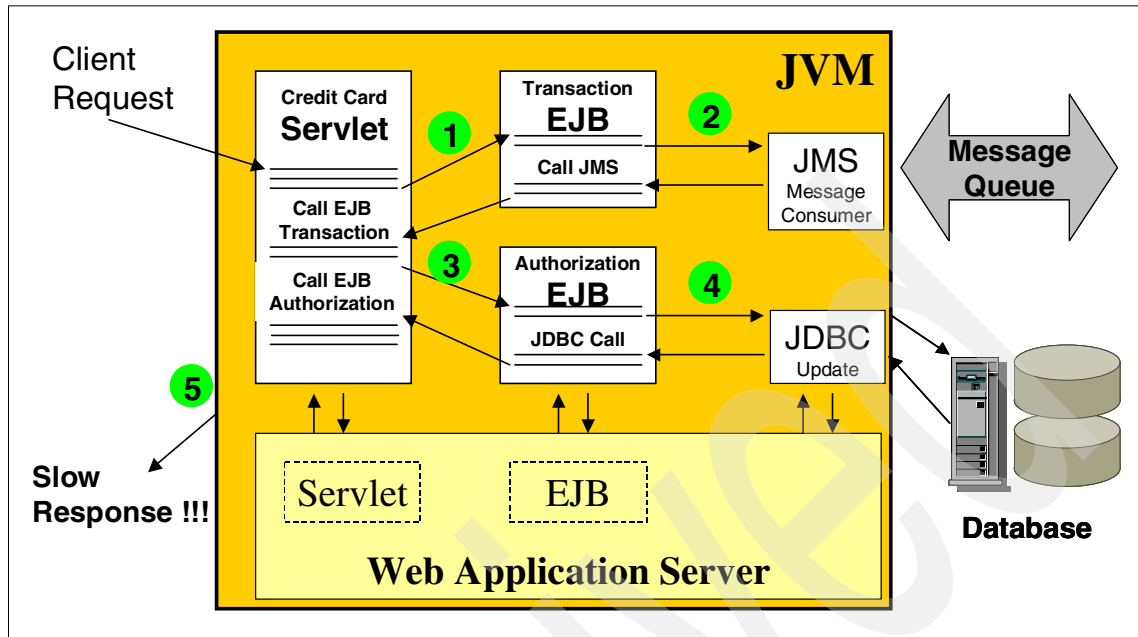


Figure 3-1 Visibility inside WebSphere

The sequence of actions needed to handle a client request is:

1. The Credit Card Servlet calls the Transaction EJB.
2. The Transaction EJB calls JMS and returns to the servlet.
3. The Credit Card Servlet then calls the Authorization EJB.
4. The Authorization EJB calls JDBC and returns to the servlet.
5. The Credit Card Servlet responds to the client request.

Tools used to monitor production applications

Application operators and administrators need to monitor these interactions between components in WebSphere to identify bottlenecks. In large applications with many components it is necessary to quickly drill down and identify the problem areas, so that efforts can be focussed on them.

Additionally, they also need to conduct historical performance analysis and trend analysis. Java application performance management tools that are designed to scale up and monitor live production Web sites in real-time are a significant asset in the production support function.

These tools can also be used in test cycles to profile entire applications before moving to production. Typically, these tools collect macro level metrics while incurring very low overhead (<5%), and this makes them suitable for production monitoring. Wily's Introscope is one such tool that can be used for monitoring a production environment.

Tools for Java code analysis and optimization

Once problem areas are identified, Java code analysis tools can be used to explode Java programs that need to be profiled. Java program profiling tools are useful for visualizing and analyzing the execution of Java programs, as follows:

- ▶ Performance analysis, tracing each execution thread, execution time and resource utilization
- ▶ Memory leak diagnosis
- ▶ Debugging or understanding what a Java program is really doing

Object-oriented Java programming hides the complexity and overhead of behavior and methods. These tools can be used to profile code, measure execution activity and memory usage—summarized at any level of detail and along call paths. JInsight by IBM, Sitraka's JProbe, and Wily Technology's Introscope are tools that can be used for Java code profiling.

There is a general guideline which should be followed with any tool: Do not switch on all “bells and whistles” of the trace or monitoring product. The more you trace, the more it will impact performance.

3.2 Introduction to tools

We deal with the tools in Table 3-1.

Table 3-1 Comparison of performance tools

Features	JInsight	JProbe	Introscope
This tool is suitable for profiling code in a <i>development environment</i> .	Yes. Use it to trace execution threads and identify areas for improvement.	Yes. Use it to trace execution threads and identify areas for improvement.	Yes Suitable for macro-level analysis during development and testing.

Features	JInsight	JProbe	Introscope
This tool is suitable for monitoring applications in a <i>production environment</i> .	No	No	Yes Low overhead tool, suitable for monitoring production systems
Profile applications	Yes Suitable for profiling Java classes and threads at a detailed level	Yes Suitable for profiling Java classes and threads at a detailed level	No Suitable for production monitoring with low overhead
Dynamically start and stop collecting application metrics	Yes	No	Yes
Requires changes to the application code	No	No	No
Provides a dynamic view of the application	No	Yes	Yes
Permits selection of objects to be traced	No	Yes	Yes Can select specific objects and drill down for details
Filter and select specific objects	Yes	Yes	Yes
Tracing granularity	Method Thread Object	Thread Object	Component, class or method
Metrics measured	Execution time, number of calls, object memory consumption	Number of calls, JVM memory usage, execution times	Response time, throughput, JVM memory, frequency, count, CPU utilization
Garbage collection tracing	Yes	Yes	Yes

Features	JInsight	JProbe	Introscope
Compatible with WebSphere Application Server V4.01	Yes	Yes	Yes

3.3 Introscope

Key features

1. Introscope can provide key live metrics on:
 - a. Java applications
 - i. Response times (keep whatever you have)
 - ii. Rates (keep whatever you have)
 - iii. Counts of total number of component (servlets, JSP, JMS, JTA, EJBs, etc.) invocations, active threads, beans
 - b. Application server performance and resource utilization
 - i. JDBC Connection Pool
 - ii. Servlet and Orb Thread Pool
 - iii. HTTP sessions
 - c. JDBC call performance

SQL Agent is an Introscope module that monitors the performance (round trip time) of individual SQL statements from the Java application perspective. It enables the JDBC driver for performance monitoring, collects performance metrics on individual SQL statements and reports data back to Introscope just like any other data gathered on J2EE components. Like other J2EE components, SQL statements can also be blamed by Introscope for performance bottlenecks.
 - d. Backend systems performance
 - i. CICS transaction performance
 - ii. WebSphere MQ Messaging and connection performance
 - e. JVM and OS performance
 - i. Memory
 - ii. CPU utilization
 - iii. File and socket I/O performance
2. It introduces low operational overhead and can be used for performance monitoring in live production applications.

Introscope works seamlessly with WebSphere Application Server and also with any standalone Java application without requiring any development effort. AutoProbe integration between Introscope and WebSphere allows automatic management of any Java application running in WebSphere. When

Java application classes are loaded by the WebSphere classloader into the JVM, Introscope AutoProbe automatically enables the Java application for performance monitoring. This allows Introscope to collect performance metrics from J2EE components, monitor interactions between components, and most importantly, blame performance bottlenecks on specific components.

3. It provides needed visibility into the JVM and Java applications at the component level. It can be used to watch applications, servers, JVMs, and all the Java components, and pinpoint bottlenecks.
4. While managing large applications, Introscope can significantly reduce time needed for problem identification and resolution. If the bottleneck components can be quickly identified, be it servlets, EJBs, JDBC connections, or connectors, etc., efforts can be focussed on the problem areas and downtime minimized. This is achieved through Introscope's unique "Blame Technology". It tracks every component interaction involved in delivering a single transaction, and leads the user down the path of bottleneck determination.
5. Performance metrics can be stored in a database and a variety of informative comparative reports can be generated. This is useful for analyzing service levels and performance trends, and for capacity planning.

Using Introscope to identify bottlenecks

Consider the following example with CreditCardServlet and Authorization EJB. Introscope performance metrics show (see Figure 3-2 on page 36):

1. Spikes in overall response time.
2. Credit servlet's response time is OK.
3. Authorization EJB's response time has increased.
4. JDBC call response time has a spike.

These data lead us to conclude that the JDBC connection/DB Server is causing the performance bottleneck.

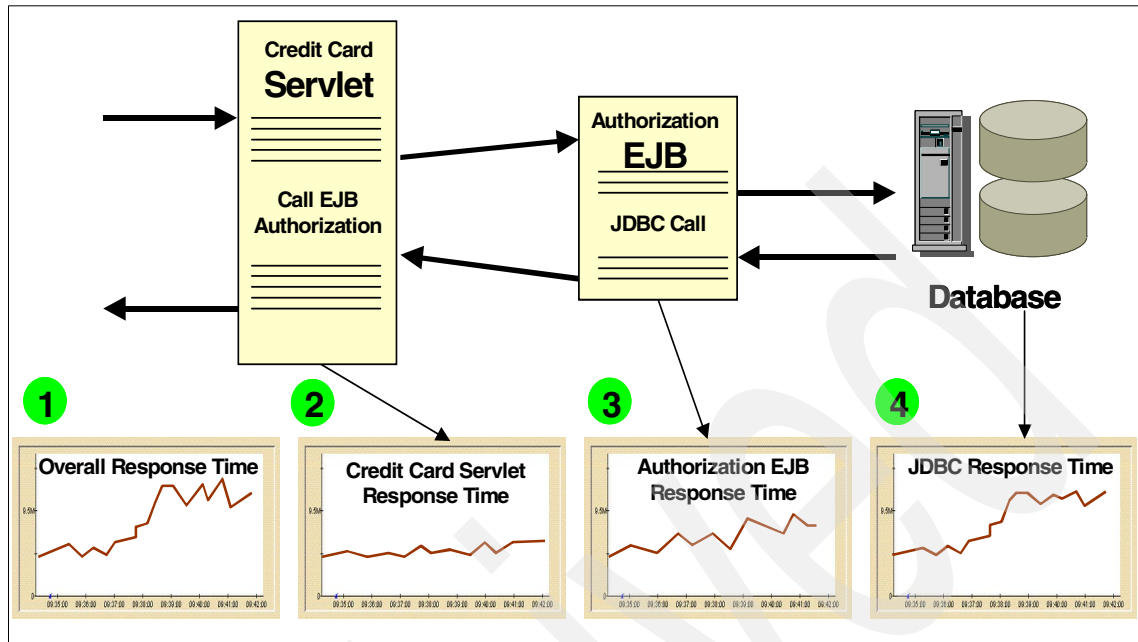


Figure 3-2 How to identify bottlenecks

How Introscope works

Introscope has three components (Figure 3-3 on page 37):

1. *Agents* collect performance metrics from WebSphere and applications that are to be monitored.
2. Agents pass performance data to *Enterprise Manager* for persistence to a database and analysis.
3. Enterprise Manager sends data to *Workstation GUI*, produces reports, or sends data to SNMP-enabled systems.

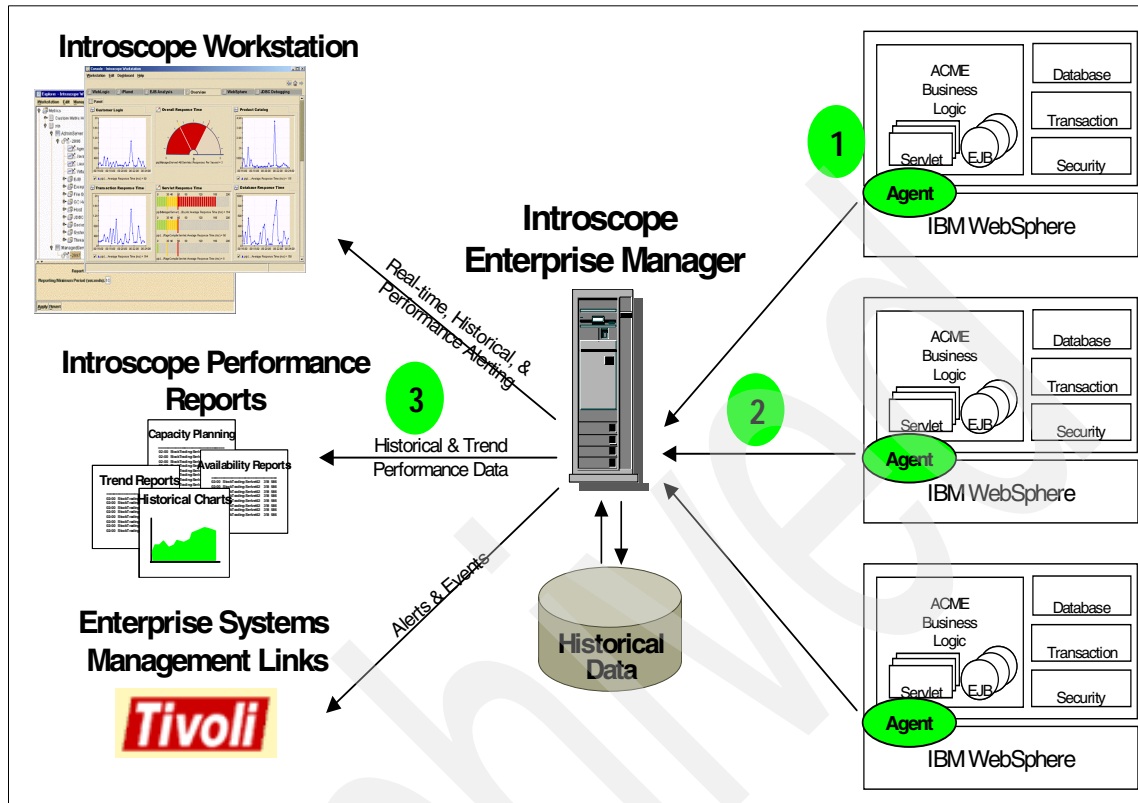


Figure 3-3 Introscope monitoring architecture

Monitoring WebSphere applications with Introscope

For the following steps, see Figure 3-4 on page 38.

1. Install and configure the Introscope Enterprise Manager. Identify the port on which it expects to listen to Agents (managed application) and the port on which it will connect with a workstation client.
2. Configure the WebSphere instance to report metrics to the Introscope Enterprise Manager. For more information on this refer to the Introscope Users Guide published by Wily Technology and chapter 9 of *Migrating WebSphere Applications to z/OS*, SG24-6521.
3. The application becomes "Introscope-enabled" in one of two ways: Use AutoProbe (available on WebSphere 3.5.2 and later) to seamlessly add instrumentation at deployment time, or separately run the ProbeBuilder utility to add instrumentation before deployment of code.

4. Introscope Agent collects performance metrics from instrumented application and sends performance data to the Introscope Enterprise Manager.
5. Enterprise Manager displays data on Introscope workstation.

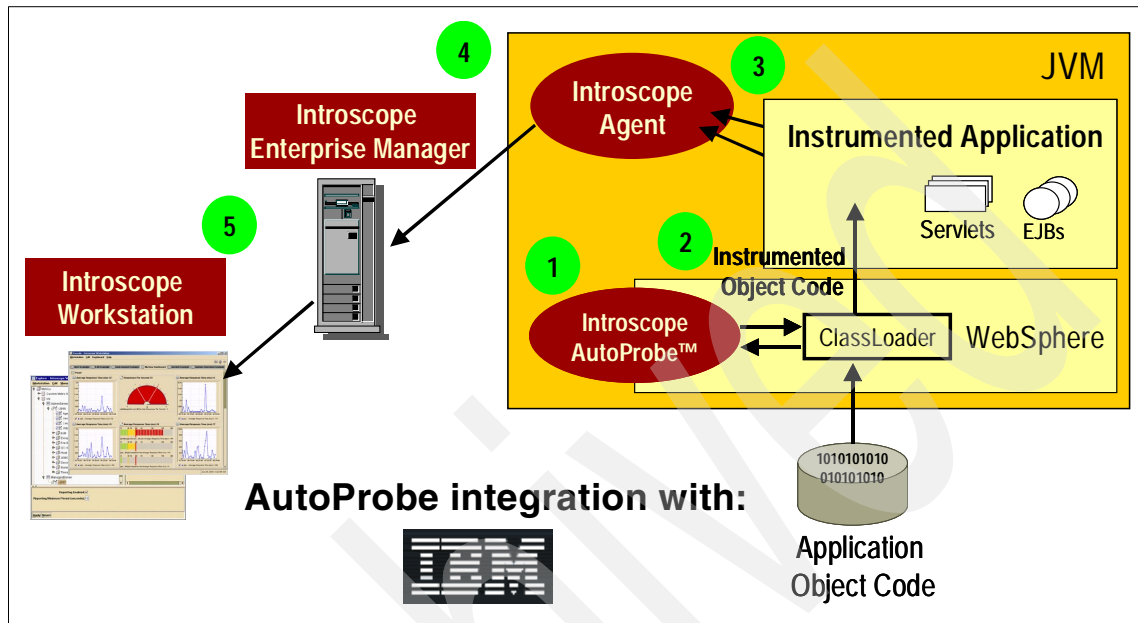


Figure 3-4 Integration with WebSphere

Introscope workstation

Data collected by the Enterprise Manager can be accessed through one or more workstations. The workstation interface is composed of two windows, the Console and the Introscope Explorer. Both display information collected by Introscope. You can use the workstation to view performance data and configure the Enterprise Manager to perform such tasks as collecting information for later analysis, and creating alerts.

Alerts take status information from a specific measurement of performance data, compare it to user-defined thresholds, and produce a status: normal, caution, or danger. Alerts can notify the user about the performance degradation in a number of ways: e-mail, pager, or a workstation notification as shown in Figure 3-5 on page 39.

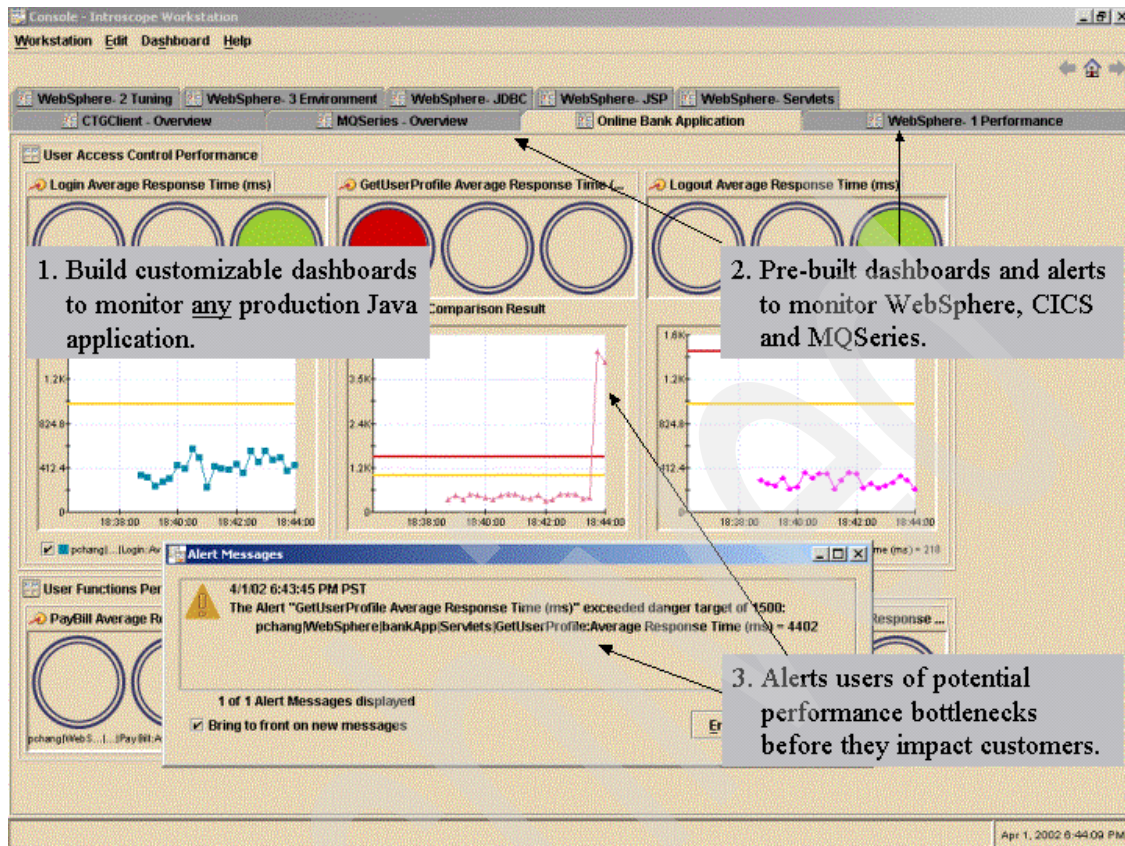


Figure 3-5 Introscope sends alerts

You use the Explorer to set up elements such as Alerts and Calculators that filter performance data, so that you can view it in a meaningful way in the Console.

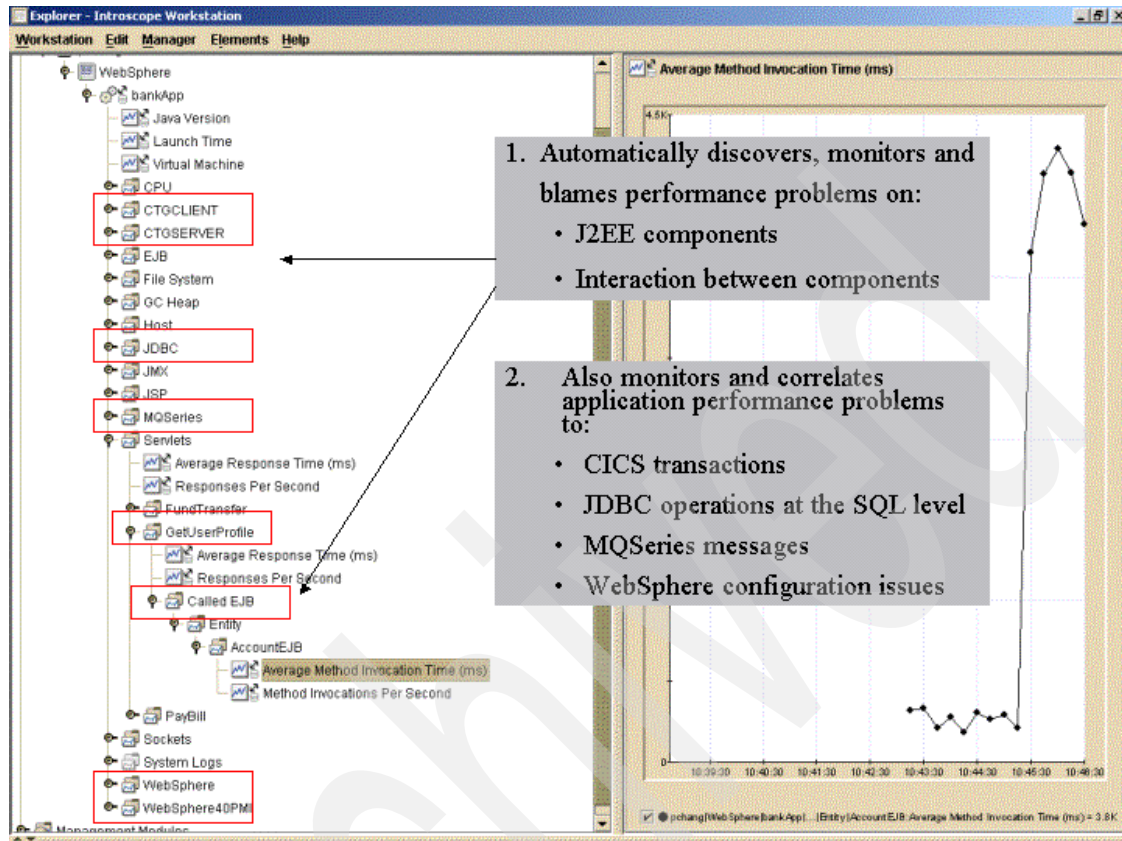


Figure 3-6 Introscope reveals the component hierarchy

Introscope contains Blame technology, which is used to show the components involved in the Java application and also to isolate the problem to a specific component. For example, if an application login takes a long time, Blame technology can pinpoint the component responsible for the problem. This process is shown on Figure 3-6 and Figure 3-7 on page 41.

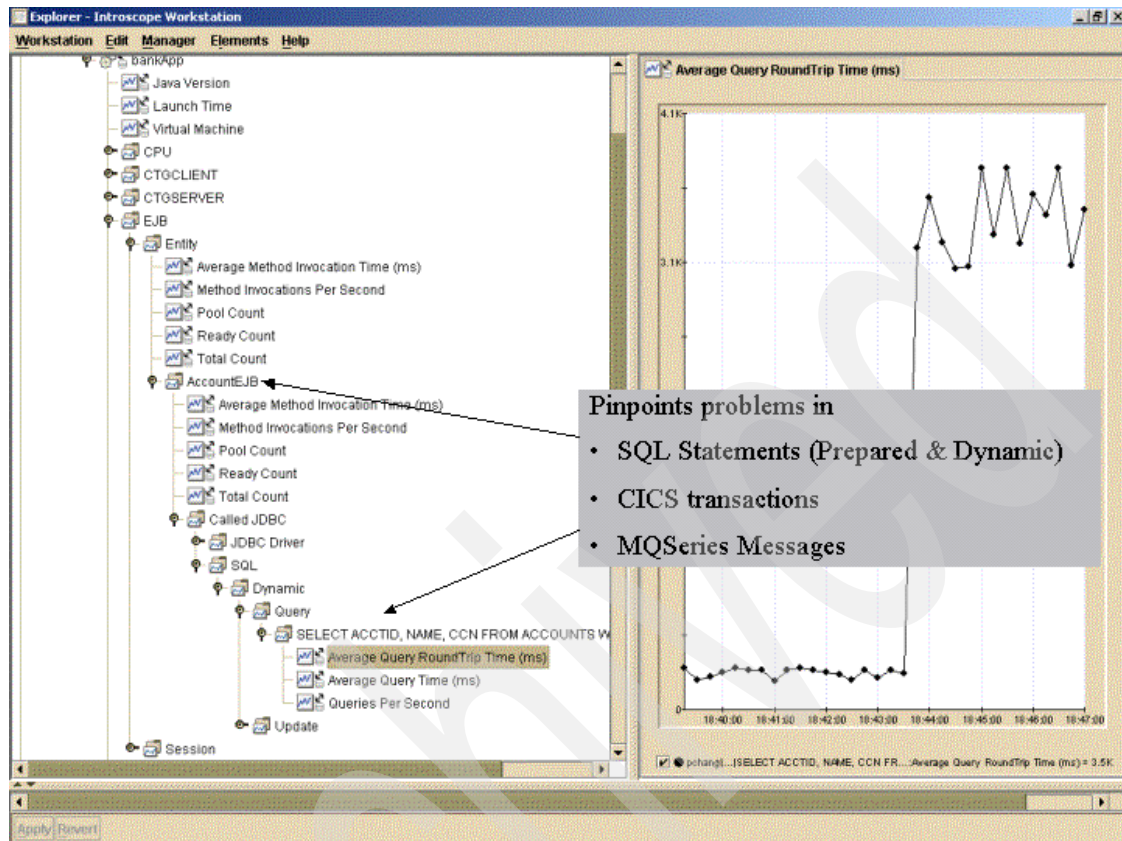


Figure 3-7 Introscope pinpoints bottlenecks

Summary of features

Table 3-2 Introscope features

Features	Actions
Component-level monitoring of live production Web applications	<p>Introscope can be used to proactively monitor the performance of the following:</p> <ul style="list-style-type: none"> EJB, Servlet, JSP, JTA, JMS, JNDI, XML, RMI, CORBA CPU Utilization, Memory, File Activity, Network Socket Traffic Classes, Methods, Java Threads Http, JDBC, SQL <p>It can pinpoint the cause of a performance bottleneck. Since Introscope agents create less than 5% overhead, they can be used to monitor live production Web applications.</p>

Features	Actions
Java component interaction monitoring	Tracks calls between components to identify the problematic component inside the application. Also helps to determine if the performance problem is inside or outside the application.
Flexible component monitoring	Provides Custom Tracing (methods, classes, transactions). Trace any desired method, class or data associated with a component for a more complete understanding of your application.
Transaction Management	Measure complete transaction times (such as Checkout) to understand exactly what your customers are experiencing.
Alert Notification	Sends alerts in the desired form (e-mail, page, on-screen) at a specified interval.
No development required	The Introscope ProbeBuilder operates on Java bytecodes. Programmers do not have to program any API into their code to make their application "monitorable" by Introscope.
Cluster support	Aggregate data at a clustered application level so you can determine where the bottleneck is in the cluster.
Data Payload Measurement	Monitor parameters passed between methods in applications and watch customer interactions (closure rate, approval rate, performance vs. Sales Close, etc.).
Quick real-time performance view	The Explorer Preview pane provides a quick real-time view of any metric.
Easy-to-use and configure console simplifies the monitoring task	Introscope's workstation console boasts the following convenience features: drag and drop, multiple views, and customizable Dashboards. Users can select the desired metric or groups of metrics within the Explorer, simply drag it to the desired Dashboard location and customize the view to suit their preference.
Integration with system management frameworks	Introscope enables the integration of both performance-triggered alerts and live application data into System Management frameworks such as Tivoli so operators can monitor Java applications from their System Management consoles.
Historical performance monitoring and trend analysis	Store data in database, extracting reports and other information on an As- Needed Basis. See how your system operated last night, yesterday, or even last month. Use 3rd party reporting software to group statistics and metrics in meaningful ways.

Features	Actions
Management of third-party vendor Java components	Introscope can monitor third-party vendor Java components just like components inside Web applications running on WebSphere application servers. This helps customers see inside third-party applications and resolve any related performance issues.
Integration WebSphere application servers	With AutoProbe, customers can start monitoring WebSphere applications in about 10 minutes.
Full application lifecycle support	Introscope brings value in development, QA, and production. Introscope allows different views of data for different users. It is an easy way to transfer knowledge between each group in the development cycle, and from employee to employee within a team.

3.4 JInsight

Important: Many of JInsight's features are now being made available as part of the WebSphere Studio Application Developer profiling tool. Check the Application Developer documentation for the latest features of that profiling tool.

Note that not all JDK levels support all functionality of JInsight.

JInsight is a prototype technology and so there is no support from IBM.

JInsight is a tool for visualizing and analyzing the execution of Java programs. It is useful for performance analysis, memory leak diagnosis, debugging, or any task in which you need to better understand what your Java program is really doing.

JInsight brings together a range of techniques that let you explore many aspects of your program:

- ▶ **Visualization**
Visualizations let you understand object usage and garbage collection, and the sequence of activity in each thread, all from an object-oriented perspective.
- ▶ **Patterns**
Pattern visualizations extract the essential structure in repetitive calling sequences and complex data structures, letting you analyze large amounts of information in a concise form.
- ▶ **Information exploration**
You may specify filtering criteria to focus your study, or drill down from one view to another to explore details. Create your own units that precisely match features you are studying, and then use them as an additional dimension in many of the views.
- ▶ **Measurement**
Study measurements of execution activity or memory summarized at any level of detail, along call paths, and along two dimensions simultaneously.
- ▶ **Memory leak diagnosis**
Special features are provided to help you diagnose memory leaks.

Attention: We used JInsight as a profiling tool for this redbook. During the editing process, JInsight profiling functions were integrated into WebSphere Studio Application Developer. Please check the latest release of that tool for application profiling.

Configure your J2EE server

To install JInsight, use the System Management User Interface and modify the configuration of your J2EE server. To do so, add or alter the following environment variables:

```
LIBPATH=/usr/lpp/jinsight/jinsight2.1
JVM_EXTRA_OPTIONS=-XrunjinsightPA
JINSIGHT_TRACE_FILENAME=/u/java1/Tracefilename.trc
```

Then check the Debugger Allowed field. The WebSphere for z/OS runtime limits certain real-time debugging capabilities in production servers in production mode. You have to enable the full functions of debugging.

You are now done with the J2EE server configuration.

Configure and install the JInsight application

To monitor the traces in your server, JInsight uses an application driven by one servlet and an HTML page. You need to configure and install this application. The steps are the generation of a war file and then an ear file. To create a war file for the JInsight application, you can use WebSphere Studio. Just modify jinsight.htm and check that your deployment descriptor looks like Example 3-1.

Example 3-1 Deployment descriptor of JInsight

```
<?xml version="1.0"?>

<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
  <display-name>Jinsight2.1-WebApplication</display-name>
  <servlet>
    <servlet-name>jintrace</servlet-name>
    <description>THE Jinsight servlet</description>
    <servlet-class>com.ibm.jinsight.tracing.TraceControlServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>jintrace</servlet-name>
    <url-pattern>/jintrace</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>jintrace.htm</welcome-file>
```

```

</welcome-file-list>
<login-config>
  <auth-method>BASIC</auth-method>
</login-config>
</web-app>

```

After you deploy the JInsight application and configure your J2EE server, *do not forget* to add a service directive in your httpd.conf file corresponding to the context root URI you have chosen in the Application Assembly Tool.

Start your server and monitor

You are ready to start your server. Monitor the startup messages as shown in Figure 3-8.

```

*****
Jinsight2.1, build 20010629
Jinsight Profiler is Licensed Materials - Property of IBM
Copyright (c) IBM Corp. 2000
IBM is a Trademark of International Business Machines

*****
Authors:  Jeaha Yang      Fereydoun Maali
*****

*****
OS/390 : use CTRL-V for tracing options

*****
Number of segments used will be  32
Size of a segment used will be  524288
Total buffer size used will be  16777216
parser moduleTypes:
-- listing properties --
Online21plus.war=2
vaprt.jar=3
WebShop21z0S.jar=3
APPLICATIONNAME=EJBShop

```

Figure 3-8 JInsight messages at server startup time

You can use JInsight to start the tracing of your server:

```
http://YourHostName:port/YourContextRoot/jintrace.htm
```

You can run any application deployed in this server and then stop the traces. JInsight generates a file in the directory.

What is measured

Once you have launched JInsight on your workstation, click **Workspace Window** -> **Load**.

Wait until the end of the trace is reached (that is, when all trace events have been read, as indicated by the Workspace Window).

When you get the message, choose **Workspace Window** -> **Views** -> **Histogram** -> **Objects**.

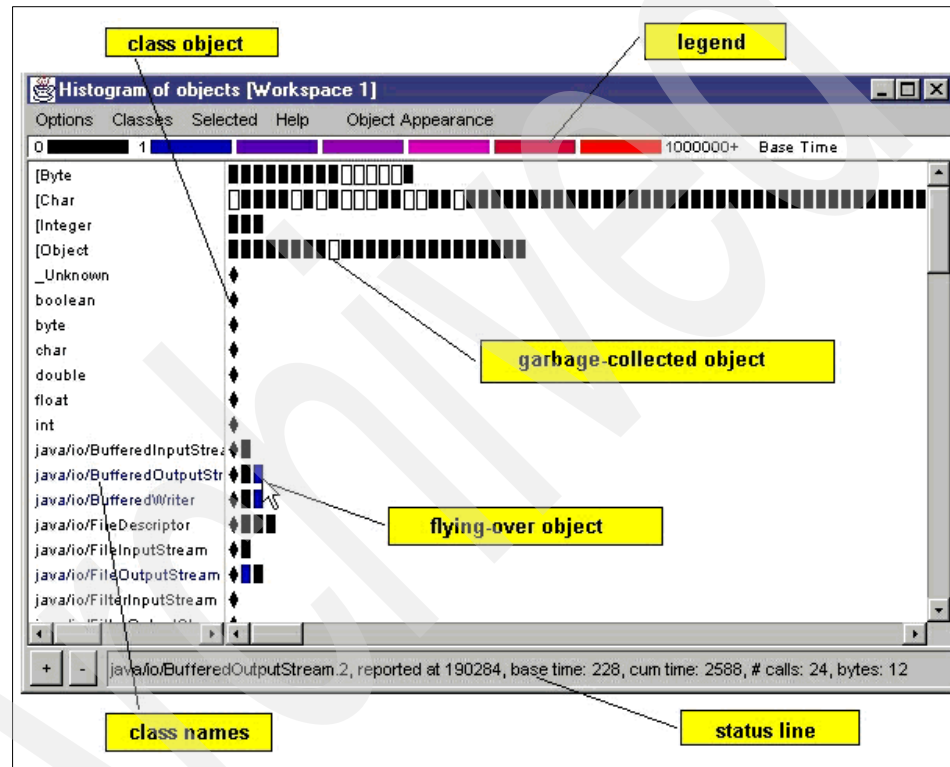


Figure 3-9 JInsight object histogram viewer

The names of classes used in the program appear along the left of the view. Objects appear to the right of each class name. The first, a colored diamond, represents the class object. To the right of the class object are colored rectangles, one for each instance of the class.

JInsight allows you to check for the time spent in each of your methods. You can select a method to trace and see the details. Choose **Workspace Window** -> **Views** -> **Table** -> **Methods**. (See Figure 3-10 on page 48.)

class name	method name	cumulative time Workspace
java/util/Random	Random ()	79
java/util/Random	Random (long)	43
java/util/Random	setSeed (long)	11
java/util/Random	next (int)	288
java/util/Random	nextBytes ([byte)	0
java/util/Random	nextInt ()	0
java/util/Random	nextLong ()	0
java/util/Random	nextFloat ()	0
java/util/Random	nextDouble ()	638
java/util/Random	nextGaussian ()	0
java/util/Random	<clinit> ()	8

Figure 3-10 JInsight Methods view

A Call Tree view (Figure 3-11) shows the outline of calls made by this particular invocation of `println(String)`. The time contributions of each call from `println(String)` are shown.

Call Tree: Calls from java/io/PrintStream.println (String): 1 occurrences [Workspace 1]			
	contribution %	contribution	number of calls
println	100.0%	2320	1
├─ print	82.6%	1917	1
│ └─ write	82.0%	1903	1
│ │ └─ write	57.5%	1335	1
│ │ └─ flushBuffer	12.8%	297	1
│ │ └─ flushBuffer	7.1%	165	1
│ │ └─ indexOf	1.4%	33	1
│ │ └─ ensureOpen	0.3%	7	1
└─ newLine	16.2%	377	1
├─ flushBuffer	7.5%	173	1
├─ newLine	3.1%	71	1
├─ flushBuffer	2.8%	65	1
├─ flush	0.9%	20	1
└─ ensureOpen	0.2%	4	1

Figure 3-11 JInsight Call Tree view

The detailed use of JInsightYou is discussed in 5.1, “The sample application: OnlineBuying” on page 118.

3.5 JProbe

Overview

JProbe is a set of graphical tools for diagnosing and eliminating code errors and inefficiencies in Java applications. JProbe helps programmers understand potential problems in the application. JProbe provides, in graphical form, information concerning various topics from memory usage to calling relationships.

Additionally, JProbe makes it possible to analyze interactions with Java modules outside the application being analyzed, even if source code for those modules is unavailable. However, if the source code is available, JProbe allows you to pinpoint the line of code that is causing the problem.

JProbe identifies problems caused by interactions with third-party components, queries to a relational database via Java database connectivity (JDBC), and calls to other distributed elements of the application through remote method invocation (RMI).

While it isn't possible to modify parts of an application without source code, understanding where performance problems lie is essential to solving the problem. This capability allows programmers to design solutions without potential problems. For example, if a particular call to a third-party Java module is expected to become a key bottleneck, the programmer may write the application to implement caching algorithms to decrease the overhead time spent in these calls.

The JProbe product line includes four diagnostic tools:

- ▶ JProbe Profiler with integrated Memory Debugger
Profiler identifies where time is being spent inside an application, helping developers increase the efficiency and scalability of their code. Memory Debugger tracks down memory leaks and memory mismanagement, both of which can seriously hinder scalability
- ▶ JProbe Threadalyzer
Tracks down threading issues, such as deadlock and data race situations.
- ▶ JProbe Coverage
Ensures that test cases are complete by identifying which parts of an application have been executed during testing.

Environment

The JProbe environment in a remote session is shown in Figure 3-12 on page 50.

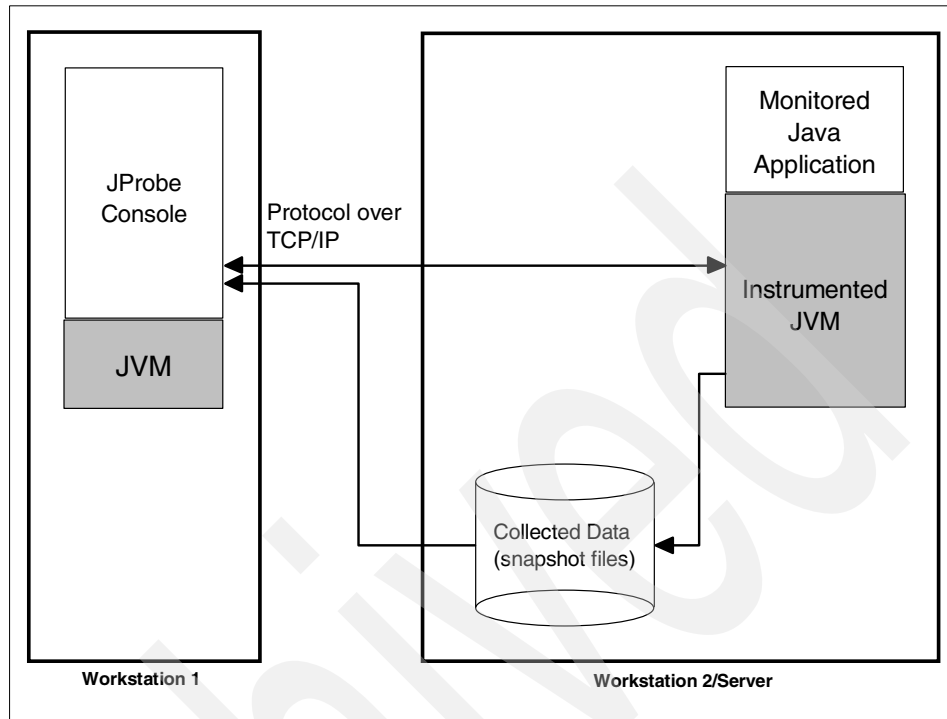


Figure 3-12 JProbe environment in a remote session

Starting a JProbe product brings up the JProbe Console. The JProbe Console, a Java application itself, executes in a separate JVM from the program it monitors. Your Java application runs in a different, instrumented JVM. An instrumented JVM is a JVM with a profiling interface. Your Java 2 JVM must fully support Sun Microsystems's JVM profiling interface, JVMPi. Different JVMs are used to ensure that the collected data is specific to your Java application.

Collected data is stored in snapshot files, saved in a temporary directory specified by the user. You can choose to rename and relocate the snapshots to a permanent location. Additionally, you can create snapshots via either an interactive session or setting triggers to turn data collection on/off automatically.

Configure your J2EE server and JProbe

JProbe can analyze Java applications running under WebSphere Application Server V4. However, activation of JProbe requires certain changes to the WebSphere current.env file. The following variables need to be added or modified in this file. The JProbe directories should be appended to the current environment variables.

Example 3-2 Environment variables to be added for JProbe

```
JVM_BOOTCLASSPATH=/JProbe30/bin/jpagent.jar
JVM_EXTRA_OPTIONS=-Xrunjprobeagent:-jp_input=/web/server4/was40remote.jpl
JVM_ENABLE_CLASS_GC=no
JAVA_COMPILER=
PATH=${PATH}:/JProbe30/bin
LD_LIBRARY_PATH=/JProbe30/bin
LIBPATH=${LIBPATH}:/JProbe30/bin
CLASSPATH=${CLASSPATH}:/JProbe30/bin
```

Then you need to check the Debugger Allowed field. The WebSphere for z/OS runtime limits certain real-time debugging capabilities in production servers in production mode. You have to enable the full functions of debugging.

To monitor applications under WebSphere 4.0 from a workstation, set up a remote session. The jpl file specified in JVM_EXTRA_OPTIONS serves as the XML-based configuration file for JProbe. In this file, you need to define the remote client viewer that connects to the server by specifying the workstation's name and port number. The default port used is 4444. A sample of a jpl file may look as follows:

Example 3-3 JPL configuration file for JProbe

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jpl SYSTEM "jpl.dtd" >

<jpl version="1.5">
  <program type="application">
    ...application parameters
  </program>

  <vm
    args=""
    location="/usr/lpp/java/IBM/J1.3/bin/java"
    snapshot_dir="."
    type="java2"
    snapshot_prefix="WAS40"
    use_jit="false"/>
  <viewer socket="machine1:4444" type="remote"/>

  <analysis type="profile">
    ...analysis parameters
  </analysis>
</jpl>
```

Monitoring your application

JProbe Profiler

JProbe Profiler is capable of measuring the following performance metrics:

Number of Calls	Number of times the method was invoked.
Method Time	Time spent executing the method, <i>excluding</i> time spent in its descendants.
Cumulative Time	Total amount of time spent executing the method, <i>including</i> time spent in its descendants but <i>excluding</i> time spent in recursive calls to descendants.
Method Object Count	Number of objects created during the method's execution, <i>excluding</i> those created by its descendants.
Cumulative Object Count	Total number of objects created during the method's execution, <i>including</i> those created by its descendants.
Average Method Time	Method Time divided by Number of Calls.
Average Cumulative Time	Cumulative Time divided by Number of Calls.
Average Method Object Count	Method Object Count divided by Number of Calls.
Average Cumulative Object Count	Cumulative Object Count divided by Number of Calls.

Clicking **Call Graph** brings you to the Call Graph screen shown in Figure 3-13 on page 53.

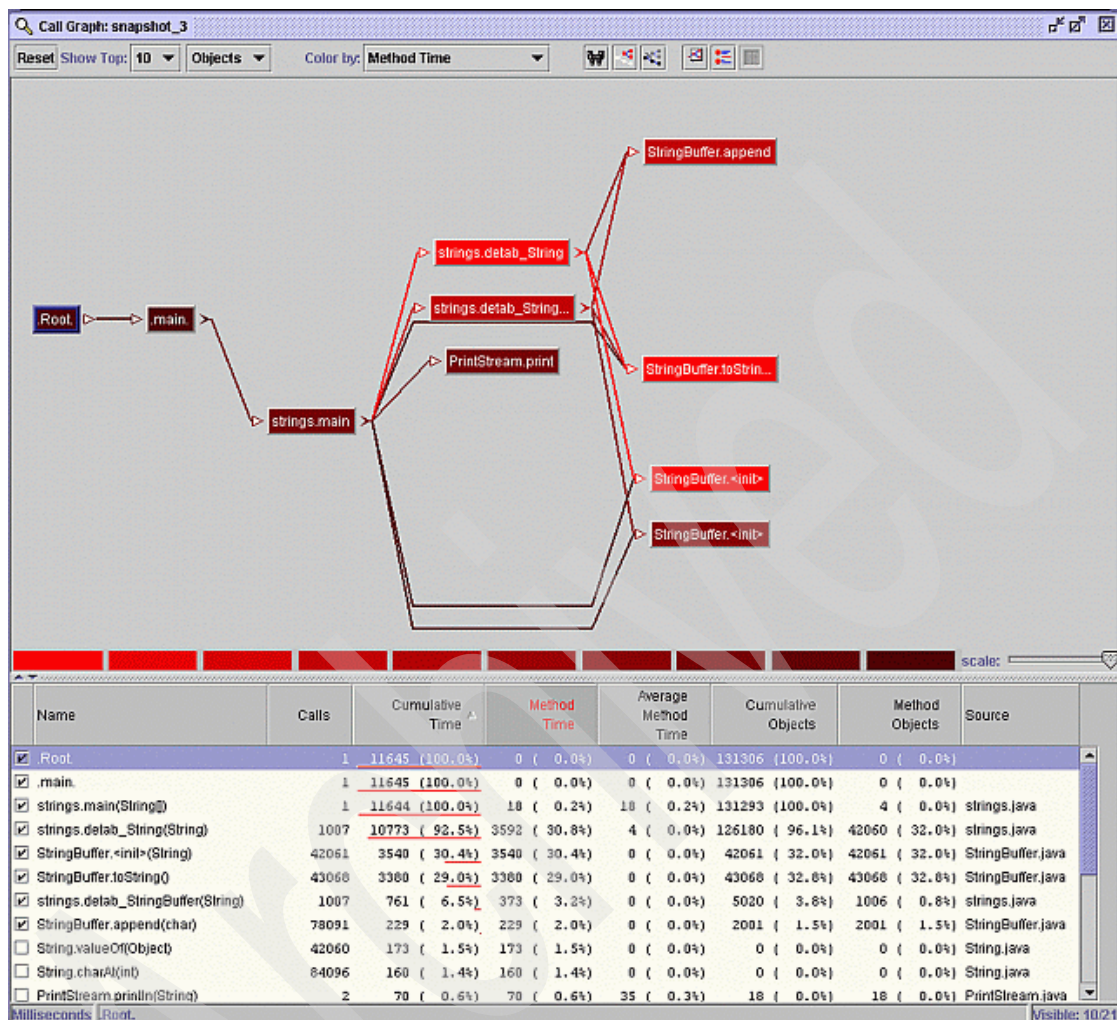


Figure 3-13 Method call view with JProbe

Figure 3-13 diagrams the flow of the method calls in the upper half of the screen. The lower half of the screen lists the methods with their corresponding data, such as number of calls, cumulative and method execution times, and cumulative and method object counts. You can click the method name in the upper part of the screen or in the list in the lower part of the screen. This highlights the method for improved readability. Additionally, you can set a specific method as the root to concentrate on code from that part of the program.

Double-clicking on a specific method brings you to the Method Detail screen shown in Figure 3-14.

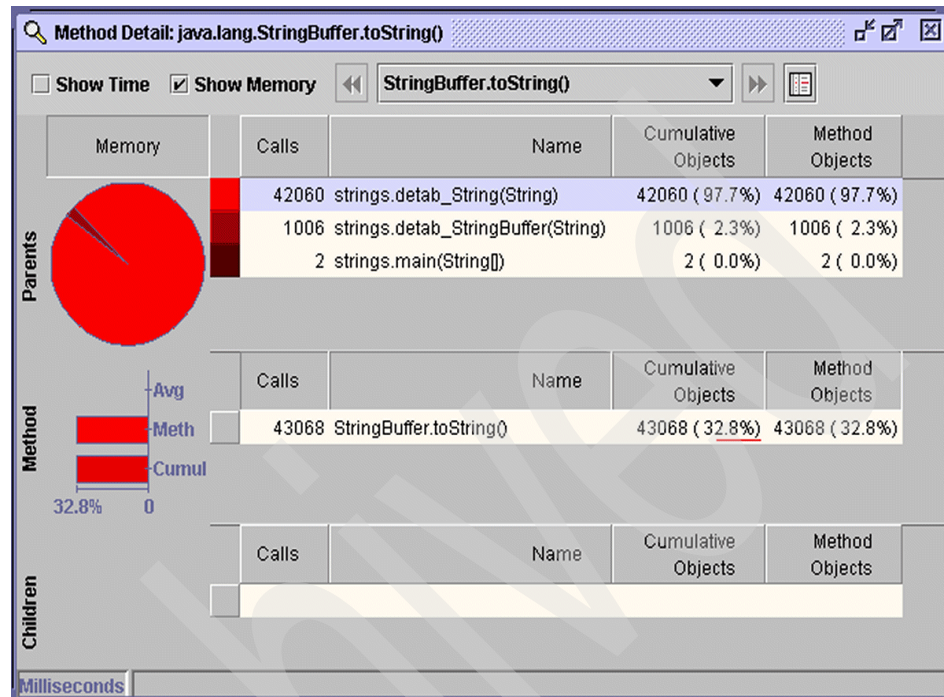


Figure 3-14 Viewing method details with JProbe

This screen shows the same performance metrics in pie chart and tabular formats. However, it limits the methods displayed to the ones called from the method that you double-clicked. You can choose to display metrics for either time or objects, or both.

JProbe Memory Debugger allows heap analysis for classes, instances and allocation points.

Class-level metrics

The Memory Debugger collects the following metrics for classes:

- Count** Current number of instances of the class in the Java heap.
- Cumulative Count** Total number of instances allocated in the heap since the beginning of the session or the last clear data action.
- Memory** Amount of memory consumed by all the instances of the class currently in the Java heap.

Cumulative Memory Total memory consumed by the instances allocated in the heap since the beginning of the session or the last clear data action.

Setting checkpoints allows you to collect the following metrics on classes:

Count Change Difference between the number of objects in the heap when a checkpoint was set, and the objects currently in the heap.

Memory Change Difference between the memory used when the checkpoint was set, and the memory currently in use.

Instance-level metrics

The Memory Debugger collects the following metrics for instances:

Size Amount of memory consumed by this instance of the class.

Creation Time Point in time when the instance was allocated.

of Referrers Number of referrers to the selected instance, and the instance IDs and field names for the referrers.

of References Number of references from the selected instance, and the instance IDs and field names for the references.

Allocation point metrics

You can trace constructor invocations and collect data on the method calls that led to the allocating method. You collect stack traces to the depth you specified, plus links to the source code. The Memory Debugger collects the following metrics for allocation points:

Method Name Name of the method that allocated instances of the selected class (or that called a method in the series of calls leading up to the allocation). The variable/field name appears in brackets.

Source The class and the line of code that allocated the selected instance. This requires the source code to be available.

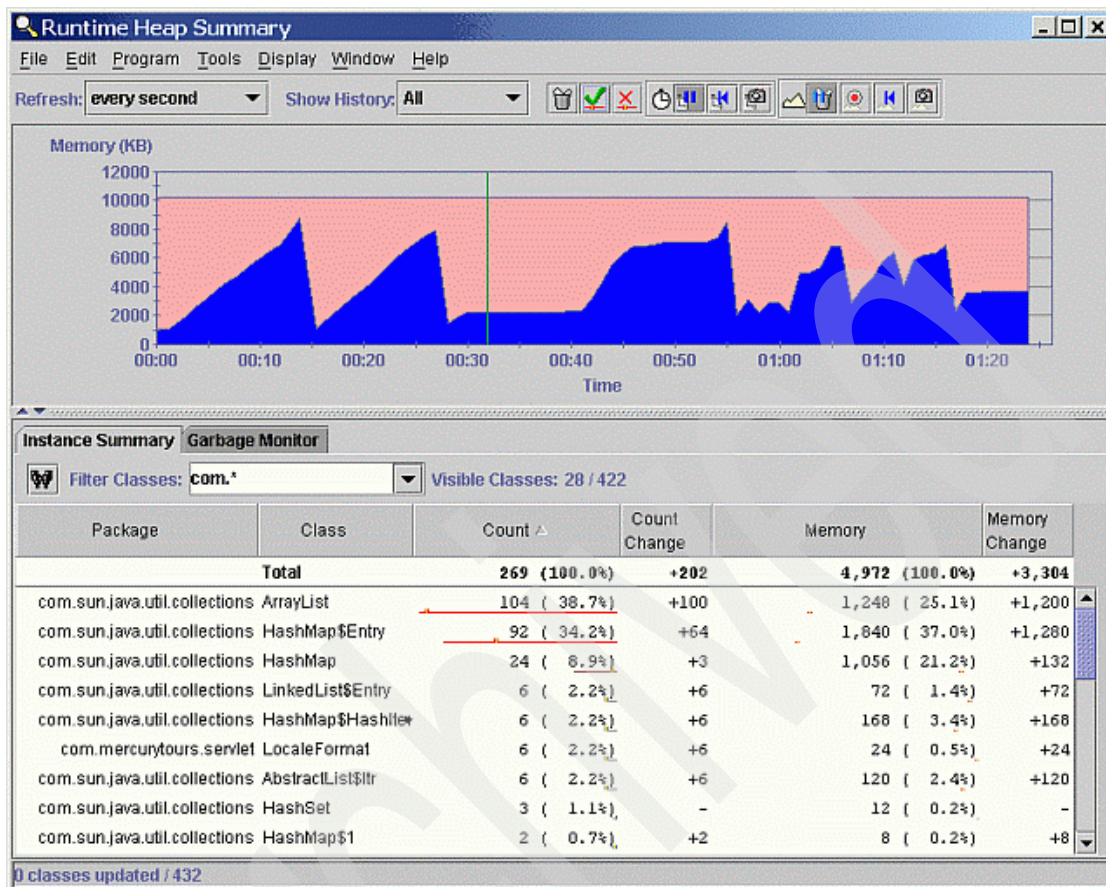


Figure 3-15 JProbe heap usage chart

Figure 3-15 displays the Heap Usage Chart in the upper part. It shows the size of the Java heap in pink and the amount of memory in use in blue. The lower part of the screen displays tables containing information on instances in the heap or garbage collected objects. The Instance Summary tab summarizes the instances allocated for each class, including the number of instances and memory used by those instances. The Garbage Monitor tab displays the top ten classes contributing to garbage collections. Stack traces and garbage collected objects must be included in the JProbe configuration to get this information.

Programming techniques

This chapter provides two different kinds of information. It gives an overview of design patterns that can be used to better design, architect and code Java applications. It then introduces Java programming techniques for optimization, issues about JDBC versus SQLJ, and optimization topics about servlets, JSPs and EJBs.

4.1 Applying design patterns

How to design efficient object-oriented software? How to solve some common design problems? And how to be sure the final application will perform well?

In the following section, we describe some useful patterns about the performance aspect and some best practices for the design phases. The role of these patterns is to develop solutions that are more reusable, flexible and efficient. For each pattern we give a simple description, a figure of its structure, and the value of using such a pattern.

This section is based on the book *Design Patterns: Elements of Reusable Object-Oriented Software*, ISBN 0-201-63361-2.

You may also check the following Web sites, which contain descriptions and UML diagrams of the referenced patterns:

<http://www.theserverside.com/patterns/index.jsp>
<http://www.precisejava.com/javaperf/j2ee/Patterns.htm>
<http://pages.cpsc.ucalgary.ca/~kremer/patterns/>
http://java.sun.com/blueprints/patterns/j2ee_patterns/index.html
<http://exciton.cs.oberlin.edu/JavaResources/DesignPatterns/default.htm>
<http://www.dofactory.com/patterns/patterns.asp#list>
http://www.mindspring.com/~mgrand/pattern_synopses.htm#synopses
<http://www.enteract.com/~bradapp/docs/patterns-intro.html>
<http://hillside.net/patterns/>

The design of a pattern can be *creational*, *structural* or *behavioral*. A creational pattern is used to generate other objects from a well-defined one. Structural patterns provide better communication among objects and avoid having too many interconnections. Behavioral patterns can be used to define or optimize the role of objects and their respective relations.

4.1.1 Creational patterns

As we are focusing on performance, we discuss the *prototype*, the *abstract factory* and the *singleton* patterns. These are the most common and useful creational patterns related to performance.

Prototype pattern

The role of this pattern is to generate an object by making a copy of an already existing object called a “prototype”. The copy of the prototype is done at object creation.

Structure

See the referenced book and Web sites for the UML diagram of the pattern.

When to use

- ▶ Instead of instantiating a class, create a copy of a prototype.
- ▶ You do not want to have a duplicated hierarchy of objects (objects and factories).
- ▶ You need independence from the system in the creation and representation of the objects.
- ▶ You need to do some dynamic class loading.

Benefits

One of the main benefits of this pattern is that you can add or remove a prototype object at runtime. This makes this one of the most flexible patterns. Your client only needs to have registered the new class instance.

The dynamic advantage of this pattern is the control through parameters set in the prototype for the behavior of your cloned objects. You do not have to code a new class, you only need to make a copy of the prototype. This reduces the number of classes. The Java Virtual Machine therefore has fewer objects to manage in memory, which is a plus for performance.

Another benefit is the reduced subclassing and the fact that you do not have to create parallel hierarchies of creators and objects.

Abstract factory pattern

This pattern allows you to create objects without specifying their concrete classes. It can be used to generate similar objects, for instance a Concrete Business Class and a Dummy Business Class, to test different scenarios.

Structure

See the referenced book and Web sites for the UML diagram of the pattern.

When to use

- ▶ You need independence from the system in the creation and representation of the objects.
- ▶ You have to build many families of products (classes categories).
- ▶ You want to force the use of a collection of objects or a group of complementary classes.

Benefits

Because you can hide the implementation, the client code does not have to use the concrete class, so it is simpler to manipulate the object.

You can change an easy way the family of objects used by changing the concrete factory. As its instantiation is done just once in the code, you just need to modify this class. The pattern is also a guarantee that your application will use only one set of classes at a time.

But be aware that if you need to add a new category of products to your application, you need to extend the factory interface, which implies changes in the `AbstractFactory` and, of course, in all its subclasses.

This pattern is illustrated in Example 4-1, which shows the declaration of the interface.

Example 4-1 Declaration of the `ActionBusiness` interface

```
public interface ActionBusiness {
    public void checkUserCode(String code) throws UserException;
    public void initForm(User user, DeclarationFormBean dfb) throws UserException;
    public void store(User user, FormDetails fd) throws UserException;
    public void verifyFormNumber(FormNumberInput input, FormNumberOutput output)
        throws UserException;
    public void verifyUser(User user) throws UserException;}

```

We define a class factory using the `AbstractFactory`:

Example 4-2 Defining a class factory

```
public class ActionBusinessFactory {
    private static ActionBusiness actionBusiness = new
    ActionBusinessConcrete();
    public static ActionBusiness createUserBusiness() {
        return actionBusiness;}}

```

Now we define the `Products` classes, first a dummy implementation, then a concrete one.

Example 4-3 The dummy `ActionBusiness` class

```
public class ActionBusinessDummy implements ActionBusiness {
    public ActionBusinessDummy() {super();}
    public void checkUserCode(String code) throws UserException {
        .....dummy user code...}
    public void initForm(User user, DeclarationFormBean dfb) throws UserException{
        System.out.println("ActionBusinessInterface.initForm()");
        .....dummy user code....}
    public void store(User user, FormDetails fd) throws UserException {

```



```

        System.out.println("ActionBusinessInterface.store()");
    ....dummy user code...}
    public void verifyUserNumber(UserNumberInput input, UserNumberOutput output)
    throws UserException {....dummy user code...}
    public void verifyUser(User user) throws UserException {
        System.out.println("UserBusinessInterface.verifyUser()");
    ....dummy user code...}
    } /* End of ActionBusinessDummy */

```

Then you can create a concrete class with a real implementation, in the same way the dummy class was created:

Example 4-4 The concrete ActionBusiness class

```

public class ActionBusinessConcrete implements ActionBusiness {
    public ActionBusinessConcrete() {
        .....concrete user code....}
    public void checkUserCode(String code) throws UserException {
        .....concrete user code.....}
    public void initForm(User user, DeclarationFormBean dfb) throws UserException
    .....concrete user code.....}
        public void store(User user, FormDetails fd) throws UserException {
            .....concrete user code.....}
    public void verifyUserNumber(UserNumberInput input, UserNumberOutput output)
    throws UserException {
        .....concrete user code.....}
    public void verifyUser(User user) throws UserException {
        .....concrete user code.....}
    } /* End of ActionConcreteClass */

```

Singleton pattern

The Singleton pattern controls the creation of objects. It allows the creation of only one instance. Use the Singleton method to access the instance. This method is the main entry point to use your objects.

Structure

See the referenced book and Web sites for the UML diagram of the pattern.

When to use

- ▶ You need to have just one single entry point to access a class.
- ▶ You want to control the access to your instances.
- ▶ You can extend a single instance for client use without implying any changes in its code.

Benefits

This pattern gives you total control of the instance. You can take any actions before allowing the client to use the instance. For example, you can control the client identity or check for the state of a transaction, etc.

There is no need to define your own variables to store the current instance of the object. This means avoiding too many global names in your Java Virtual Machine. Of course, this pattern allows you to control a single instance, but you can adapt the code to control any number of instances desired.

Example 4-5 illustrates this pattern.

Example 4-5 Singleton pattern

```
public static DB2Class singleton() {  
    if (instance == null) instance = new DB2Class();  
    return instance;}  

```

If your application is multithreaded, you can have two concurrent threads trying to use this Singleton method. To be sure that only one instance is generated, you can use the following declaration:

```
public synchronized static DB2Class singleton() {...}.
```

This ensures that the first thread will complete the creation of the instance, and the second thread will use the already generated instance.

Important: In the scalable server environment of WebSphere for z/OS, customers need to understand that they may not get a single instance across the entire server instance. That is not to imply that the usage pattern is different, merely that there may be additional concerns.

For example, it may not be wise to use a singleton to maintain a count of the number of times a process is executed, because the count may only reflect the number of times the process executed in a particular class loader scope and/or JVM, not the total number of executions. For most applications this would not be a problem. This could, however, become a problem if the application uses this to generate a "unique" ID for a request or operation.

4.1.2 Structural patterns

Among the structural patterns, the *proxy*, *adapter*, *flyweight* and *facade* patterns can be used to improve performance. This is due to their capacity to define a structure that allows better use of the objects.

Proxy pattern

This pattern allows you to control access to another object. Most of the time, this is a remote object and you want to defer the time of creation and initialization until you really need to use the object.

Structure

See the referenced book and Web sites for the UML diagram of the pattern.

When to use

- ▶ You want to have advanced control over objects and their references.
- ▶ You need to control the access to an original object (use of Protection Proxy).
- ▶ You want to do some usually time-consuming operations (loading of images, etc.) in the fastest possible way.

Benefits

Depending on the proxy you use, you can improve the efficiency and performance of your application. For instance, if you choose a remote proxy, you can simulate a local object, but the real one is located at a remote address.

You also have the option of using a virtual proxy to create an object of a client request. This is an optimization, because the proxy can have control of the created object. The system usually makes some copy of large objects even if an object has not been modified. A virtual proxy can only copy the modified objects, saving CPU costs.

This option is called copy-on-write. It can improve the performance of your application by avoiding the copy of heavy objects if this is not needed.

Example 4-6 illustrates this pattern.

Example 4-6 Proxy pattern

```
public interface CustomerInfo {
    public CustomerInfo();
    public void loadInfo(String info);
    public void saveInfo(String info);
    public Info getKey();
}

public class Identity implements CustomerInfo {
    public Identity(int idnumber) { }
    public void loadInfo(String info) {...user code....}
    public void saveInfo(String info){...user code...}
    public Info getKey() {...user code...}
}

public class IdentityProxy implements CustomerInfo {
    public Info customerData;
```

```

        public int anidnumber;
        public Info aKeyInfo;
    public IdentityProxy(int idnumber) {
        anidnumber=store(idnumber);
        aKeyInfo=null;
        customerData=null;
    }
    public void loadInfo(Info theReturnedKey) {...load the customer info....}
    public void saveInfo(Info theReturnedKey){...save the customer info+...}
    public Info getId() {
        if (customerData == null) {
            customerData = new Identity(anidnunmber);
        }
        return customerData;}
    public Info getKey() {
        if (aKeyInfo==null) {aKeyInfo=this.getId();}
        return aKeyInfo}
    }

```

Adapter pattern

As its name suggests, this pattern allows you to adapt an interface to another one. It introduces more flexibility in your application by letting you use classes whose interfaces are not compatible.

Structure

See the referenced book and Web sites for the UML diagram of the pattern.

When to use

- ▶ You want to access a specific class but its interface does not match your needs.
- ▶ You need to create a reusable object to access new classes added to your application.
- ▶ If you do not want to subclass too many existing classes, you can adapt the superclass interface of the adapter object.

Benefits

An adapter (sometimes also called a *wrapper*) is a versatile tool for connecting an already existing reusable class into a new context with minimal expectations from the reused class. This can be very helpful for classes that implement some algorithm with near optimum runtime behaviour, or any other piece of software with incompatible interfaces. The adapter code consists mostly of short and simple methods dispatching external calls from client to operations provided by the reused code.

Because of the versatility, the amount of mostly non-reusable adapter code can become substantial. There is always a cost in reusing any existing class, and if the class is very different from the expectations of the environment, it can be more beneficial to rewrite the needed part of the "reusable" class so that it fits into the new context better.

Example 4-7 on page 65 illustrates this pattern with an example of a class adapter.

Example 4-7 Example of a class adapter

```
public interface ExampleA {
    public methodA1();
    public void methodA2(String testA);
}

public interface ExampleB {
    public methodB1();
    public void methodB2(String testB);
}

public class AdapterAB implements A, implements B {
    public void methodA1() {...code here can adapt methodA2...};
    public void methodA2(String testA);
    public void methodB1(..code here can adapt methodB2...);
    public void methodB2(String testB);
}
```

Flyweight pattern

If you need to have several small objects, find a solution to share their use to save your system resources. This pattern is able to hold many objects with a fine level of granularity.

Structure

See the referenced book and Web sites for the UML diagram of the pattern.

When to use

Because this pattern has a very specific use, all of the following conditions should exist:

- ▶ Your application needs a lot of small objects.
- ▶ You need to store a lot of information for each of these objects.
- ▶ Most of your object states can be extrinsic, that is, the state can be shared among instances of your object. This consumes fewer resources than an intrinsic object, which needs to keep state information for each object instance.
- ▶ You have no need to use the identity of your objects.

Benefits

The flyweight pattern has a performance benefit due to the reduction of space through the use of shared resources. When the state of your object is intrinsic, it requires more resources. The key with this pattern is to have a large number of objects with an extrinsic state computed instead of being stored.

Facade pattern

Because the structural design of your application is important, you need to be aware of this pattern. Once your application has many interfaces with many objects, it can be difficult to use it. The facade pattern allows you to group objects under one common interface.

Structure

See the referenced book and Web sites for the UML diagram of the pattern.

When to use

- ▶ Your application becomes complex and you need to introduce a level of abstraction to facilitate its access.
- ▶ You want to introduce independence between the client subsystem and the other ones.
- ▶ You need to improve the portability and the use of your application through several entry points (facade objects). Each corresponds to a layer or a subsystem.

Benefits

This pattern is a structural one, to organize your application. You can hide some parts of the one behind a facade interface. A facade interface is an object that adds a layer between the service implementation and the client. Generally, a facade object is designed to hide complexity. The client just accesses the high-level subsystem.

Another benefit of this pattern is in the independent structures you can generate by avoiding the dependencies of the classical approach. Your application is more flexible from the client's view, and the client still has the option to directly use the different subclasses.

4.1.3 Behavioral patterns

This layer deals with how the objects are interacting. The *state*, *mediator* and *command* patterns can be linked with the performance aspect by their ability to impose a specific behavior on other components.

State pattern

This pattern has a trigger on its state value. If its state changes, it can modify its behavior dynamically. It allows fine control depending on the object state by dispatching the execution to specific objects.

Structure

See the referenced book and Web sites for the UML diagram of the pattern.

When to use

- ▶ You have an application which can be controlled by the value of an object state.
- ▶ You have a lot of conditional statements in your code based on a value set by constant fields.
- ▶ You still need independence among your objects if the state of your application changes.

Benefits

The first option is that you can define a new state to your application without impacting its current behavior. Maintenance is easier because you do not have to modify the existing code. If you had coded your object with context object then some modifications would have been required.

For better performance, we do not recommend that you have a lot of objects but if you need so, it is better to have several small objects than bigger ones. You can notice this pattern represents your application's state by an object. That allows you to manage easily the action depending on the object involved.

Another advantage is the use of flyweight patterns because your state is no more intrinsic and you can share some objects. See "Flyweight pattern" on page 65.

Mediator pattern

Define an encapsulation for a given set of objects that need to interact. All the references from one object to another are stored by the mediator. It allows you to change their behavior independently.

Structure

See the referenced book and Web sites for the UML diagram of the pattern.

When to use

- ▶ You need to customize the behavior of your objects.
- ▶ You encounter some problems to reuse an existing class due to its interconnections with others classes.

- ▶ You want to reorganize the communication between your objects to render less complex links between them.

Benefits

The mediator pattern offers benefits because it avoids too much subclassing. Separate the mediator from the participants and replace your N-to-N (many-to-many) model relationship by a 1-to-N (1-to-many) relationship.

This implies easier communication and better understanding of the behavior of the application. The mediator is a kind of central point for as many objects as you want. Note: your mediator can become a complex object, difficult to modify. You have to control the complexity of the mediator objects unless you will not be able to maintain it for a long time.

Command pattern

This pattern allows you to have several requests within one unit of work. It translates the request to specific objects (the client does not even know about their existence) and gathers the responses to provide an answer to the client.

Structure

See the referenced book and Web sites for the UML diagram of the pattern.

When to use

- ▶ You need to group several actions into one call, for instance, to parameterize some objects.
- ▶ You want to control the request and wait for a while before executing the requested action.
- ▶ You look for a way to roll back your modifications depending on the result code.
- ▶ You need to save your information and reload these ones if any problem occurs during the process of running your application.
- ▶ You want to manipulate your objects as one transactional global unit of work.

Benefits

You have the possibility to access multiple subsystems and manage the result as one unique transaction. If no errors are encountered, you can commit your modifications or roll them back.

You can avoid too many remote calls and this can be a big performance improvement, especially if you are transferring data through large objects.

The command pattern introduces a level of abstraction between the client and the executive object. This means the command object is hiding the implementation of the service (connection, call, etc.) to the backend system, so you just need to use the methods provided to you by the command object. You do not have to code the services (connection, execution of a transaction, etc.) yourself. A command pattern can be extended and combined with other patterns. Changes to existing classes are not required to add some new commands.

4.2 Code optimization techniques

Refer to chapters 9 and 10 of *WebSphere for z/OS Operation and Administration*, SA22-7835 for performance measurement and tuning tips.

See the following Web site for Java performance issues on zSeries:

<http://www-1.ibm.com/servers/eserver/zseries/software/java/perform.html>

4.2.1 Java programming techniques

Garbage collection, object reuse

Java uses a garbage-collection mechanism that frees the programmer from worrying about deallocating memory after use. The Java Virtual Machine (JVM) runs the garbage collector regularly to free memory where possible. The garbage collector frees up only the memory no longer referenced by the application. Thus, although programmers do not need to worry about memory management, they do need to worry about memory retention. To avoid this problem, they should set references to null when no longer needed.

The garbage collector, a low priority thread, usually runs when the processor is free and no higher-priority threads are in queue. However, it will run immediately when the system needs more memory. Usually, this thread has about 5%-15% overhead, but how often garbage collection occurs depends on the application. When the JVM reaches the memory threshold, the garbage collector runs continuously until it frees up enough memory for the application to continue.

We recommend reusing objects whenever possible. Instead of creating and destroying objects continuously, programmers should try to maintain as few objects as possible and reuse them by reinitializing them with newer values. Although this focus on memory management negates the benefits of the garbage collection mechanism, it enhances performance. This requires some extra work on programmers' part but is compensated by an improvement in performance. Reusing objects not only reduces the overhead of creating objects, but also prevents allocated memory to build up to the point where the JVM spends most of its time running the garbage collector.

Example 4-8 shows two potential ways of doing the same work.

Example 4-8 Garbage collection

```
public void gc1()
{
    Date d1 = new Date(1, 20, 2002);
    Date d2 = new Date(10, 26, 2002);
    d1.printDate();
    d2.printDate();
}

public void gc2()
{
    Date d1 = new Date(1, 20, 2002);
    d1.printDate();

    d1.setMonth(10);
    d1.setDay(26);
    d1.setYear(2002);
    d1.printDate();
}
```

Method `gc1()` creates and initializes two `Date` objects, prints the dates, and finally deallocates the memory. Method `gc2()` enhances performance as it creates and initializes only one `Date` object, and prints the date. However, instead of deallocating the memory, it reuses the first object by setting the data members as needed. This way saves time by avoiding the creation and initialization of a new object, in addition to saving time later during the execution of the garbage collector thread.

Primitive data types vs. objects

Objects cost significant CPU time because there is much overhead associated with their creation. Each time an object is created, its constructor and the constructors of all its parent classes are called. Primitive data types are much faster than objects that encapsulate them. Additionally, primitive data types reduce memory requirements and decrease variable access times. Avoid the costs of object creation and manipulation by using primitives instead of objects where possible.

Example 4-9 shows two different ways to represent and work with the same data.

Example 4-9 Using primitive data types vs. objects

```
class Money
{
    double amount;
}
```

```
public void po1()
{
    for(int i=0; i<10000; i++)
    {
        Money m = new Money(i);
        // do something with the amount in Money
    }
}

public void po2()
{
    for(int i=0; i<10000; i++)
    {
        double amount = i;
        // do something with amount
    }
}
```

Method `po2()` performs much faster than method `po1()` by using a primitive data type instead of an object, thus avoiding the overhead of object creation and initialization. In our tests, `po2()` performed 10-20 times faster than `po1()`. Furthermore, method `po2()` also experiences better performance later in the application because it saves the garbage collector thread from working overtime.

Granularity

The issue of granularity has no right or wrong answer. Granularity varies with each establishment. However, one basic rule is to size an object only as big or small as necessary. You don't want an object to be so big and vague that it causes unnecessary overhead by executing or loading unnecessary code into memory. On the other hand, you also don't want an object to be so small and specific that more time is spent in calling the object's methods than in their execution.

Loop optimization

Loops repeatedly execute the same code. There exist multiple ways to optimize loops: loop constants, local variables, and early termination.

Loop constants

Loops continue to execute until a certain condition becomes false. Often, this condition involves the testing of a variable against another value, usually a constant. If this value remains constant during the execution of the loop, then it is expensive and unnecessary to calculate that value during each iteration of the loop. It is best to calculate the loop constant once before the loop rather than repeatedly during each iteration. Example 4-10 shows two ways of executing the loop.

Example 4-10 Different use of loop variable

```
public lc1()
{
    String str1 = "abcdefghij";
    for(int i=0; i<str1.length(); i++)
    {
        // do something that doesn't change length of str1
    }
}

public lc2()
{
    String str1 = "abcdefghij";
    int len = str1.length();
    for(int i=0; i<len; i++)
    {
        // do something that doesn't change length of str1
    }
}
```

Method `lc2()` is much faster than method `lc1()`. Method `lc1()` wastes time by calculating the loop constant during each iteration of the loop. Method `lc2()` calculates the value of the loop constant only once. In our tests, `lc2()` performed 150% faster than `lc1()`.

Local variables

Local variables cost less than instance variables. The JVM knows the exact location of local or stack variables at compile time. However, the JVM performs lookups to determine the location of an object's instance variables. This lookup is very expensive.

Example 4-11 The use of local variables

```
int i, j;
public void lv1
{
    for(i=0; i<10000; i++)
    {
```

```

        j += i;
    }
}

public void lv2
{
    int temp;
    for(int k=0; k<10000; k++)
    {
        temp += k;
    }
    j = temp;
}

```

Method lv2() is much faster than method lv1() because it uses local variables as a loop counter and to store the temporary result before assigning it to the class instance variable after the loop. The compiler stores temp, a local variable, on the stack and has quick access to it. On the other hand, the compiler stores j, an instance variable, on the heap and requires a lookup on each usage. Thus, every time temp is used, it can be accessed much quicker than j, an instance variable stored on the heap. Similarly, using k as a loop counter instead of i also gives faster performance.

Early termination

Often, a program cycles through loops without executing anything. Although the code in the loop body may not execute, the program still performs each iteration of the loop. In large loops with multiple conditions, this becomes expensive. The best solution is to avoid execution of the code. In such cases, terminate the loop early and allow the program to move forward.

Example 4-12 shows two functions that do the identical job: Print a line if a specified character is in the string.

Example 4-12 Early termination of the loop

```

public void et1(char c)
{
    String str = "abcdefghijklmnopqrstuvwxyz";
    int len = str.length();
    for(int i=0; i<len; i++)
    {
        if (str.charAt(i) == c)
        {
            System.out.println("Character " + c + " is in the string.");
        }
    }
}

```

```

public void et2(char c)
{
    String str = "abcdefghijklmnopqrstuvwxyz";
    int len = str.length();
    for(int i=0; i<len; i++)
    {
        if (str.charAt(i) == c)
        {
            System.out.println("Character " + c + " is in the string.");
            break;
        }
    }
}

```

Method `et2()` improves performance over method `et1()` because it may terminate the loop much earlier. For example, method `et1()` continues to iterate through the loop even after it finds the specified character in the list. Method `et2()` breaks out of the loop as soon as it finds the character. If the character lies towards the end of the string, the performance is almost identical for the two methods. However, if the character lies at the beginning of the string, method `et2()` breaks the loop almost immediately, whereas method `et1()` continues to iterate through the loop until it checks the whole string.

String handling (String, StringBuffer, char[])

In Java, Strings are immutable and String processing is very expensive. Any modification of a String object causes at least one other String object to be created. Creation and initialization of new objects involves a significant amount of overhead. Increasing the modifications of a String leads to more processing and results in higher CPU utilization, preventing the CPU from completing tasks more important to the application. Thus, we recommend that you avoid using String processing unless absolutely necessary.

StringBuffer objects present a better alternative because they are mutable. StringBuffers can be modified without creation of any new objects. However, one potential pitfall to avoid with StringBuffers is resizing. As they grow dynamically to accommodate the data they hold, StringBuffers are quite flexible but costly. When initially creating a StringBuffer object, you should always size it according to the maximum expected size. An even faster possibility is to use character arrays because characters are a primitive data type. Again, arrays should also be sized according to the maximum expected size.

Data structures (Hashtable, ArrayList, Vector)

Vector/ArrayList

Commercial applications often use data collections, specifically Vectors, ArrayLists, Hashtables, and their derivatives. ArrayLists tend to display faster performance than Vectors mainly due to the synchronization overhead in Vectors. ArrayLists are Vectors without synchronization, and are the better choice in a single-threaded application. As most applications use multithreading, we'll concentrate on optimizing Vectors. The same rules can be applied to ArrayLists.

Vectors allocate contiguous memory and grow dynamically in increments. Dynamic growth is accomplished in the following four steps:

1. Creating a new larger object to accommodate more elements
2. Copying the original elements into the new object
3. Destroying the old object
4. Returning the reference to the new object

The above process is expensive and therefore it is best to presize Vectors to the maximum expected amount to avoid dynamic growth.

Performance varies when inserting elements into Vectors. Insertion causes shifting of elements. Performance varies inversely with the number of elements to be shifted. Insertion of a new element is accomplished in three steps:

1. Copying elements, starting from the point of insertion to the end, into a temporary object
2. Copying the new element into the original object at the point of insertion
3. Copying the elements in the temporary object back into the original object

Thus, insertions at the end of a Vector give the best performance as no elements need to be shifted. Not surprisingly, insertions at the beginning give the worst performance due to the overhead of shifting each element. The worst scenario occurs when the program adds an element at the beginning of a full Vector (no room to add elements). This prompts the creation of two temporary copies of the same data for the dynamic growth of the Vector and the shifting of all elements. Avoid such scenarios by presizing Vectors and adding elements at the end of a Vector.

Hashtable

A Hashtable can be thought of as an “array” of linked lists. The program adds an element based on its unique key in the following three steps:

1. Computing the element's hashcode based on its unique key
2. Mapping the hashcode to a place in the array

3. Adding the element as a node to the linked list starting at the place in the array

Similarly, retrieval involves computing the hashcode to map to the starting of the linked list, and then sequentially traversing the linked list. Many factors, including capacity, load factor, and the `hashCode()` function, affect the performance of Hashtables.

Ideally, each place in the array contains a linked list with a single node for maximum efficiency. To accomplish this, the capacity and load factor must be set appropriately. The capacity determines the size of the array. If the number of expected elements is larger than the capacity of the array, you will end up with certain linked lists with multiple nodes, thereby reducing efficiency. The threshold, based on the capacity and load factor, determines when the Hashtable is rehashed, or resized, to accommodate more entries, and is calculated by the following formula:

$$\text{threshold} = \text{capacity} * \text{load factor}$$

If the load factor is more than 1.0, the threshold value becomes higher than the capacity, leading to more elements in the Hashtable than the capacity. We recommend setting the load factor below 1.0 to avoid multi-node linked lists.

Another factor, distribution of the entries within the Hashtable, affects performance. You want all entries distributed evenly throughout the Hashtable rather than many entries clustered in one area. The `hashCode()` method handles the distribution and should do so by using all unique aspects of an object to generate the hashcode. For example, use of only the first three digits of a social security number to generate the hashcode results in poor distribution. However, a good hashcode implementation would use all nine digits to generate a more unique hashcode, resulting in a more even distribution overall. Similarly, searching of a Hashtable uses the `equals()` method, which should be implemented in a manner consistent with the `hashCode()` method implementation.

Codepage issues

Java stores characters, Strings, StringBuffers and such data in the Unicode format. The zSeries servers use the EBCDIC format to store data. The necessary conversion between the different formats can be left to the Java program or the subsystem with which it communicates. Delegating format conversions to subsystem utilities is significantly faster than taking care of it in the Java program.

Synchronization

Applications where separate, concurrently running threads share data must consider the state and activities of other threads. Code segments within a program can access the same object from separate, concurrent threads. This code segment can be a block or a method and is identified with the keyword `synchronized`. The JVM then associates a lock with every object that has synchronized code and permits only one thread to access it at a time. This preserves the integrity of each thread but adds processing overhead, and each thread has to wait for the object to become available.

Most applications have several concurrent threads competing for resources, so it is necessary to design programs with the smallest possible synchronized code segments.

Consider the following while writing programs with synchronized code segments:

- ▶ The more you use synchronized methods, the more overhead you encounter, because of contention. Much contention can mean 10 times or worse compared to a non-synchronized case. With JIT compilers, this performance gap can increase 50-100 times. Use synchronized methods only when necessary. You can consider writing a synchronized and non-synchronized version of a method and calling them selectively.
- ▶ The JDK is written for general usage and therefore has a number of synchronized classes and methods. For example, `Hashtables`, `StringBuffers` and `Streams` are all extensively synchronized. If your program is not multithreaded, or if its threads do not access the same objects asynchronously, then you can improve the performance of your program by writing unsynchronized versions of the methods that these objects use or find an unsynchronized version that already exists.
- ▶ `Hashtable` and `Vector` are synchronized. If your application design does not need synchronization, then use Java 2 classes `HashMap` or `ArrayList` or implement your classes without synchronization.
- ▶ If you synchronize on an object via a synchronized block or method you are locking up the object, because each Java object has only one associated lock. To improve scalability you could use an entirely different object (`myLock`) to lock the critical path and protect access to a special member.
- ▶ Synchronization False Sharing Optimization: If two synchronized methods in the same object try to access completely unrelated resources, use two independent and distinct lock objects to prevent the shared resources. Otherwise they will lock each other out.
- ▶ Code Fusion Optimization: If two shared resources are related to one another and their updates seem to happen in tandem, fuse those two updates under the protection of a single object lock.

- ▶ Code Motion Optimization: Speed up execution inside the synchronized methods and blocks and release shared resources as fast as possible (e.g., Loop optimization).
- ▶ Synchronized code should contain access to shared resources only and nothing else. Code that does not directly manipulate shared resources should not reside within the synchronization scope.
- ▶ When analyzing synchronization hot spots, the first question should be: Can I restructure my design so as to eliminate the need for synchronization?
- ▶ Consider writing a synchronized and a non-synchronized method and calling them as required in the code.

Exception handling

Java provides a mechanism known as *exceptions* to help programs report and handle errors. When an error occurs, the program throws an exception. This means that the normal flow of the program is interrupted and the runtime environment attempts to find an exception handler—a block of code that can handle a particular type of error. The exception handler can attempt to recover from the error or, if the error is unrecoverable, provide an orderly exit from the program.

Here are some points to consider while using exception handling:

- ▶ There is significant overhead in handling exceptions, so exceptions should be used only in “exceptional” conditions. There is overhead equivalent to several hundred lines of code execution for getting a snapshot of the stack when an exception is created.
- ▶ In the Web application environment with multithreaded components like servlets, reuse exception objects as often as possible to avoid the overhead of creating a new instance of exception each time. You can create exception objects in the `init()` method and reuse them in the business methods.

Example 4-13 Reusing an exception object

```
public static Exception REUSABLE_EX = new Exception();
....
public void method1(int i) throws Exception {

    ....
    if (i==100)
        throw REUSABLE_EX;
}
```

instead of

Example 4-14 Creating many exception objects

```
public void method2(int i) throws Exception {  
  
    ....  
    if (i==100)  
        throw new Exception(); // this is 50 - 100 x slower!  
}
```

Final variables

Use `static final` when creating constants when data is invariant, that is, declare it as static and final. Performance can be improved by using local variables in the following examples. The first array, Example 4-15, needs to be created and initialized each time the object test is instantiated, but the second array, Example 4-16, is only initialized once.

Example 4-15 Initializing an object each time

```
int myarray[] = {1,2,3,4,5,6,7,8,9,10,2,3,4,5,6,7,8,9,10,11,  
3,4,5,6,7,8,9,10,11,12, 4,5,6,7,8,9,10,11,12,13,5,6,7,8,9,10,11,12,13,14};
```

Example 4-16 Initializing an object only once

```
static final int myarray2 [] = {1,2,3,4,5,6,7,8,9,10,  
2,3,4,5,6,7,8,9,10,11, 3,4,5,6,7,8,9,10,11,12,  
4,5,6,7,8,9,10,11,12,13,5,6,7,8,9,10,11,12,13,14};
```

Object caching

When an object is created, then in addition to the JVM's handling the basic object creation, the object's constructor and constructors of all its parent classes are also run and new memory is allocated by the operating system. Therefore, creating objects is expensive and one should avoid creating unnecessary objects.

A good technique is to reuse objects as often as possible because this can lower the runtime overheads and garbage collection costs. Example 4-18 ran about 50 times faster than Example 4-17.

Example 4-17 Instantiating many times

```
for (int _i = 0; _i < 10000; _i++) {  
    h = GregorianCalendar.getInstance().get(Calendar.HOUR);  
    m = GregorianCalendar.getInstance().get(Calendar.MINUTE);  
    s = GregorianCalendar.getInstance().get(Calendar.SECOND);  
}
```

Example 4-18 Reusing the instantiated object

```
GregorianCalendar g = new GregorianCalendar();
for (int _i = 0; _i < 10000; _i++) {
    h = g.get(Calendar.HOUR);
    m = g.get(Calendar.MINUTE);
    s = g.get(Calendar.SECOND);
}
```

Cache and reuse frequently used objects when possible. This saves the cost of object creation/initialization and later garbage collection and offers significant performance improvement to your applications. The efficiency of cache and the performance gain obtained by a cache mechanism depends on the following:

- ▶ The time spent on retrieving the requested information if not found in the cache. This could be little, for example, when reading from a fast local hard-drive, or a lot, when using a slow network connection.
- ▶ The ratio of the number of times that requested information is actually in the cache (a cache hit) over the number of times it is not in the cache (a cache miss).
- ▶ The right choice of collection to store and access the cached objects (see “Data structures (Hashtable, ArrayList, Vector)” on page 75).
- ▶ When implementing a cache, provide a cache statistic to verify the cache efficiency. Do the following:
 - Collect the following information: Number of requests, number of not satisfied requests, number of possible entries, number of current entries with/without active status
 - Cache read-only and not frequently changed information, if possible.
 - Consider that a very large set of objects in memory may strain memory resources and lead to excessive paging.
- ▶ Use Command objects to transparently add a remote stub cache to an RMI application. Caching stubs keeps them from being garbage collected, and may prevent an RMI server from closing. Use a policy to expire stubs and delete them from the cache.

Java Naming Interface (JNI)

Minimize the number of JNI calls. Group native operations to reduce the number of JNI calls. Consider consolidating transactions or operations to minimize the number of JNI calls needed to accomplish a task. Reducing the number of times the JNI overhead needs to be paid improves performance.

Attention: If you use JNI and something goes wrong in the called code, it may crash the J2EE server, so use of JNI is not recommended.

CLASS files and CLASSPATH

Prudent use of zip and jar files can improve load time. Zip and jar files can be used to combine many class files into one file for easier loading or file transfer. When done properly, this can improve application load time. However, avoid adding unneeded class files, which will increase file size and increase memory usage. One common mistake is to produce a zip file of every utility and builder class when the application may need only a small proportion of the class files to execute. This results in increased load time.

Reorder the CLASSPATH. When the JVM is looking for a class, it searches the directories in the order they are given in CLASSPATH. You can improve the start-up times of your Java applications by reordering the paths in the CLASSPATH environment variable by placing the most used libraries earlier in the CLASSPATH.

4.2.2 JDBC and SQLJ

You can see many JDBC-related optimization techniques on the following Web site:

<http://www.precisejava.com/javaperf/j2ee/JDBC.htm>

JDBC, SQLJ and their differences

What is JDBC?

JDBC is an application programming interface (API) that Java applications use to access any relational database. DB2 for OS/390 and z/OS's support for JDBC enables you to write Java applications that access local DB2 data or remote relational data on a server that supports DRDA. DB2 for OS/390 and z/OS V7 is fully compliant with the JavaSoft JDBC 1.2 and 2.0 specification.

The purpose of the APIs is to provide a generic interface for writing platform-independent applications that can access any SQL database. The APIs are defined in 16 classes that support basic SQL functionality for connecting to a database, executing SQL statements, and processing results. Together, these interfaces and classes represent the JDBC capabilities by which a Java application can access relational data.

More information on JDBC and SQLJ programming techniques and examples can be found in *e-business Cookbook for z/OS: Java Development*, SG24-5980.

Using JDBC with DB2

DB2 JDBC offers a number of advantages for accessing DB2 data:

- ▶ JDBC combines the benefit of running your applications in an OS/390 environment with the portability and ease of writing Java applications. Using the Java language, you can write an application on any platform and execute it on any platform to which the Java Development Kit (JDK™) is ported.
- ▶ The ability to develop an application once and execute it anywhere offers the potential benefits of reduced development, maintenance, and systems management costs, and flexibility in supporting diverse hardware and software configurations.
- ▶ The JDBC interface offers the ability to change between drivers and access a variety of databases without recoding your Java program.
- ▶ JDBC applications do not require precompiles.

DB2 supports both JDBC1.2 and JDBC 2.0, as follows:

- ▶ The SQLJ/JDBC driver with JDBC 1.2 support, which is fully compliant with the JDBC 1.2 and SQLJ – Part 0 specification
To use this version of JDBC, you need the JDK for OS/390, Version 1.1.6 or higher.
- ▶ The SQLJ/JDBC driver with JDBC 2.0 support, which is fully compliant with the JDBC 1.2 and SQLJ – Part 0 specification and includes most of the functions of the JDBC 2.0 specification
To use this version of JDBC, you need the JDK for OS/390, Version 1.3 or higher. You need this JDBC driver if you use any of the JDBC 2.0 DataSource function or the global transactions that run under WebSphere Application Server Version 4.0 and above.

SQLJ and SQLJ for DB2

SQLJ provides support for embedded static SQL in Java applications and servlets. SQLJ was initially developed by Oracle, Tandem, and IBM to complement the dynamic SQL JDBC model with a static SQL model.

In general, Java applications use JDBC for dynamic SQL and SQLJ for static SQL. However, because SQLJ includes JDBC 1.2 or JDBC 2.0, an application program can create a JDBC connection and then use that connection to execute dynamic SQL statements through JDBC and embedded static SQL statements through SQLJ.

The SQLJ specification consists of three parts:

- ▶ *Database Languages – SQL – Part 0: Object Language Bindings (SQL/OLB)* is also known as SQLJ Part 0. It was approved by ANSI in 1998, and it

specifies the SQLJ language syntax and semantics for embedded SQL statements in a Java application.

- ▶ *Database Languages – SQLJ – Part 1: SQL Routines using the Java Programming Language* was approved by ANSI in 1999, and it specifies extensions that define:
 - a. Installation of Java classes in an SQL database
 - b. Invocation of static methods as stored procedures
- ▶ *Database Languages – SQLJ – Part 2: SQL Types using the Java Programming Language* is under development. It specifies extensions for accessing Java classes as SQL user-defined types.

The DB2 for OS/390 and z/OS implementation of SQLJ includes support for Part 0 and the ability to invoke a Java static method as a stored procedure, which is in Part 1.

SQLJ with static SQL can be faster. Figure 4-1 on page 84 shows that the database engine can perform authorization checking, validation and optimization at Bind time and avoid this overhead during execution. This can lead to a 20%-50% performance gain for SQLJ over JDBC.

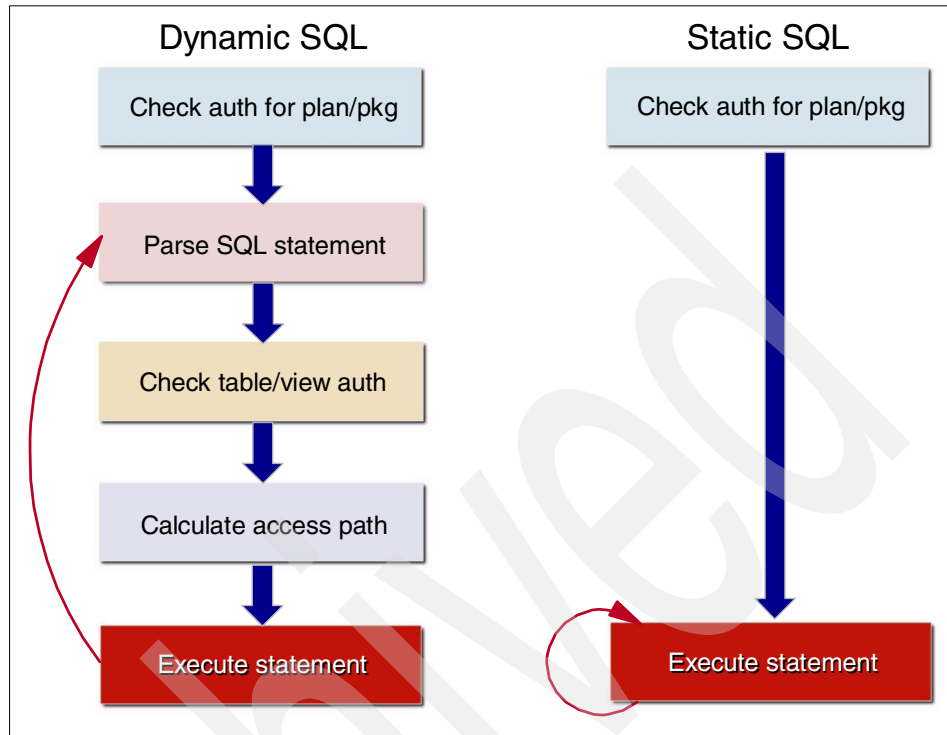


Figure 4-1 Dynamic SQL (JDBC) vs. static SQL (SQLJ)

Some of the major differences between SQLJ and JDBC

- SQLJ follows the static SQL model and JDBC follows the dynamic SQL model.

When the syntax of embedded SQL statements is fully known at precompile time, the statements are referred to as *static* SQL. This is in contrast to *dynamic* SQL statements whose syntax is not known until runtime.

When a static SQL statement is prepared, an executable form of the statement is created and stored in the package in the database. See Figure 4-2 on page 85. The executable form can be constructed either at precompile time, or at a later Bind time. In either case, preparation occurs before runtime. The authorization of the person binding the application is used, and optimization is based upon database statistics and configuration parameters that may not be current when the application runs.

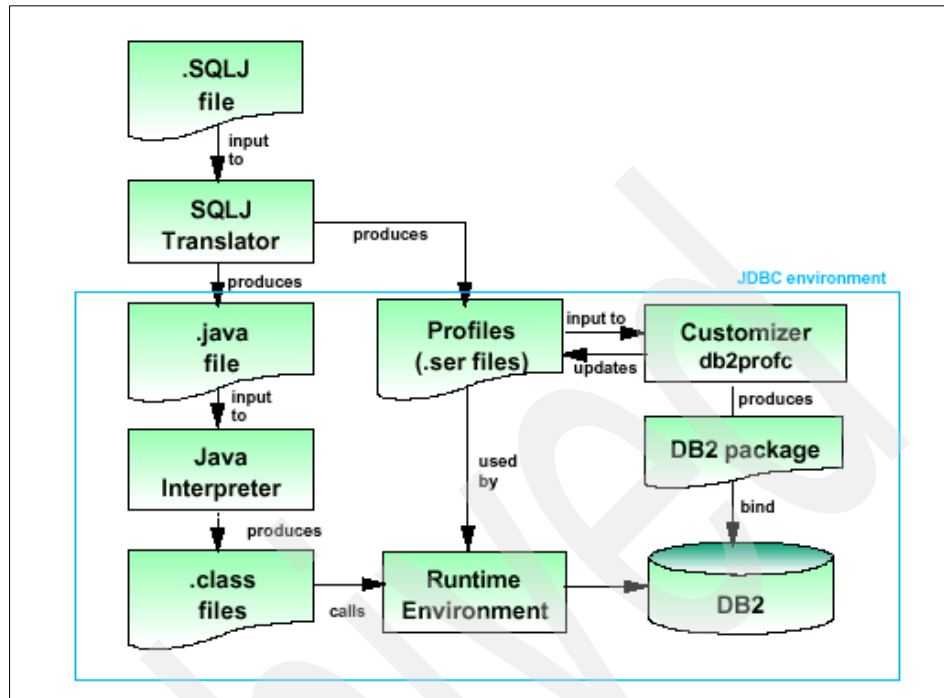


Figure 4-2 Embedded SQL for Java

- Because the authorization of the person binding the application is used, the end user does not require direct privileges to execute the statements in the package. For example, an application could allow a user to update parts of a table without granting an update privilege on the entire table. This can be achieved by restricting the static SQL statements to allow updates only to certain columns or a range of values.

SQLJ provides the advantages of static SQL authorization checking. With SQLJ, the authorization ID under which SQL statements execute is the plan or package owner. DB2 checks table privileges at Bind time. Because JDBC uses dynamic SQL, the authorization ID under which SQL statements execute is not known until runtime, so no authorization checking of table privileges can occur until runtime.

- Static SQL statements are persistent, meaning that the statements last for as long as the package exists. Dynamic SQL statements are cached until they are invalidated, or freed for space management reasons, or the database is shut down. If required, the dynamic SQL statements are recompiled implicitly by the DB2 SQL compiler whenever a cached statement becomes invalid. For information on caching and the reasons for invalidation (for example an

ALTER TABLE statement) of a cached statement, refer to *DB2 Universal Database for OS/390 and z/OS, SQL Reference, Version 7*, SC26-9944-01.

- ▶ The key advantage of static SQL, with respect to persistence, is that the static statements exist after a particular database is shut down, whereas dynamic SQL statements cease to exist when this occurs. In addition, static SQL does not have to be compiled by the DB2 SQL compiler at runtime, while dynamic SQL must be explicitly compiled at runtime. Because DB2 caches dynamic SQL statements, the statements may not need to be compiled often by DB2, but they must be compiled at least once when you execute the application.
- ▶ There can be significant performance advantages to static SQL. For simple, short-running SQL programs, a static SQL statement executes faster than the same statement processed dynamically since the overhead of preparing an executable form of the statement is done at precompile time instead of at runtime.
- ▶ SQLJ source programs are smaller than equivalent JDBC programs, because certain code that the programmer must include in JDBC programs is generated automatically by SQLJ.
- ▶ SQLJ can do data type checking during the program preparation process to determine whether table columns are compatible with Java host expressions. JDBC passes values to and from SQL tables without compile-time data type checking.

For more information on JDBC and SQLJ programming techniques and choosing between static and dynamic SQL, refer to *e-business Cookbook for z/OS: Java Development*, SG24-5980.

A simple SQLJ vs. JDBC performance test

In Example 4-19 on page 87, program `Sqljtest1.sqlj` uses SQLJ with plan `SQLJDB2` and package `DB2A.SQLJDB2.SQLJ11` to insert rows into a DB2 table. To compile and execute the SQLJ program, it is necessary to do the following:

- ▶ Preprocess it with the SQLJ translator.
- ▶ Compile the Java code.
- ▶ Customize the serialized profile.
- ▶ Bind the DBRMs.
- ▶ Configure the environment variable `DB2SQLJPROPERTIES` with the plan name `SQLJDB2`.

See *e-business Cookbook for z/OS: Java Development*, SG24-5980 for more details on SQLJ and JDBC programming. After this is done, the SQLJ program can be executed from the OMVS shell.

Jdbctest2.java is a sample java program that uses JDBC to insert rows into a DB2 table.

Example 4-19 Sqljtest1.sqlj

```
/**
 * Sample SQLJ code to insert rows into a table using SQLJ
 * Creation date: (12/3/2001 9:30:19 PM)
 * @author: Vasukh
 */
import java.sql.*;
import sqlj.runtime.ref.*;
import java.io.*;
public class Sqljtest1 {
    static {
        try {
            // Registers the DB2 for OS/390 driver with DriverManager
            Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
    public Sqljtest1() {
        super();
    }
    public static void main(String args[]) {
        String url = "jdbc:db2os390sqlj:";
        String st = "This is the jdbc versus sqlj test";
        int balance = 1;
        int rs = 0;
        try {
            System.out.println("**** SQLJ Entry within class sample01.");
            // Create the connection. SQLJ uses the JDBC connection object
            Connection con0 = DriverManager.getConnection(url);
            System.out.println("**** SQLJ Connection to DB2 for OS/390.");
            // Create the Statement object
            Statement stmt = con0.createStatement();
            System.out.println("**** SQLJ Statement Created");
            for (int j = 1; j < 100; j++) {
                for (int i = 1; i < 1000; i++) {
                    #sql { INSERT INTO JAVA3.JDBCSQLJTEST VALUES( :st, :balance) };
                    balance = balance + i;
                }
                System.out.println("COMPLETED set of : " + j * 1000 + "
inserts");
            }
            System.out.println("**** COMPLETED all the inserts!!!");

            // Close the statement
            stmt.close();
        }
    }
}
```

```

        System.out.println("**** SQLJ Statement Closed");
        // Assure connection is on a unit of work boundary prior to close
        // Close the connection
        con0.close();
        System.out.println("**** SQLJ Disconnect from DB2 for OS/390.");
        System.out.println("**** SQLJ Exit from class sample01 - no
Errors.");
    } catch (SQLException sqle) {
        //=====> Processing of a standard java.sql.SQLException;
        System.out.println(
            "SQLException: "
            + sqle
            + ". SQLSTATE="
            + sqle.getSQLState()
            + " SQLCODE="
            + sqle.getErrorCode());
        sqle.printStackTrace();
    } catch (Exception e) {
        System.out.println("Exception: " + e);
        e.printStackTrace();
    }
}
}

```

Example 4-20 Jdbctest2.java

```

/**
 * Jdbctest2 class uses JDBC to access DB2 with Prepared Statement
 * Creation date: (12/3/2001 9:30:19 PM)
 * @author: Vasukh
 */
import java.sql.*;
public class Jdbctest2 {
    static {
        try {
            // register the DB2 for OS/390 driver with DriverManager
            Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
    public Jdbctest2() {
        super();
    }
    public static void main(String args[]) {
        String url = "jdbc:db2os390sqlj:";
        int balance = 1;
        int rs = 0;
        try {

```

```

        System.out.println("**** JDBC Entry within class Jdbctest2");
        // Create the connection
        Connection con = DriverManager.getConnection(url);
        System.out.println("**** JDBC Connection to DB2 for OS/390.");
        // Declare and create the Prepared Statement
        PreparedStatement pstmt;
        pstmt = con.prepareStatement("INSERT INTO JDBCSQLJTEST VALUES(?,
?)");
        System.out.println("**** JDBC Prepared Statement Created");
        for (int j = 0; j < 100; j++) {
            for (int i = 0; i < 1000; i++) {
                pstmt.setString(1, "OK perfect insert");
                // pstmt.setString(2, String.valueOf(balance))
                pstmt.setInt(2, balance);
                pstmt.executeUpdate();
                balance = balance + i;
            }
            System.out.println("COMPLETED set of : " + j * 1000 + "
inserts");
        }
        System.out.println("**** JDBC Result Set Created");
        // Close the statement
        pstmt.close();
        System.out.println("**** JDBC Prepared Statement Closed");
        // Close the connection
        con.close();
        System.out.println("**** JDBC Disconnect from DB2 for OS/390.");
        System.out.println("**** JDBC Exit from class Jdbctest2- no
Errors.");
    } catch (SQLException sqle) {
        //=====> Processing of a standard java.sql.SQLException;
        System.out.println(
            "SQLException: "
            + sqle
            + ". SQLSTATE="
            + sqle.getSQLState()
            + " SQLCODE="
            + sqle.getErrorCode());
        sqle.printStackTrace();
    } catch (Exception e) {
        System.out.println("Exception: " + e);
        e.printStackTrace();
    }
}
}

```

The programs Jdbctest1 and SQLJtest1 were executed from the OMVS shell and the performance recorded. In this simple test the SQLJ program executed around 30% faster than the equivalent JDBC program.

In large applications with many tables and a large number of SQL statements, there can be a significant performance gain by using SQLJ.

JDBC and SQLJ connection pooling

To serve client requests, database connections are acquired and released several times. This is an expensive operation and the application server process can take around one to three seconds to establish a database connection (that includes communicating with the server, authenticating, and so forth), and that needs to be done for every client (EJB) request.

Connection pooling is a technique that allows multiple clients to share a cached set of connection objects that provide access to a database resource.

Connection pooling is part of JDBC 2.0 data source support. Connection pooling is a framework for caching physical data source connections, which are equivalent to DB2 threads. When JDBC reuses physical data source connections, the expensive operations that are required for the creation and subsequent closing of `java.sql.Connection` objects are minimized. Connection pooling is a built-in part of JDBC 2.0 data source support. Connection pooling support is completely transparent to the JDBC application.

Without connection pooling, each `java.sql.Connection` object represents a physical connection to the database. When the application establishes a connection to a data source, DB2 creates a new physical connection to the data source. When the application calls the `java.sql.Connection.close` method, DB2 terminates the physical connection to the data source.

In contrast, with connection pooling, a `java.sql.Connection` object is a temporary, logical representation of the physical data source connection.

The JDBC 2.0 connection pooling framework lets a single physical data source connection be serially reused by logical `java.sql.Connection` instances. The application can use the logical `java.sql.Connection` object in exactly the same manner as it uses a `java.sql.Connection` object when there is no connection pooling support.

With connection pooling, when a JDBC application invokes the `DataSource.getConnection` method, the data source determines whether an appropriate physical connection exists. If an appropriate physical connection exists, the data source returns a `java.sql.Connection` instance to the application.

When the JDBC application invokes the `java.sql.Connection.close` method, JDBC does not close the physical data source connection. Instead, JDBC closes only JDBC resources, such as `Statement` or `ResultSet` objects. The data source returns the physical connection to the connection pool for reuse.

WebSphere Application Server implements a connection pool manager and can optimize resource usage by dynamically altering the pool size, based on demand. The client code performs a JNDI lookup for the `DataSource` defined in WebSphere. To improve performance, the overhead of JNDI lookup can be avoided by using a caching technique described in the Context lookup section.

Performance hints and tips using JDBC/SQLJ

Use stored procedures for multiple SQL statements

A *stored procedure* is a compiled program, stored at a DB2 local or remote server, that can execute SQL statements. A typical stored procedure contains two or more SQL statements and some manipulative or logical processing in a host language. A client application program uses the SQL statement `CALL` to invoke the stored procedure. Stored procedures in DB2 can be called using the JDBC or SQLJ APIs as well. There are some performance gains that can be achieved by correctly exploiting stored procedures.

The best reason for using a stored procedure is that you have a complex series of SQL statements that must be executed in the same sequence every time. By encapsulating this work in a stored procedure, your JDBC can simply call the stored procedure. This also means that you can change the details of the stored procedure, and your JDBC code is not affected.

Another benefit is that a stored procedure can be compiled, and therefore use static SQL, which is inherently faster than any forms of dynamic SQL (including JDBC) since the SQL processing and optimization is done at compile time and stored with the program object. The performance benefit of using stored procedures increases as the number of SQL statements increases and the transaction gets more complex.

Use SQLJ instead of JDBC if possible

As discussed earlier, SQLJ generally performs better than JDBC. But make sure you follow all four SQLJ preparation steps (see Figure 4-3 on page 92) every time you change anything. If you fail to do this, you will not get the benefits of static SQL.

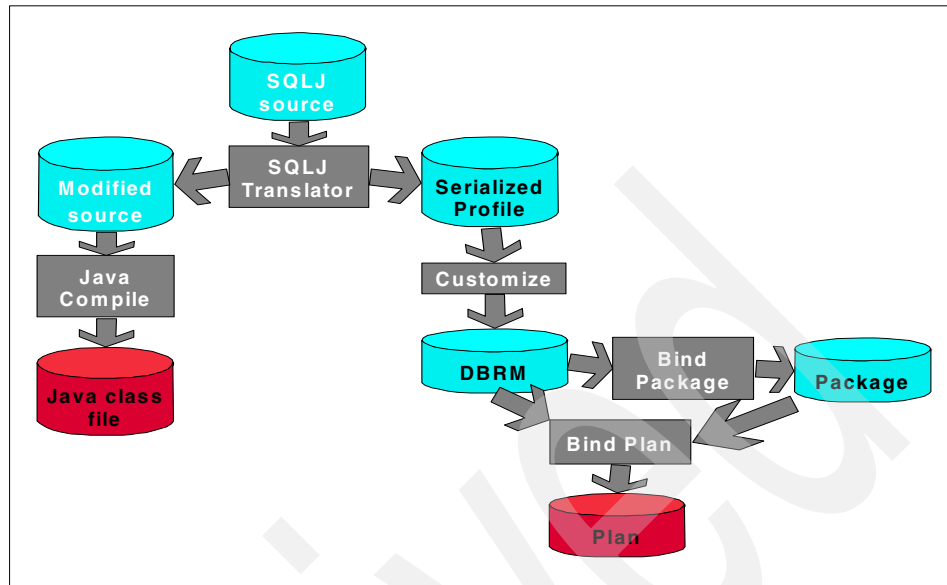


Figure 4-3 SQLJ preparation steps

The four steps are:

1. Translate SQLJ source into modified source and serialized profile.
2. Compile modified Java source into bytecodes.
3. Customize serialized profiles to produce DBRMs.
4. Bind DBRMs into a plan.

Use connection pooling

Creating and closing connection is expensive. The JDBC 2.0 specification, which is part of the J2EE specification, provides a standard mechanism of connection pooling by extending `javax.sql.DataSource`. Most application server vendors and database vendors have implemented this extension. When the application requires a connection, it gets one from a preconfigured pool of the application server, and the connection is returned to the pool when the application closes the connection. Example 4-21 illustrates obtaining a data source.

Example 4-21 Obtaining a data source

```

// obtain the initial JNDI context
Context initCtx = new InitialContext();
// perform JNDI lookup to obtain resource factory
javax.sql.DataSource ds = (javax.sql.DataSource)
initCtx.lookup("java:comp/env/jdbc/sampleDB");
// Invoke factory to obtain a resource. The security

```



```
// principal for the resource is not given, and therefore
// it will be configured by the Deployer.
java.sql.Connection con = ds.getConnection();
....// data access logic
con.close();
```

Make sure the connection is closed in the “catch” or “finally” clause of the try statement as soon as it is no longer needed so it can be released back to the pool for other requests. Failing to close and release JDBC connections can cause other users to experience long waits for connections. Although a JDBC connection that is left unclosed will be reaped and returned by WebSphere Application Server after a time-out period, others may have to wait for this to occur.

Optimize SQL statements

First, one SQL is better than two. Second, SQL statements need to be examined and tuned. This can lead to performance improvement in query processing by DB2. SQL tuning can produce performance improvements for both JDBC or SQLJ. The access plan produced by DB2 for each query is retrieved from the PLAN_TABLE and examined for possible improvements. See *Application Programming and SQL Guide*, SC26-9933 for more information on tuning SQL for DB2.

Data type matching

Make sure variable data types in the Java application match database schema exactly to prevent unnecessary table scans and poor response times.

Close Statement, PreparedStatement and ResultSet

These objects should be closed as soon as they are no longer needed to avoid running out of cursor in the application.

Using java.sql.PreparedStatement instead of java.sql.Statement

This applies if the application uses the same statement repeatedly. Prepared statements are prepared once and can be reused over and over. There is another reason why JDBC application should use PreparedStatement to improve performance. In order to set up a result set, two database calls are made for each column in the ResultSet object. The first one asks the database to describe the column, the second one tells the database where to put the data when it is fetched. When a Statement object is used, these two extra calls are made for every SQL statement that is executed. For PreparedStatement objects, these calls are only made at the time the object is constructed.

Example 4-22 and Example 4-23 show the difference between how to use Statement and PreparedStatement.

Example 4-22 Using Statement

```
//SAMPLEA: example uses java.sql.Statement
java.sql.Connection con = ds.getConnection();
Statement stmt = con.createStatement();
// Execute a Query
ResultSet rs=stmt.executeQuery("SELECT NAME FROM EMPLOYEE WHERE ID=7");
```

Example 4-23 Using PreparedStatement

```
//SAMPLEB: example uses java.sql.PreparedStatement
java.sql.Connection con = ds.getConnection();
PreparedStatement ps =
    con.prepareStatement("SELECT NAME FROM EMPLOYEE WHERE ID=?");
// Execute a Query and generate a ResultSet instance
ps.setInt(1, 7);
ResultSet rs = ps.executeQuery();
```

A test we did on these samples showed that ps.executeQuery() method in Example 4-23 ran at least 20-50 times faster during repeat calls, compared to the first call on the same method and stmt.executeQuery() in Example 4-22.

Eliminating retrieve extra column

One common example is to use “SELECT * FROM...” to retrieve all columns of a result set but only a smaller portion of the columns are needed. The application performance is degraded by forcing the database to move more data and JDBC to make more column Describe and Bind calls.

Choose optimal transaction isolation level

The isolation level represents how a database maintains data integrity against problems such as dirty reads, phantom reads, and nonrepeatable reads that can occur due to concurrent transactions. java.sql.Connection interface provides methods and constants to avoid the above-mentioned problems by setting different isolation levels, as follows:

```
public abstract interface Connection {
    public static final int TRANSACTION_NONE = 0
    public static final int TRANSACTION_READ_COMMITTED = 2
    public static final int TRANSACTION_READ_UNCOMMITTED = 1
    public static final int TRANSACTION_REPEATABLE_READ = 4
    public static final int TRANSACTION_SERIALIZABLE = 8
    int getTransactionIsolation();
    void setTransactionIsolation(int isolationlevelconstant);
}
```

You can get the existing isolation level with the `getTransactionIsolation()` method and set the isolation level with `setTransactionIsolation(int isolationlevelconstant)` by passing above constants to this method. The following table describes isolation levels against the problems they prevent.

Table 4-1 Isolation levels and the problems they prevent

Transaction level	Performance phenomena			Performance impact
	Dirty reads	Non-repeatable reads	Phantom reads	
TRANSACTION_NONE	N/A	N/A	N/A	FASTEST
TRANSACTION_READ_UNCOMMITTED	YES	YES	YES	FASTEST
TRANSACTION_READ_COMMITTED	NO	YES	YES	FAST
TRANSACTION_REPEATABLE_READ	NO	NO	YES	MEDIUM
TRANSACTION_SERIALIZABLE	NO	NO	NO	SLOW

YES means that the isolation level does not prevent the problem.

NO means that the isolation level prevents the problem.

By setting isolation levels, you are having an impact on the performance as mentioned in Table 4-1. The database uses read and write locks to control the isolation levels. Let us have a look at each of these problems and then look at the impact on performance.

► The dirty read problem

Figure 4-4 on page 96 illustrates the dirty read problem.

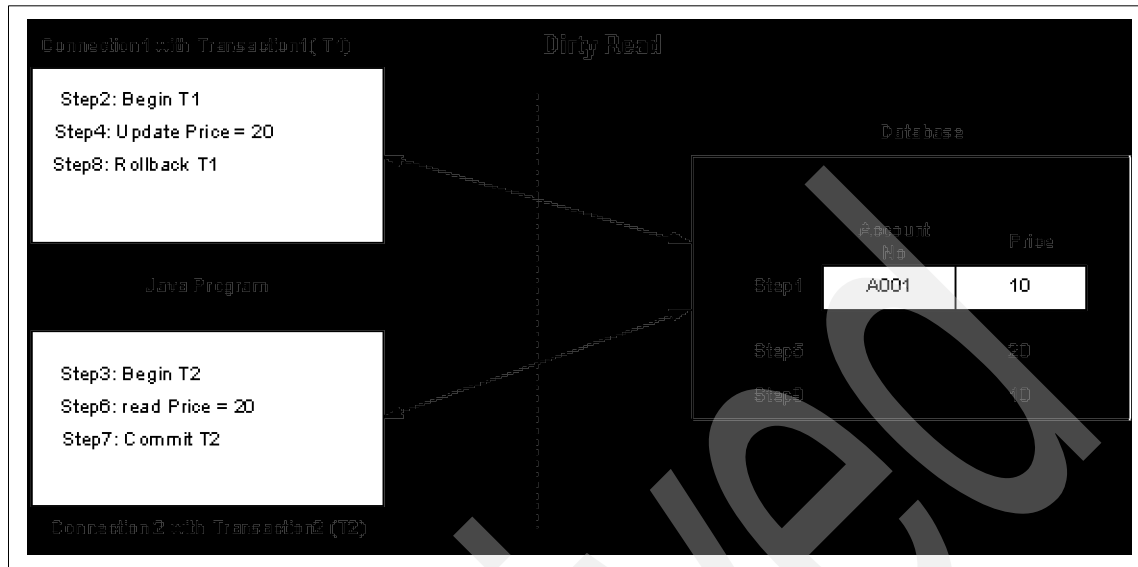


Figure 4-4 Dirty read problem

Description of the Steps:

- Step 1: Database row has PRODUCT = A001 and PRICE = 10.
- Step 2: Connection1 starts Transaction1 (T1).
- Step 3: Connection2 starts Transaction2 (T2).
- Step 4: T1 updates PRICE =20 for PRODUCT = A001.
- Step 5: Database has now PRICE = 20 for PRODUCT = A001.
- Step 6: T2 reads PRICE = 20 for PRODUCT = A001.
- Step 7: T2 commits the transaction.
- Step 8: T1 rolls back the transaction because of some problem.

The problem is that T2 gets incorrect PRICE=20 for PRODUCT = A001 instead of 10 because of an uncommitted read. Obviously, it is dangerous in critical transactions to read inconsistent data. If you are sure about not accessing data concurrently, then you can allow this problem by setting TRANSACTION_READ_UNCOMMITTED or TRANSACTION_NONE, which in turn improves performance, otherwise you have to use TRANSACTION_READ_COMMITTED to avoid this problem.

► The unrepeatable read problem

Figure 4-5 on page 97 illustrates the unrepeatable read problem.

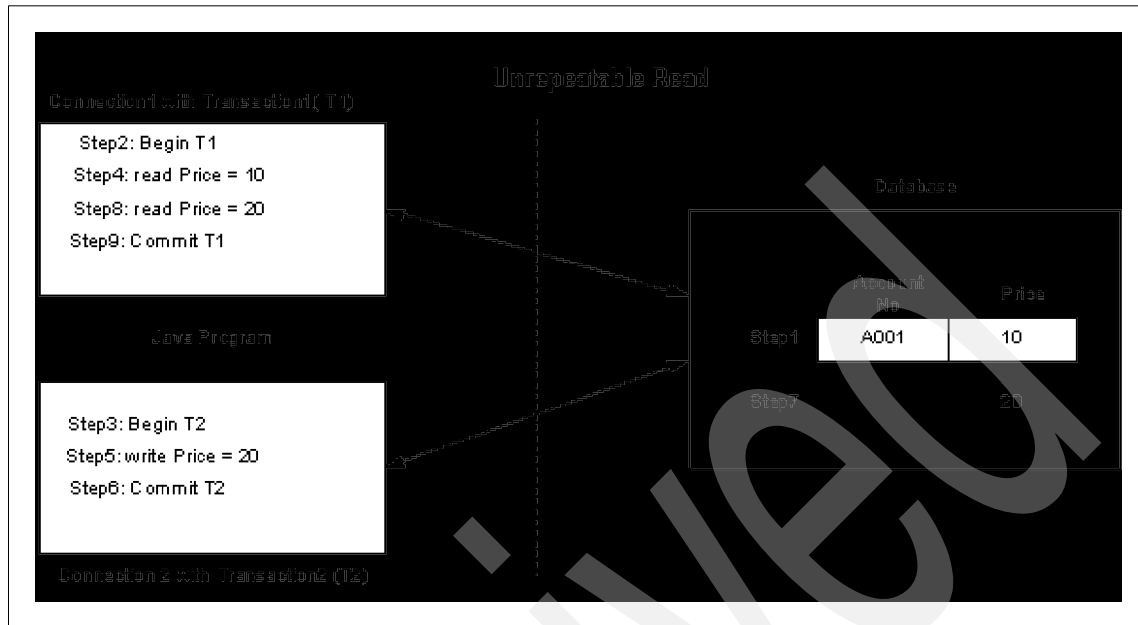


Figure 4-5 Unrepeatable read problem

Description of the Steps:

- Step 1: Database row has PRODUCT = A001 and PRICE = 10.
- Step 2: Connection1 starts Transaction1 (T1).
- Step 3: Connection2 starts Transaction2 (T2).
- Step 4: T1 reads PRICE =10 for PRODUCT = A001.
- Step 5: T2 updates PRICE = 20 for PRODUCT = A001.
- Step 6: T2 commits the transaction.
- Step 7: Database row has PRODUCT = A001 and PRICE = 20.
- Step 8: T1 reads PRICE = 20 for PRODUCT = A001.
- Step 9: T1 commits the transaction.

Here the problem is that Transaction1 reads 10 the first time and 20 the second time, but it is supposed to always be 10 whenever it reads a record in that transaction. You can control this problem by setting the isolation level as TRANSACTION_REPEATABLE_READ.

► The phantom read problem

Figure 4-6 on page 98 illustrates the phantom read problem.

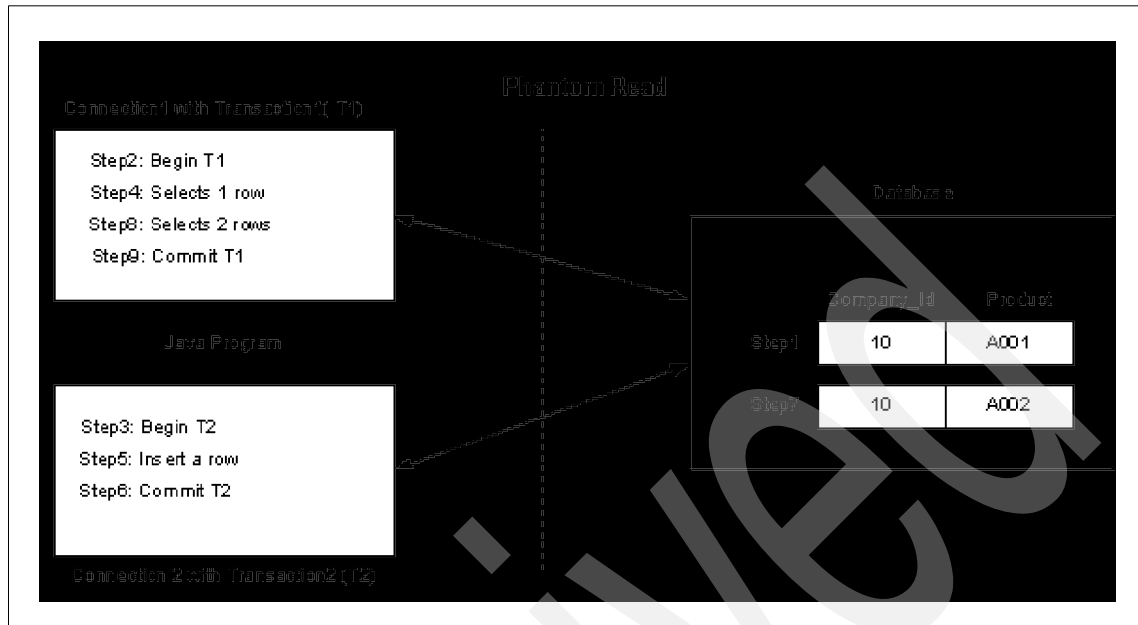


Figure 4-6 Phantom read problem

Description of the Steps:

Step 1: The database has a row `PRODUCT = A001` and `COMPANY_ID = 10`.

Step 2: Connection1 starts Transaction1 (T1).

Step 3: Connection2 starts Transaction2 (T2).

Step 4: T1 selects a row with condition `SELECT PRODUCT WHERE COMPANY_ID = 10`.

Step 5: T2 inserts a row with condition `INSERT PRODUCT=A002 WHERE COMPANY_ID= 10`.

Step 6: T2 commits the transaction.

Step 7: The database has two rows with that condition.

Step 8: T1 selects again with condition `SELECT PRODUCT WHERE COMPANY_ID=10` and gets two rows instead of one.

Step 9: T1 commits the transaction.

Here the problem is that T1 gets two rows instead of one row up on selecting the same condition the second time. You can control this problem by setting the isolation level as `TRANSACTION_SERIALIZABLE`.

► Choosing the right isolation level for your program

Choosing the right isolation level for your program depends upon your application's requirements. If you write a program for searching a product catalog from your database, you can easily choose TRANSACTION_READ_UNCOMMITTED because you need not worry about the problems that are mentioned above. Some other program can insert records at the same time and you don't have to bother much about that insertion. Obviously, this improves performance significantly.

If you write a critical program like a bank or stock analysis program, where you want to control all of the above-mentioned problems, you can choose TRANSACTION_SERIALIZABLE for maximum safety. This is a trade-off between safety and performance. Ultimately, we need safety here.

If you don't have to deal with concurrent transactions in your application, then the best choice is TRANSACTION_NONE to improve performance.

The other two isolation levels need a good understanding of your requirements. If your application needs only committed records, then the TRANSACTION_READ_COMMITTED isolation is a good choice. If your application needs to read a row exclusively till you finish your work, then TRANSACTION_REPEATABLE_READ is the best choice.

DB2 for z/OS supports all transaction isolation levels. The relationships are shown in Table 4-2.

Table 4-2 Relationships between SQLJ and DB2 isolation levels

TRANSACTION_READ_COMMITTED	Cursor stability
TRANSACTION_READ_UNCOMMITTED	Uncommitted read
TRANSACTION_REPEATABLE_READ	Read stability
TRANSACTION_SERIALIZABLE	Repeatable read

Control transaction

In general, a transaction represents one unit of work or a piece of code in the program that executes in its entirety or none at all. To be precise, it is *all or no* work. In JDBC, a transaction is a set of one or more Statements that execute as a single unit.

The java.sql.Connection interface provides some methods to control transactions, as follows:

```
public abstract interface Connection {
    boolean getAutoCommit();
    void      setAutoCommit(boolean autocommit);
    void      commit();
```

```

        void        rollback();
    }

```

In JDBC, by default, a transaction starts and commits after each statement's execution on a connection. That is, the AutoCommit mode is true. Programmers need not write a commit() method explicitly after each statement.

Obviously, this default mechanism offers a good facility to programmers to execute a single statement. But it gives poor performance when multiple statements on a connection are to be executed because a commit is issued after each statement by default, which in turn reduces performance by issuing unnecessary commits. The remedy is to turn AutoCommit mode to false and issue a commit() method after a set of statements executes. This is called a batch transaction. Use rollback() in catch block to roll back the transaction whenever an exception occurs in your program. Example 4-24 illustrates the batch transaction approach.

Example 4-24 Setting AutoCommit

```

try{
    connection.setAutoCommit(false);
    PreparedStatement ps = connection.prepareStatement( "UPDATE employee
SET Address=? WHERE name=?");
    ps.setString(1,"Austin");
    ps.setString(2,"RR");
    ps.executeUpdate();
    PreparedStatement ps1 = connection.prepareStatement( "UPDATE account
SET salary=? WHERE name=?");
    ps1.setDouble(1, 5000.00);
    ps1.setString(2,"RR");
    ps1.executeUpdate();
    connection.commit();
    connection.setAutoCommit(true);
}catch(SQLException e){ connection.rollback();}
finally{
    if(ps != null){ ps.close();}
    if(ps1 != null){ps1.close();}
    if(connection != null){connection.close();}
}

```

This batch transaction gives good performance by reducing commit calls after each statement's execution.

Use setFetchSize()

Use `java.sql.Statement.setFetchSize()` and `java.sql.PreparedStatement.setFetchSize()` to fetch more rows in the result set at once, as needed. These are the new APIs provided in the JDBC 2.0 driver to make access to result set data more efficient.

Use scrollable result sets

The JDBC 1.0 API provided result sets that scrolled in a forward direction only. JDBC 2.0 scrollable result sets allow for more flexibility in the processing of results by providing both forward and backward movement through their contents. In addition, scrollable result sets allow for relative and absolute positioning. An application should take advantage of this new feature to improve performance, especially when the result set is large.

Example 4-25 Setting scrollable result sets

```
Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
stmt.setFetchSize(25);
```

Use batch updates

The new batch update feature in the JDBC 2.0 specification allows an application to submit multiple update statements (insert/update/delete) in a single request to the database. This provides a significant performance improvement when a large number of update statements need to be executed.

Example 4-26 Batch update feature

```
// turn off autocommit
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.addBatch("INSERT INTO employees VALUES (1000, 'Joe Jones')");
stmt.addBatch("INSERT INTO departments VALUES (260, 'Shoe')");
stmt.addBatch("INSERT INTO emp_dept VALUES (1000, 260)");
// submit a batch of update commands for execution
int[] updateCounts = stmt.executeBatch();
```

Some getxxx functions are costly

Costly `getxxx()` functions include those getting complex objects. Examples are: `getDate()`, `getTimestamp()` are more expensive and slower than `getshort`, `getboolean`, `getInt`.

Positioned iterator and named iterator

In SQLJ, use *positioned* iterators rather than *named* iterators when possible. A *result set* iterator in SQLJ is a Java object that you use to retrieve rows from a result table. It is an instantiation of a specifically declared iterator class, with a fixed number of columns of predefined type. For a positioned iterator, the columns of the result set iterator correspond to the columns of the result table. You specify the type of the columns in left-to-right order. For a named iterator, you specify names and types for each of the iterator columns. These names and types must match the names of columns in the result table for the query.

Example 4-27 Use of a positioned iterator

```
#sql iterator ByPos(String,Date);
{
#sql iterator ByPos(String,Date);
// Declare positionediterator class ByPos
ByPos positer; // Declare object of ByPos class
String name = null;
Date hrdate;
#sql positer = { SELECT LASTNAME, HIREDATE FROM EMP };
#sql { FETCH :positer INTO :name, :hrdate };
// Retrieve the first row
while ( !positer.endFetch() )
{ System.out.println(name + " was hiredin " +
hrdate);
#sql { FETCH :positer INTO :name, :hrdate };
// Retrieve the rest of the rows
}
}
```

Example 4-28 Use of a named iterator

```
#sql iterator ByName(String LastName, Date HireDate);
{
#sql iterator ByName(String LastName, Date HireDate);
ByName nameiter; // Declare object of ByName class
#sql nameiter={SELECT LASTNAME, HIREDATE FROM EMP};
while (nameiter.next())
{
System.out.println( nameiter.LastName() + " was hiredon "
+ nameiter.HireDate());
}
}
```

Use xxxMetaData method calls infrequently

DatabaseMetaData provides comprehensive information about the database as a whole. We recommend to call it during initialization if needed. Use ResultSetMetaData method calls only once per result set, not once per row/column.

Fetch small amounts of data iteratively

Applications generally require to retrieve huge amounts of data from the database using JDBC in operations such as searching data. If the client requests a search, the application might return the whole result set at once. This process takes lots of time and has an impact on performance. Solutions to this problem are:

- ▶ Cache the search data at the server side and return the data iteratively to the client. For example, if the search returns 1000 records, return data to the client in 10 iterations of 100 records each.
- ▶ Use stored procedures to return data iteratively. This does not use server-side caching. Instead, a server-side application uses stored procedures to return small amounts of data iteratively.

The second solution gives better performance because it need not keep the data in the cache (in memory). The first procedure is useful when the total amount of data to be returned is not huge.

4.2.3 Servlet and JSP performance hints and tips

You can view more optimization techniques on the following Web sites:

<http://www.precisejava.com/javaperf/j2ee/JSP.htm>

<http://www.precisejava.com/javaperf/j2ee/Servlets.htm>

JSPs

JavaServer Pages is the Java platform technology for building applications containing dynamic Web content such as HTML, DHTML, XHTML and XML. The JavaServer Pages technology enables the authoring of Web pages that create dynamic content easily but with maximum power and flexibility. The content presented by JSP is generated dynamically by JVM during execution time.

Use HTML for static content

This sounds obvious, but is often forgotten. JSP is a good way of presenting dynamic content. But it will never be as fast as HTML because of dynamic code generation, compiling, and JVM interpretation. If the content is static, be sure to use HTML instead of JSP.

Use Directive `<%@ include file=...%>` tag to include static content

The content included by this tag is generated at JSP compile time, which improves JSP calling efficiency.

Use action `<jsp:include page= />` carefully

Use this tag carefully because the page included is generated dynamically every time JSP is requested. Use it only for dynamic content.

Set `<%@ page session="false"%>` if page does not participate in session

In JSP, set `<%@ page session="false"%>` if the page is not required to participate in an HTTP session. By default this is set to “true” and the implicit script language variable named “session” of type `javax.servlet.http.HttpSession` is no longer null; it references the current/new session for the page.

HttpSession

Sessions are used to maintain state and user identity across multiple page requests over the normally stateless HTTP protocol. The `javax.servlet.HttpSession` interface is implemented by the servlet engine to provide association between an HTTP client and an HTTP session. There are other possibilities to hold session data; refer to “Choosing the right session mechanism” on page 106.

Avoid storing large objects in HttpSession

Avoiding the storing of large objects in `HttpSession` not only reduces the memory usage, but also reduces the serializing and deserializing cost when session data is persisted.

Large applications require using persistent `HttpSessions`. However, there is a cost. An `HttpSession` must be read by the servlet whenever it is used and rewritten whenever it is updated. This involves serializing the data and reading it from, and writing it to, a database. In most applications, each servlet requires only a fraction of the total session data. However, by storing the data in the `HttpSession` as one large object, an application forces WebSphere Application Server to process the entire `HttpSession` object each time. This involves serializing the data and reading it from, and writing it to, a database.

WebSphere Application Server has `HttpSession` configuration options that can optimize the performance impact of using persistent `HttpSessions`. The `HttpSession` configuration options are discussed in detail in the WebSphere Application Server documentation. Also consider alternatives to storing the entire servlet state data object in the `HttpSession`.

Figure 4-7 compares the relative performance of a sample application with a single object of different sizes. As the size of the objects stored in the HttpSession increases, throughput decreases, mainly due to the serialization cost.

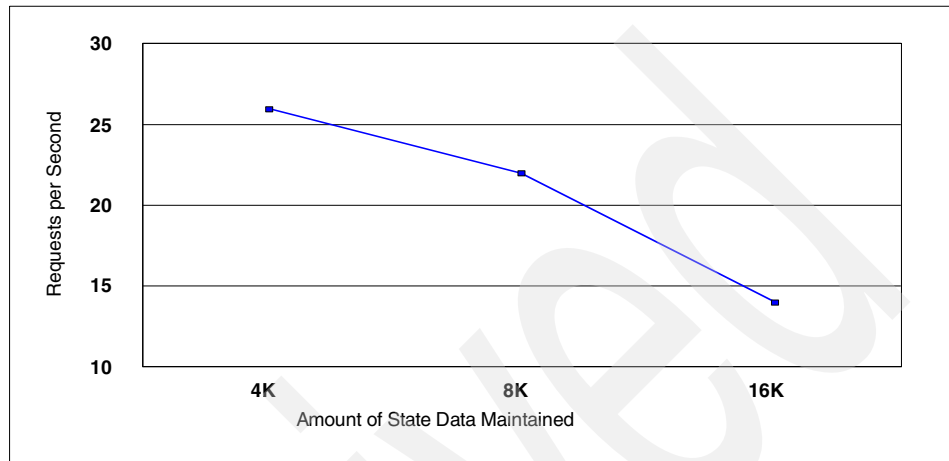


Figure 4-7 Throughput vs. size of objects stored in HttpSession

Use HttpSession invalidate()

Use HttpSession invalidate() to clean up session data when it is no longer needed.

Time out a session

Time out a session more often if possible to release resources. Use the HttpSession.setMaxInactiveInterval() method.

Use HttpSessionBindingListener for complex session objects

If the session object is to store complex beans and hold resources, use the HttpSessionBindingListener interface with these beans and implement the valueUnbound() method to release resources. The servlet container calls valueUnbound() when HttpSession.removeValue() is called. The following is an example:

Example 4-29 The HttpSessionBindingListener interface

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionBindingServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse res)
```

```

        throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    // Get session object
    HttpSession session = req.getSession(true);

    // add MyBindingListener
    session.putValue("AccountBindings.listener", new
        MyBindingListener(getServletContext()));
    }
}

class MyBindingListener implements HttpSessionBindingListener {
    ServletContext context;

    public MyBindingListener(ServletContext context) {
        this.context = context;
    }

    public void valueBound(HttpSessionBindingEvent bindingEvent) {
        System.out.println("Bound object - " + event.getName() + " to
            session-" + bindingEvent.getSession().getId());
    }

    public void valueUnbound(HttpSessionBindingEvent bindingEvent) {
        System.out.println("Unbound object - " + bindingEvent.getName() +
            " from session-" +
            bindingEvent.getSession().getId());
    }
}

```

Choosing the right session mechanism

We use a session mechanism to maintain the client state across multiple pages. The session starts when the client, such as a browser, requests for a URL to the Web server and ends when the Web server ends the session, or times out the session, or the user logs off or closes the browser. The application may hold session data like a shopping cart related to each client.

There are a few approaches available for maintaining session data. These are:

- ▶ Session (implicit) object in memory available for any servlet or JSP (this is HttpSession object provided by the servlet API)
- ▶ Hidden HTML fields
- ▶ Cookies (the Web container uses the JSESSIONID cookie to identify the client, but here we mean holding other client-related data in cookies)

- URL rewriting
- Persistent mechanism

It is difficult to select one mechanism for holding session data from this list. Each approach has an impact on performance depending on the amount of data to be stored as session data, and the number of concurrent users.

Table 4-3 gives an idea of the performance of each approach.

Table 4-3 Comparison between holding session data approaches.

Session mechanism	Performance	Remarks
Servlet session in memory	good	There is no limit on the size of keeping session data.
Hidden fields	moderate	There is no limit on the size of keeping session data.
Cookies	moderate	There is a limit on the size of keeping session data.
URL rewriting	moderate	There is a limit on the size of keeping session data.
Persistent mechanism	moderate to poor	There is a limit on the size of keeping session data.

Here the persistent mechanism means that you store the session data in the database, file storage, or any other persistent storage. There are two approaches for this mechanism:

- Using your application server's persistent mechanism for session data storage
- Using your own persistent mechanism by maintaining your own database schema

If you use the first approach, WebSphere for z/OS converts the session objects into a varchar data type and stores it in the database, which has a size limitation. If you use the second approach, you need to design the schema as per your session fields and need to store the session data by writing JDBC code for that, which may give better performance than the first approach.

Both of these persistent mechanisms give moderate to poor performance compared to other approaches because of overhead involved in database calls through JDBC. It makes calls to a database on every request in order to store that session data, and finally needs to retrieve the whole session data from the database. On the other hand, it scales well upon increasing session data and concurrent users.

URL rewriting gives moderate performance because the data has to pass between the client and server for every request, so an overhead is involved on the network for passing data on every request. Also, there is a limitation on the amount of data that can pass through URL rewriting.

Cookies also give moderate performance because they need to pass the session data between client and server. They also have a size limit of 4k for each cookie.

Like URL rewriting and cookies, hidden fields need to pass the data between client and server and give moderate performance. All of these three session mechanisms are inversely proportional to the amount of session data.

Unlike these mechanisms, the session (implicit) object in the memory mechanism gives better performance because it stores the session data in memory and reduces overhead on the network. Only the session ID (JSESSIONID cookie) is passed between client and server. But it does not scale well on increasing session data and concurrent users because of an increase in memory overhead and also in overhead on garbage collection. Also, sharing of session data in memory between members of a cluster is impossible.

Remember that choosing the session mechanism out of one of the above approaches not only depends on performance, but also scalability and security. The best approach is to maintain a balance between performance, scalability and security. A mixture of session mechanism and hidden fields gives both performance and scalability. By putting secure data in a session mechanism and non-secure data in hidden fields you can achieve better security.

Threading and synchronization

Avoid synchronization in servlet to reduce potential bottleneck

The servlet engine is a multi-threaded runtime environment. Problems associated with concurrency issues occur and sometimes synchronization is used to solve the problem. If synchronization has to be used, minimize the code section to be synchronized on. The synchronization mechanism involves lock acquiring and releasing, which is expensive in itself.

Frequent use of synchronization results in frequent context switching on the thread when lock acquiring fails. Synchronization on large code segments results in longer lock waiting. Example 4-30 synchronizes the major code path of the servlet processing to protect the instance variable `numberOfRows`. Example 4-31 on page 109 moves the lock to a variable and out of the critical code path. It shortens the locking time. This is the recommended way for synchronization.

Example 4-30 Synchronizing on a large code block

```
public class SyncExample1Servlet extends HttpServlet
```



```

{
    private int numberOfRows = 0;
    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
        ...
    }
    public void doGet(HttpServletRequest req, HttpServletResponse res)
    {
        int startingRows = 0;

        try
        {
            Synchronized(this)
            {
                // business logic
            }
        }
        catch (Exception e)
        {
            // ...
        }
    }
}

```

Example 4-31 Synchronizing on a short code block

```

public class SyncExample2Servlet extends HttpServlet
{
    private int numberOfRows = 0;
    private Object lockObject = new Object();
    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
        ...
    }
    public void doGet(HttpServletRequest req, HttpServletResponse res)
    {
        int startingRows = 0;
        Synchronized (lockObject)
        {
            startingRows = numberOfRows;
        }
        try
        {
            {
                // business logic
            }
        }
    }
}

```

```

        catch (Exception e)
        {
            ...
        }
    }
}

```

Figure 4-8 shows a performance comparison.

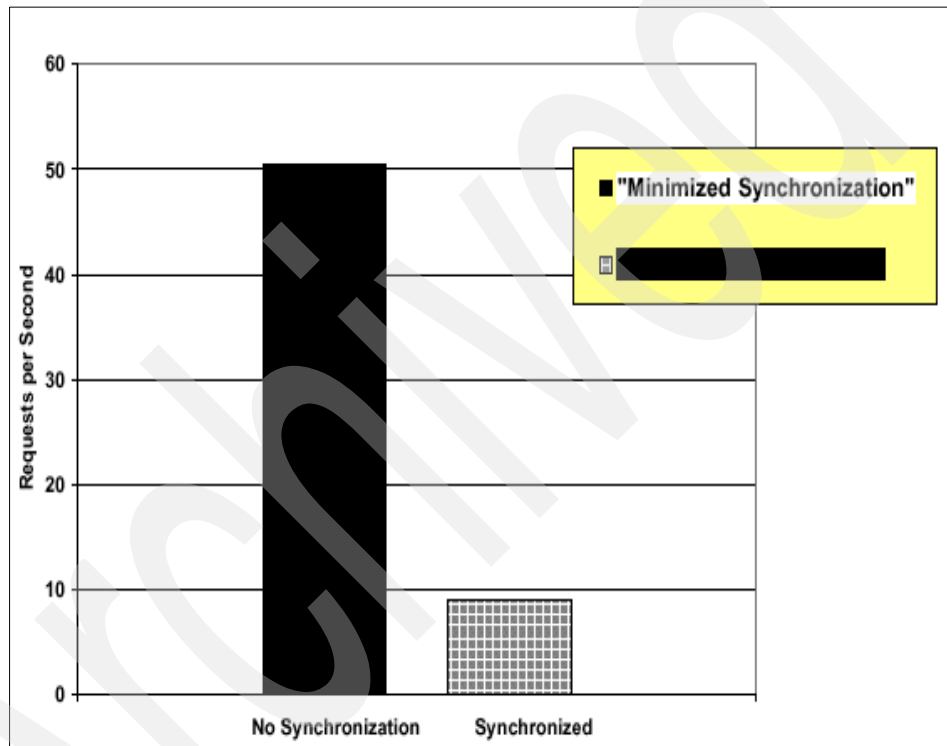


Figure 4-8 Synchronization performance comparison

Use `javax.servlet.SingleThreadModel` carefully

`SingleThreadModel` is a tag interface that a servlet can implement to transfer its reentrancy problem to the servlet engine. The WebSphere servlet engine handles the servlet's reentrancy problem by creating separate servlet instances for each user. Because of the great amount of system overhead, `SingleThreadModel` should be avoided.

Developers typically use `javax.servlet.SingleThreadModel` to protect updatable servlet instances in a multithreaded environment. The better approach is to avoid using servlet instance variables that are updated from the servlet's service method.

ServletRequest.getRemoteHost() is very inefficient

It can take seconds to complete the reverse DNS lookup it performs. Try to avoid it if possible.

Use OutputStream instead of PrintWriter

OutputStream can be faster than PrintWriter. JSPs are generally slower than servlets when returning binary data, since JSPs always use a PrintWriter.

Choosing custom tags versus no custom tags

Custom tags in JSP give you reusability and simplicity. Simplicity means that you need not write Java code in JSP, rather you write custom tags for that. Reusability means that once you write a piece of code as custom tag handler, you can use this tag handler in any JSP.

But what will happen if you write a tag handler that is not reused often and is not simple? In such cases it is better not to use custom tags since you need to use classes, interfaces of `javax.servlet.jsp.tagext`, deployment descriptor files, and you also need to override methods of those classes and interfaces in order to write a tag handler. The JSP engine has to look at the descriptor file to figure out the tag handler class and execute that handler.

All these operations do not come for free. They reduce performance, which is proportional to the number of tag handlers you use in JSP. So don't use custom tags unless you are sure of their reusability.

Avoid using too many levels of body tags

Using multiple levels of body tags combined with iteration will likely slow down the processing of the page significantly.

Take advantage of the Servlet.init() method

Use the `init()` method of `HttpServlet` to perform operations that only need to be done once, for example, reading property files, initializing static variables, etc. The `HttpServlet.init()` method is only called once in the life cycle of a servlet, and can be called at the servlet container startup time if so configured.

Minimize logging

Minimize the use of `System.out`, `System.in` and `System.err` to reduce logging. In WebSphere Application Server on z/OS, these logs can be disabled by configuration.

4.2.4 EJB programming hints and tips

You can view more optimization techniques on the following Web site:

<http://www.precisejava.com/javaperf/j2ee/EJB.htm>

Context lookup

A distributed application's components need to locate one another so that they can work together. Therefore, distributed applications require something to help the components find each other. The Java Naming and Directory Interface (JNDI) provides just this capability.

JNDI

A naming service maintains a set of bindings. These relate names to objects. Clients use the naming service to locate objects by name. There are a number of existing naming services such as LDAP, DNS etc. They each follow the above technique, but differ in the details.

A directory service manages the storage and distribution of shared information. A directory service manages a directory of entries. An entry can refer to a name, service, or almost any other concrete object or abstract concept. An entry also has attributes associated with it such as a name or identifier and one or more values. These attributes describe the entry.

JNDI is an interface that accesses an existing naming service (such as an LDAP service). This allows JNDI to integrate seamlessly into an existing computing environment where an existing naming service provides service. However, there can be significant overhead in doing a JNDI lookup over the network and Java programs need to optimize JNDI lookup for performance.

Keep the JNDI name tree as flat as possible

It takes longer to retrieve deeper name trees.

Cache references

In J2EE servers, the naming service is often used to look up EJB home interface, DataSource and other object references. Each EJB has a home interface (the interface used to find the EJB) and requires a separate lookup. Because JNDI lookup is an expensive operation, frequent JNDI lookup adds significant overhead. Caching references for later use can significantly improve performance.

For simple applications, it might be enough to acquire the EJB home in the servlet `init()` method. Example 4-32 shows how to acquire an EJB home in the `HttpServlet.init()` method. This is consistent with the best practice of using the `HttpServlet Init()` method to perform expensive operations that need only be done once.

Example 4-32 Acquiring the home interface

```
public class MySessionEJBServlet extends HttpServlet
{
    private MySessionManagerHome seHome = null;
    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
        javax.naming.Context ctx = null;
        try
        {
            java.util.Hashtable env = new java.util.Hashtable();
            env.put(Context.INITIAL_CONTEXT_FACTORY,FACTORY);
            ctx = new InitialContext(env);

            Object homeObject = nameContext.lookup("EJB JNDI NAME");
            seHome = (MySessionManagerHome)
                javax.rmi.PortableRemoteObject.narrow(
                    homeObject, MySessionManagerHome.class);
        }
        catch (Exception e)
        {
            throw new ServletException("init error" + e.getMessage());
        }
        finally
        {
            try
            {
                if (ctx != null)
                    ctx.close();
            }
            catch (Exception e)
            {
                //handle this exception
            }
        }
    }
}
```

More complicated applications might require cached EJB homes in many servlets and EJBs. One possibility for these applications is to create an EJB home locator and caching class. Example 4-33 shows code that can be used to cache an EJB home.

Example 4-33 Caching EJB home interface

```
public static EJBHome lookupHome(String jndiName,
    Class homeClassName) {

    // Local Variables
    EJBHome home;
    // Check whether the home is already in the cache.
    home = (EJBHome) homeCache.get(jndiName);
    if (home != null) {
        // Home is already in Hashtable.
        return home;
    }

    // Time to Lookup the new home. The Initial Context
    // has already been retrieved at class initialization.
    if (_initContext == null) {
        return null;
    }
    try {
        // Lookup the home.
        Object homeObject = _initContext.lookup(jndiName);
        home =
        (EJBHome)javax.rmi.PortableRemoteObject.narrow(homeObject, homeClassName);
        synchronized (homeCache) {
            homeCache.put(jndiName, home);
        }
    } catch (javax.naming.NamingException ne) {
        System.err.println("[EJBHomeFactory] - Lookup Failed" +
            ne.getMessage());
        return null;
    }
    return home;
} // end lookupHome
```

Invalid cached home interfaces

One problem with caching and reusing EJB homes involves handling invalid cached homes. If a client caches an EJB home and then the container shuts down and then returns, the client's cached value is invalid and will not work. Accessing an invalid cached home interface will result in an exception. Note that the code will need to check for invalid home interfaces.

Session beans

There are two types of session EJBs, stateful and stateless. From the client's view, a stateful session bean retains conversational data between method calls and across transactions. A stateless session bean does not have the data that needs to be maintained beyond the scope of a single-method call. Here is some advice regarding session beans:

- ▶ Use stateless session EJBs instead of stateful session EJBs.
This helps maximize application scalability because a stateless session bean does not maintain session data in memory or serialize it to disk or database.
- ▶ Access entity beans from session beans.
Avoid accessing EJB entity beans from client or servlet code. Instead wrap and access EJB entity beans in EJB session beans. This best practice satisfies two performance concerns: reducing the number of remote calls and providing an outer transaction context for the entity bean.
- ▶ Call `remove()` on stateful session beans when finished.
Stateful session beans have affinity to specific clients. They stay in containers until removed by a client or timed-out by a container. A container may also passivate a session bean and serialize its state to disk in order to obtain more resources. When a new request arrives for the passivated session bean, the container activates the session bean by deserializing the session state from disk. To reduce serialization overhead and release resource, explicitly calling `remove()` on the session bean when finished can reduce the cost of passivation and activation and make more resource available for the container.
- ▶ Do not duplicate session information in `HttpSession` and a stateful session bean.

Transactions

- ▶ Use an appropriate transaction isolation level.

The four transaction levels of EJBs are:

- `TRANSACTION_READ_UNCOMMITTED`
- `TRANSACTION_READ_COMMITTED`
- `TRANSACTION_REPEATABLE_READ`
- `TRANSACTION_SERIALIZABLE`

Higher isolation levels require more and longer locking on database tables or rows, hence the performance is not as good as for lower transaction isolation levels. Certainly, data access integrity has to be considered first.

- ▶ Choose the right transaction attribute.

The six transaction attributes of EJBs are:

- NOT_SUPPORTED
- REQUIRED
- REQUIRES_NEW
- MANDATORY
- SUPPORTS
- NEVER

For example, started transactions can be kept short by setting the transaction attribute of the bean to “REQUIRES_NEW”. In this case the running transaction is suspended and the invoked bean is started on a new transaction. When the invoked bean finishes, the original running transaction continues.

Miscellaneous

- Bean instantiation.

`bean.instantiate()` incurs a file system check to create new bean instances. Use “new” to avoid this overhead.

- Group multiple small remote calls into one.

Group multiple response data into one big object to reduce the number of remote calls, cache the result at the client side. For example, instead of calling all *getter* methods on the bean remotely, implement an access method or data object that can retrieve all or most needed data. Cache the result at the client site for subsequent requests.

- Use Read-Only methods where appropriate.

When setting the deployment descriptor for an EJB entity bean (found on the +ReadOnly tab in the AAT), you can mark getter methods as Read-Only. If a transaction unit of work includes no methods other than Read-Only designated methods, then the entity bean state synchronization will not invoke store.

Data access

This chapter introduces our sample application OnlineBuying. It will examine the differences between container-managed persistence versus bean-managed persistence, and between using SQLJ versus JDBC.

At the end of the chapter, we deal with stateful and stateless EJBs.

5.1 The sample application: OnlineBuying

This is a simple online buying application. Users are prompted for a user id and password. New user ids are automatically created for new users.

To familiarize yourself with the application, start by viewing the ProductCatalog. You will be asked for a login/password. Enter whatever you like, as this user will be created on the fly. From the product catalog page, add some items to an order, and click **Order Status** to view all your orders; the current status of the order should be Opened. Then, click **Order Details** to view the current order details.

From there, you can either submit or cancel your order. Submit the order and go back to the Order Status to check that the status has now changed to Submitted.

Now you know enough about the application, so let's review how it has been built. The Product Catalog, Order Details and Order Status buttons display the respective pages.

This application uses four Entity EJBs: PRODUCT, CUSTOMER, ORDER, ORDERLINEITEM. These are mapped to the following four tables in a DB2 database.

Table 5-1 Tables and fields of the database

PRODUCT	CUSTOMER	ORDER	ORDERLINEITEM
SKU	CUSTOMERID	ORDERID	ORDERID
DESCRIPTION	OPENORDER	CUSTOMERID	PRODUCTID
		STATUS	QUANTITY

The application uses a Session EJB OnlineBuying that accesses these Entity EJBs.

The Web container has four servlets: OnlineBuyingServlet, OrderDetailsServlet, OrderStatusServlet and ProductCatalogServlet, as well as JSPs used to present the “view”. The servlets ProductCatalogServlet, OrderStatusServlet, OrderDetailsServlet inherit from OnlineBuyingServlet, which factorizes the three servlets’ common behavior. Whenever a request is made to GET data, the doGetXXX method is invoked. Whenever a request to UPDATE data is made, a HTTP POST type request is made and the doPostXXX method is called.

We now test the OnlineBuying application for performance analysis. We will use Java profiling tools to trace the execution threads and profile performance.

5.2 Cache data to reduce remote calls

As discussed in “Object caching” on page 79, caching objects to avoid many remote calls can significantly improve performance. In this section, to study the performance gain, we test two different releases of the OnlineBuying Application: ejb20 and ejb21.

A *remote* call is a call that occurs between the Web container and the EJB container, or between two different EJB containers. The EJB containers can be on different virtual machines. Figure 5-1 shows the interactions between the stub and the skeleton classes.

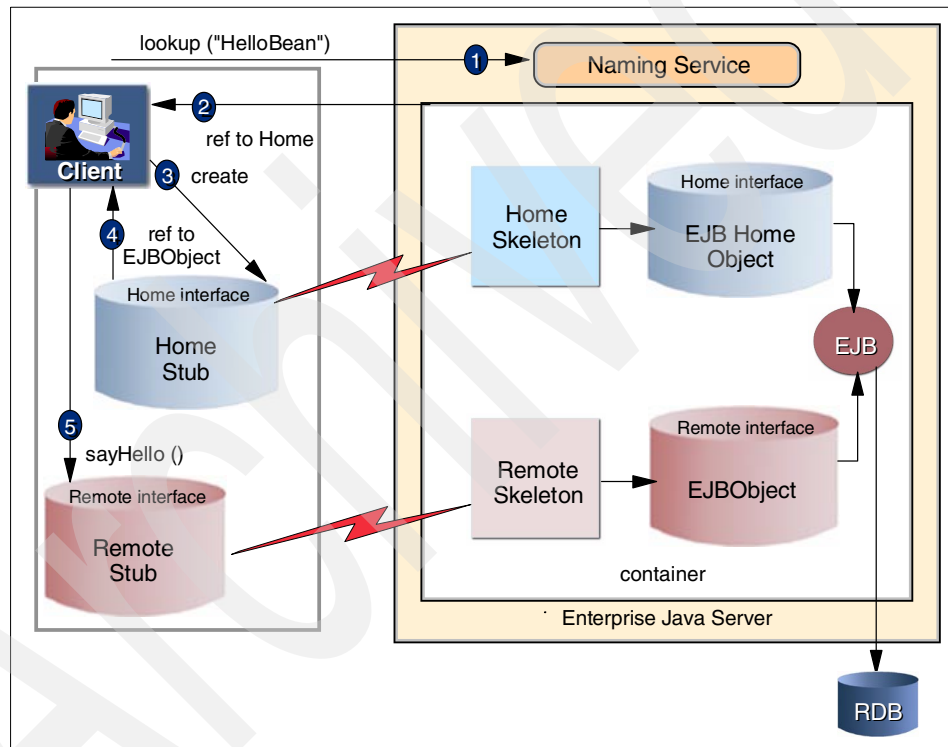


Figure 5-1 Interactions between the stub and skeleton classes

OnlineBuying version 20

In OnLineBuying Application version 20, we delegate the persistence to the EJB container and simplify the access beans' business logic. Therefore, we have four entity beans (Customer, Order, OrderLineItem, Product) with container-managed persistence and custom finders.

The different back-end access beans have been modified to use the entity beans for persistence handling. The new object interface diagram (OID) looks like Figure 5-2.

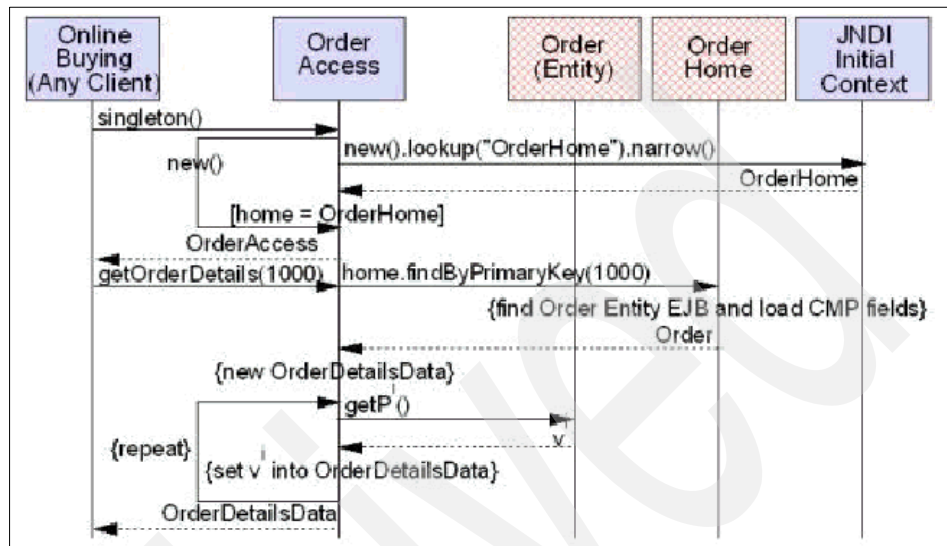


Figure 5-2 Object interface diagram of version 20

OnlineBuying Version 21

In this step, we implement copy helper methods for each of the entity beans. We now have a simpler OID: the number of interactions between the access bean and the EJBs has been reduced to a few method calls; see Figure 5-3 on page 121.

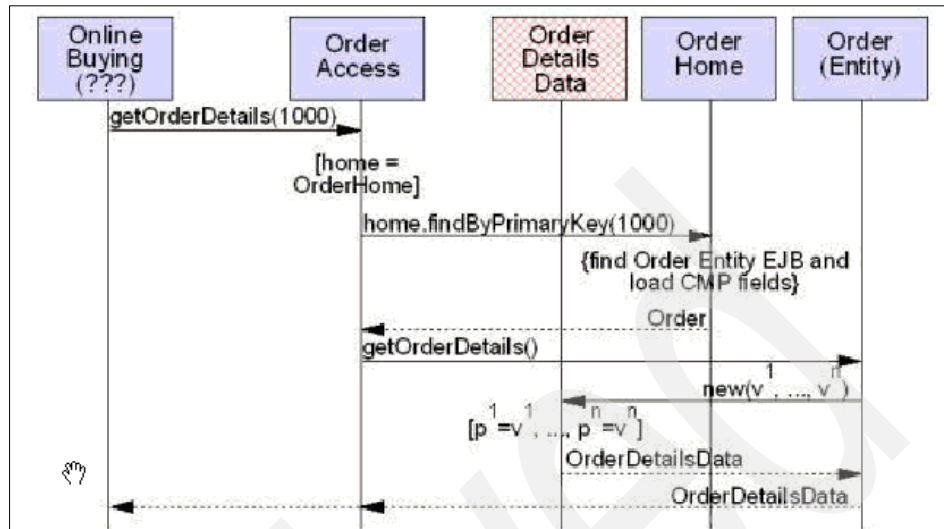


Figure 5-3 Object interface diagram of version 21

This copy helper will provide a single method (such as `getOrderData()` on `Order`) to retrieve all data. In general, restrict remote calls after `create()` and use only one remote call in case of a `find`, such as in the code shown in Example 5-1.

Example 5-1 Implementing copy helper

```

try {
    // Get catalog data from database
    Enumeration result = home.findByCustomerID(customerID);
    Object o = null;
    Order order = null;
    OrderData orderData = null;
    while (result.hasMoreElements()) {
        // Get data from record
        o = result.nextElement();
        order = (Order) javax.rmi.PortableRemoteObject.narrow(o, Order.class);
        orderData = order.getOrderData();
        v.addElement(orderData);
        // while
    } // try
}

```

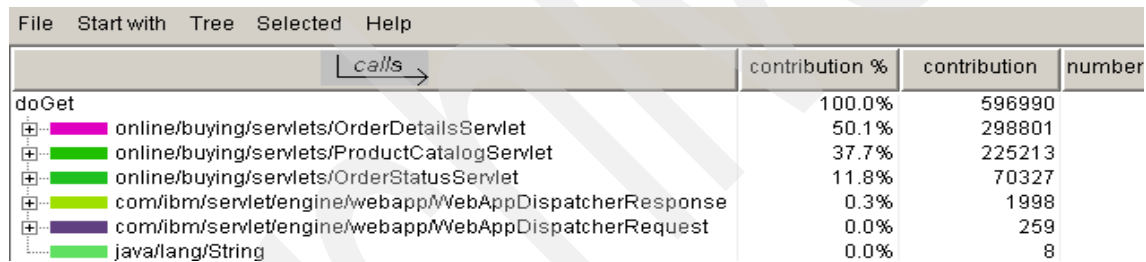
Now you can run a test scenario.

Test scenario 1

1. Set up the WebSphere Application Server with the test application to write a JInsight trace.
2. Start the JInsight trace and conduct the test run:
 - a. Load the home page (OnlineBuying.html) and click **Product Catalog**.
 - b. It launches the Login Screen, and login executes the OrderDetailsServlet.
 - c. Add a sample quality to the product “Dog food bowl”.
 - d. Add a sample quality to the product “Cat food bowl”.
 - e. Click **Order Details**.
 - f. Modify a quantity value.
 - g. Submit the order.
 - h. Click **OrderStatus**.
3. Stop the JInsight trace and analyze the trace.

Test results and analysis

The first measurement is the difference between the two implementations of the OrderDetailServlet. Figure 5-4 shows an extract from JInsight trace:



	calls	contribution %	contribution	number
doGet		100.0%	596990	
online/buying/servlets/OrderDetailsServlet		50.1%	298801	
online/buying/servlets/ProductCatalogServlet		37.7%	225213	
online/buying/servlets/OrderStatusServlet		11.8%	70327	
com.ibm.servlet.engine.webapp.WebAppDispatcherResponse		0.3%	1998	
com.ibm.servlet.engine.webapp.WebAppDispatcherRequest		0.0%	259	
java/lang/String		0.0%	8	

Figure 5-4 OnlineBuying version EJB20

On this measurement, we can note the cumulative time for each of the servlets in the application. Here, the OrderdetailsServlet is accessing the data in DB2 OrderLineItem table. The total time represents about 50% of the requests coming from the main servlets—here, the OnlineBuyingServlet—with the EJB20 release (as shown in Figure 5-4), and 41% with the EJB21 (as shown in Figure 5-5 on page 123).

We can note the difference for the same numbers of calls (28). Avoiding remote calls and caching data is improving the time spent in the method call.

File Startwith Tree Selected Help			
calls		contribution %	contribution
doGet		100.0%	660360
+	online/buying/servlets/ProductCatalogServlet	48.0%	316816
+	online/buying/servlets/OrderDetailsServlet	41.4%	273373
+	online/buying/servlets/OrderStatusServlet	10.3%	67717
+	com.ibm.servlet.engine.webapp.WebAppDispatcherResponse	0.3%	1861
+	com.ibm.servlet.engine.webapp.WebAppDispatcherRequest	0.1%	361
+	java/lang/String	0.0%	9

Figure 5-5 OnlineBuying version EJB21

In this test scenario the servlet and the EJB are in the same JV, so a local RMI/IIOP call is used. The overhead for remote calls is low in this case (however, this may not always be the case); therefore, we observe only a marginal improvement in performance.

Important: This test analyzed only a single thread of execution. In a production environment with several competing threads, the performance gain can be significantly higher.

5.3 CMP versus BMP

The OnlineBuying Application has four entity beans: Product, Customer, Order and OrderLineItem. We now convert the Product Bean from CMP to BMP, and conduct a performance test.

Steps to convert the Product CMP EJB to a BMP

1. Generate a BMP entity EJB with VisualAge for Java.
2. VisualAge for Java will generate the enterprise Javabean methods, such as ejbCreate, ejbLoad, ejbStore, etc.
3. Implement these methods using SQL statements, and change the reference in the corresponding access bean (ProductAccess).

Test scenario 2

1. Now, deploy the application and execute test scenario 2 exactly as described in the previous step.
2. Capture the JInsight trace and analyze the results.
3. We do not find significant improvement with a BMP entity EJB unless this BMP is implemented with SQLJ; this is done in the next step.

Additional considerations for using BMP

Additional development efforts are needed to implement BMP EJBs. Therefore, a good approach is to develop applications with CMP EJBs as a first step (use the profiling tools to profile the application performances), and then consider switching to BMP EJBs, based on performance and response time requirements:

- ▶ BMP Entity EJBs can be used to wrap non-relational backends, or access unsupported legacy databases.
- ▶ They can be designed to cache query results that are small in BMP finders, and reduce database access.
- ▶ They can also be involved in the translation from internal to external form, and back again.
- ▶ They can be used to map an entity EJB to more than one table with SQL JOINS. Alternatively, “views” can be created in the database, and a CMP can be mapped one-to-one to the view.
- ▶ When faced with aggressive development schedules, CMP can be faster to develop.

5.4 BMP with SQLJ

Now, we convert the Product Entity EJB from BMP that uses JDBC, to one that uses SQLJ for database access.

Java classes that use SQLJ need DB2 Plans or Packages to access data in DB2. The DB2 Plan/Package contains the data access strategy that the optimizer has developed during Bind. In general, the process shown in Figure 4-3 on page 92 is necessary to develop Java classes that use SQLJ to access DB2 data.

Implementing BMP with SQLJ

In the following sections, we detail the steps you need to follow in order to implement BMP with SQLJ.

Develop the application

1. Create a BMP entity EJB with VisualAge for Java and copy the generated code to an .sqlj file.
2. Modify this .sqlj file and implement all the methods with the SQL statements needed.
3. Import this .sqlj file into VisualAge for Java.

To do this, in VisualAge for Java, use:

Workspace>Tools>SQLJ>Import (make sure the Perform Translation Box is checked)

VisualAge for Java will generate the .java and .ser files.

4. Now, export all EJBs in the EJB group to an EJB jar file. Make sure that the resource (.ser file) is selected for the export.

Deploy the application

1. Define a DataSource with a specific DB2 Plan that you will Bind in the next step. The access to DB2 data is done with a specific DB2 Plan.
2. Associate the BMP entity EJB with the datasource.
3. Use the SMEUI as usual to deploy the .ear file.

Bind and server configuration

1. Copy the EJBs.jar from /WebSphere390/CB390/apps/<App Server Name>/EJBs.jar into another directory (such as /u/dir/).
2. Extract the files in the EJBs.jar file:

```
jar -xvf TradeEJBs.jar
```
3. Execute the DB2 for Java profile customizer to customize the serialized file:

```
db2profrc -pgmname=PRODUCT ProductBMPBean_SJProfile0.ser
```
4. Since the serialized profiles (*SJProfile0.ser files) are modified by the customization command, rejar the EJBs.jar file and copy it back to the "OnlineBuying" subdirectory.

For example, issue the following commands in the /u/dir/ subdirectory:

```
rm EJBs.jar
jar -cvf EJBs.jar *
cp EJBs.jar /WebSphere390/CB390/apps/<App Server Name>/OnlineBuying"
```

5. After you've customized the serialized profiles for your SQLJ application program, you must bind the DBRMs that are produced by the SQLJ customizer.

You can bind the DBRMs directly into a plan, or bind them into packages and then bind the packages into a plan.

Example 5-2 shows the BIND job. The contents should be modified for your environment.

Example 5-2 Bind job

```
//SQLJBIND JOB (999,P0K),'COPY',CLASS=A,REGION=OK,
//  MSGCLASS=T,MSGLEVEL=(1,1),NOTIFY=&SYSUID
//JOB LIB DD DISP=SHR,
//      DSN=DSN710.SDSNLOAD
//SQLJBIND EXEC PGM=IKJEFT01,DYNAMNBR=20
//DBRMLIB DD DISP=SHR,
//      DSN=JAVA1.DBRMLIB.DATA
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DB2A)

BIND PACKAGE (SQLJV1) QUALIFIER(CBASRU2) -
  MEMBER(PRODUCT1) ISOLATION(UR) VALIDATE(BIND)
BIND PACKAGE (SQLJV1) QUALIFIER(CBASRU2) -
  MEMBER(PRODUCT2) ISOLATION(CS) VALIDATE(BIND)
BIND PACKAGE (SQLJV1) QUALIFIER(CBASRU2) -
  MEMBER(PRODUCT3) ISOLATION(RS) VALIDATE(BIND)
BIND PACKAGE (SQLJV1) QUALIFIER(CBASRU2) -
  MEMBER(PRODUCT4) ISOLATION(RR) VALIDATE(BIND)

BIND PLAN(SQLJV1) -
  PKLIST(DSNJDBC.DSNJDBC1, -
        DSNJDBC.DSNJDBC2, -
        DSNJDBC.DSNJDBC3, -
        DSNJDBC.DSNJDBC4, -
        SQLJV1.PRODUCT1, -
        SQLJV1.PRODUCT2, -
        SQLJV1.PRODUCT3, -
        SQLJV1.PRODUCT4)

END
/*
//SYSIN DD *
GRANT EXECUTE ON PLAN SQLJV1 TO PUBLIC;
//
//
```

6. Store the db2sqljjdbc.properties file in a subdirectory in the HFS (for example, /u/dir/).

Update current.env in WebSphere390/CB390/controlinfo/envfile/<Plex Name>/<Appserver name> with the following information:

DB2SQLJPROPERTIES=/u/dir/db2sqljjdbc.properties

7. In your db2sqljjdbc.properties file, modify the parameters shown in Example 5-3 to fit your environment.

Example 5-3 db2sqljjdbc.properties file

```
DB2SQLJSSID=DB2A
(This is the name of the DB2 subsystem to which a JDBC or SQLJ
application connects)
DB2SQLJPLANNAME=SQLJV1
(This plan name needs to be in your bind job)
(The default name is DSNJDBC)
DB2SQLJDBRMLIB=JAVA1.DBRMLIB.DATA
(The default DBRM data set name is prefix.DBRMLIB.DATA)
DB2SQLJATTACHTYPE=RRSAF
DB2SQLJMULTICONTXT=YES
DB2CURSORHOLD=YES
```

Now you're ready to start your Application Server instance!

Test scenario 3

1. Start your server and execute test scenario 3 exactly as described in the previous step.
2. Capture the JInsight trace and analyze the results.

Test results and analysis

The same trace is captured as in the previous cases; see Figure 5-6 on page 128. In this one, the ProductCatalogServlet results are examined. Note that the time spent in this servlet is around 40%.

You can compare this with the EJB21 release with CMP Product EJB. The result was 48% of the total time spent in the doGet method of the OnlineBuyingServlet.

Attention: This test analyzed only a single thread of execution. In a production environment with several competing threads, the performance gain can be significantly higher.

This servlet is now calling a BMP EJB that uses SQLJ to perform the request into the Product table. The cumulative time to perform this request is lower using SQLJ, instead of JDBC.

File Start with Tree Selected Help				
calls		contribution %	contribution	number of calls
doGet		100.0%	502680	8
+	online/buying/servlets/OrderDetailsServlet	44.5%	223786	21
+	online/buying/servlets/ProductCatalogServlet	40.1%	201367	27
+	online/buying/servlets/OrderStatusServlet	15.0%	75356	7
+	com/ibm/servlet/engine/webapp/WebAppDispatcherResponse	0.4%	1779	24
+	com/ibm/servlet/engine/webapp/WebAppDispatcherRequest	0.0%	245	15
	java/lang/String	0.0%	7	17

Figure 5-6 BMP with SQLJ Version EJB21

It is possible to achieve up to 50% improvement with SQLJ over JDBC. The performance gain with BMP using SQLJ over CMP can be significantly higher. However, due to inadequate tooling it is time-consuming to develop EJBs with SQLJ, and different skills are involved during the deployment process.

Therefore, a good approach would be to develop applications with CMP EJBs as a first step and then consider switching to BMP EJBs based on performance characteristics.

5.5 Stateful vs. stateless session EJBs

A session bean encapsulates nonpermanent data associated with a particular EJB client. Unlike the data in an entity bean, the data in a session bean is not stored in a permanent data source, and no harm is caused if this data is lost. However, a session bean can update data in an underlying database, usually by accessing an entity bean. A session bean can also participate in a transaction.

When created, instances of a session bean are identical, though some session beans can store semipermanent data that makes them unique at certain points in their life cycle. A session bean is always associated with a single client; attempts to make concurrent calls result in an exception being thrown.

For example, the task associated with transferring funds between two bank accounts can be encapsulated in a session bean. Such a transfer session bean can find two instances of an account entity bean (by using the account IDs), and then subtract a specified amount from one account and add the same amount to the other account.

Unlike an entity bean, a session bean does *not* have a primary key class. A session bean does not require a primary key class because you do not need to search for specific instances of session beans.

If a session bean does not need to maintain specific data across methods, it is a *stateless* session bean. Methods in stateless session beans are complete, self-contained operations and do not rely on the state held by the data in an instance of the bean. The bank transfer bean described previously is an example of this.

For stateless session beans, a client can use any instance to call any of the methods belonging to a session bean, because all instances are the same. This makes stateless session beans very scalable.

If a session bean needs to maintain specific data across method calls, it is a *stateful* session bean. When a session bean maintains data across methods, it is said to have a *conversational* state.

A Web-based shopping cart is a classic use of a stateful session bean. As the user adds items to the shopping cart and subtracts items from the shopping cart, the underlying session bean instance must maintain a record of the contents. After a particular enterprise bean client begins using an instance of a stateful session bean, the client must continue to use that instance as long as the specific state of that instance is required.

If the session bean instance is lost before the shopper commits the contents of the shopping cart to an order, the shopper must load a new shopping cart. Figure 5-7 displays the lifecycle of a stateless session bean.

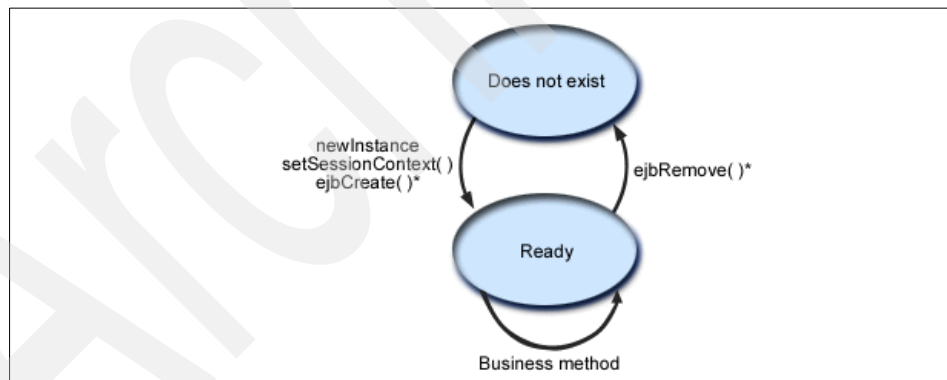


Figure 5-7 Stateless session bean lifecycle

Note: * `ejbCreate()` and `ejbRemove()` methods are called by the container when needed, and do not necessarily correspond to client calls to `create()` and `remove()`.

Because every stateless session bean instance of a particular type is the same as every other instance of that type, stateless session bean instances are not passivated or activated. These instances exist in a ready state at all times until their removal.

The container has a sophisticated algorithm for managing which enterprise bean instances are retained in memory. When a container determines that a stateful session bean instance is no longer required in memory, it invokes the bean instance's `ejbPassivate()` method and moves the bean instance into a reserve pool. A stateful session bean instance cannot be passivated (deactivated) when it is associated with a transaction.

If a client invokes a method on a passivated instance of a stateful session bean, the container activates the instance by restoring the instance's state and then invoking the bean instance's `ejbActivate()` method. When this method returns, the bean instance is again in the ready state. Figure 5-8 displays the lifecycle of a stateful session bean.

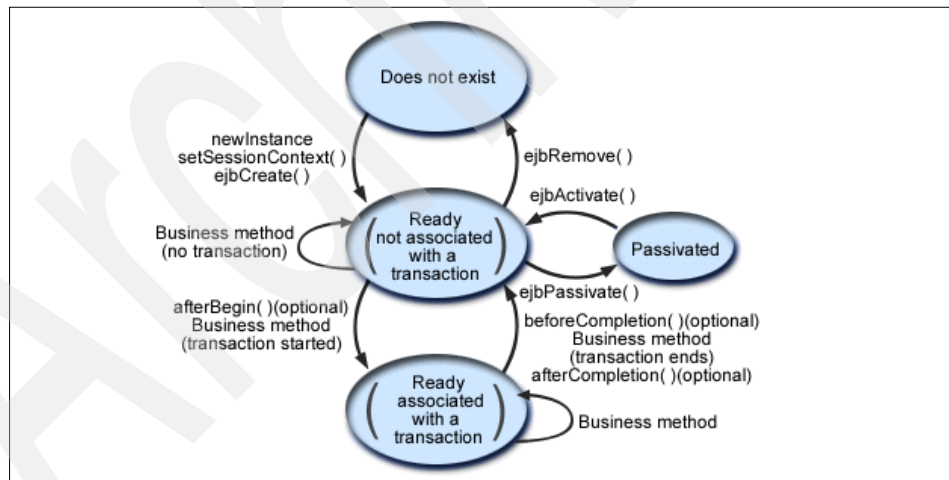


Figure 5-8 Stateful session bean lifecycle

The lifecycle of a stateful session bean ends when an enterprise bean client or the container calls a `remove()` method defined in the bean's home interface or remote interface. In response to this method invocation, the container calls the bean instance's `ejbRemove()` method. The container can end stateless session beans by this method, or it can pool them for later use.

A container can implicitly call a `remove` method on an instance after the lifetime of the EJB object has expired. The lifetime of a session EJB object is set in the deployment descriptor with the `time-out` attribute. See the + Extensions tab in the AAT.

Recommendation

Generally, using stateful session beans involves significantly higher overhead due to the processing time in `ejbActivate()` and `ejbPassivate()` methods used to write and retrieve the data from storage. Avoid stateful session beans in the design where possible.

In the Online Buying application, we changed the stateless session bean *OnlineBuying* to a stateful session bean. In our test scenario, we experienced as much as a 22% decrease in throughput.

Remove stateful session beans when finished. Instances of stateful session beans have affinity to specific clients. They will remain in the container until they are explicitly removed by the client, or removed by the container when they time-out. Meanwhile, the container might need to passivate inactive stateful session beans to disk. This requires overhead for the container and constitutes a performance hit to the application. If the passivated session bean is subsequently required by the application, the container activates it by restoring it from disk. By explicitly removing stateful session beans when finished with them, applications will decrease the need for passivation and minimize container overhead.

Deployment and execution

6.1 Isolation levels

The *isolation level* describes the degree to which the access to a resource manager by a transaction is isolated from the access to the resource manager by other concurrently executing transactions. For session beans with bean-managed transaction demarcation, the Bean Provider can specify the desirable isolation level programmatically in the enterprise bean's methods, using the resource-manager specific API.

For example, the Bean Provider can use the `java.sql.Connection.setTransactionIsolation(...)` method to set the appropriate isolation level for database access. For entity beans using container-managed persistence, transaction isolation is managed by the data access classes that are generated by the container provider's tools.

Here are the meanings of the different levels:

► **Transaction_Serializable**

This level prohibits the following types of reads:

- Dirty reads, where a transaction reads a database row containing uncommitted changes from a second transaction.
- Nonrepeatable reads, where one transaction reads a row, a second transaction changes the same row, and the first transaction rereads the row and gets a different value.
- Phantom reads, where one transaction reads all rows that satisfy an SQL WHERE condition, a second transaction inserts a row that also satisfies the WHERE condition, and the first transaction applies the same WHERE condition and gets the row inserted by the second transaction.

► **Transaction_Repeatable_Read**

This level prohibits dirty reads and non repeatable reads, but it allows phantom reads.

► **Transaction_Read_Committed**

This level prohibits dirty reads, but allows nonrepeatable reads and phantom reads.

► **Transaction_Read_Uncommitted**

This level allows dirty reads, nonrepeatable reads, and phantom reads.

► **Blanks**

Isolation level is not enforced.

For WebSphere Application Server, the isolation level can be set at the bean level or the method level through VisualAge for Java, or set at the method level using WebSphere for z/OS Application Assembly Tool. Figure 6-1 and Figure 6-2 on page 136 demonstrate how to set the isolation level on bean methods.

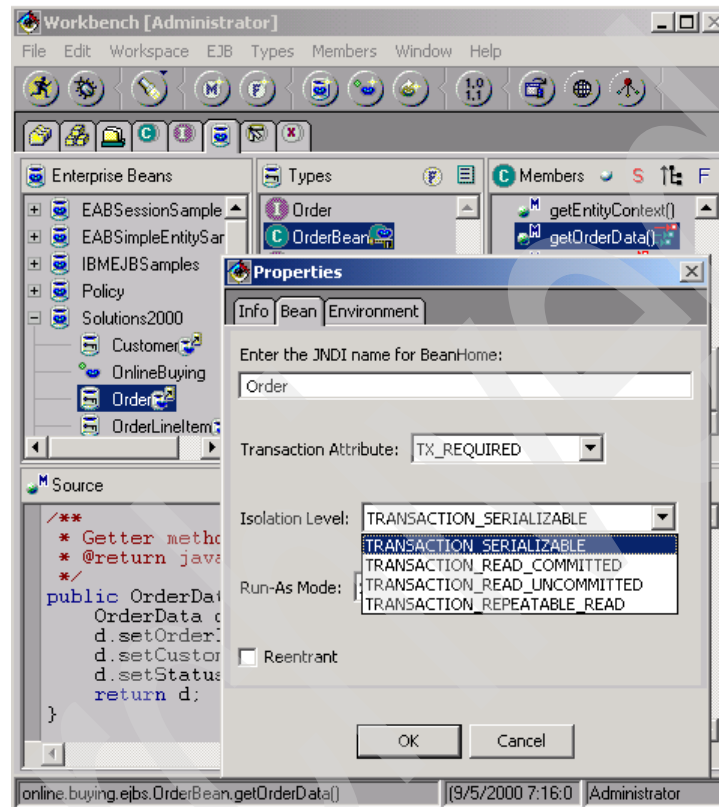


Figure 6-1 Set the isolation level on the EJB method using VAJava

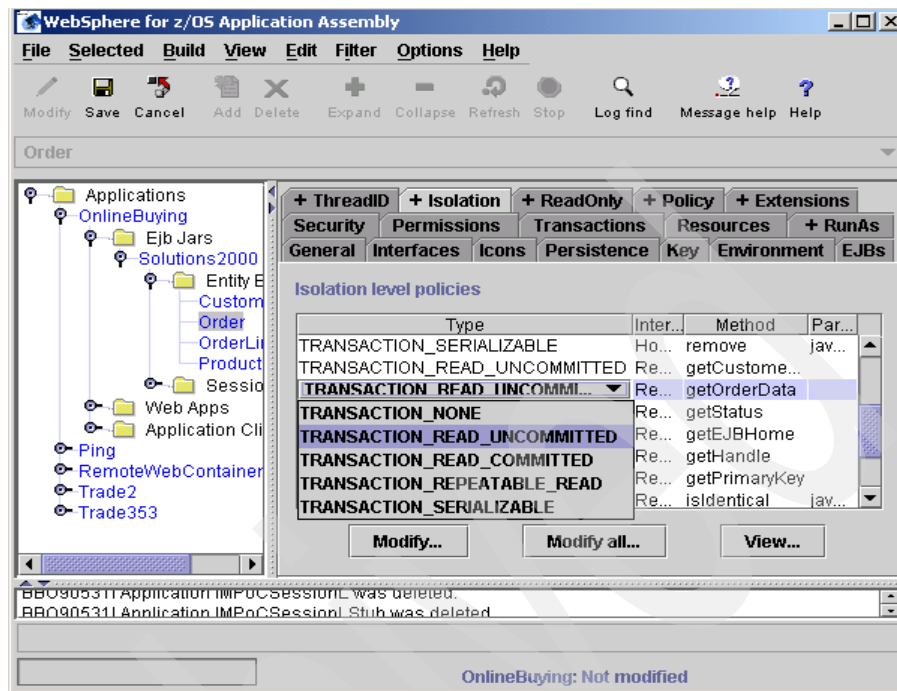


Figure 6-2 Set isolation level on EJB method using AAT

The settings done in VAJava or AAT result in an entry in the META-DATA\application-xdd.xml file in the EJB jar file, as shown in Example 6-1.

Example 6-1 META-DATA\application-xdd.xml file

```

<isolation-level>
<isolation-level-policy>TRANSACTION_READ_UNCOMMITTED</isolation-level-policy>
<method>
<ejb-name>Order</ejb-name>
<method-intf>Remote</method-intf>
<method-name>getOrderData</method-name>
<method-params />
</method>
</isolation-level>

```

In 4.2.2, “JDBC and SQLJ” on page 81, we discussed the difference between these isolation levels on java.sql.connection. The concept is the same on EJB beans and bean methods. In general, a lower isolation level requires fewer locks on the database, and the application performs and scales better, especially under heavy workloads. But it is at the risk of jeopardizing data integrity.

6.2 Read-only methods

In WebSphere, methods in Container Managed Persistent (CMP) entity beans can be set to read-only to optimize end-of-transaction processing. During the course of a transaction, if the only bean methods invoked are read-only, then the EJB container avoids the overhead of storing a bean's essential state back to the database.

Read-only methods can be set in Visual Age for Java (Figure 6-3), or in WebSphere for z/OS Application Assembly Tool (Figure 6-4 on page 138).

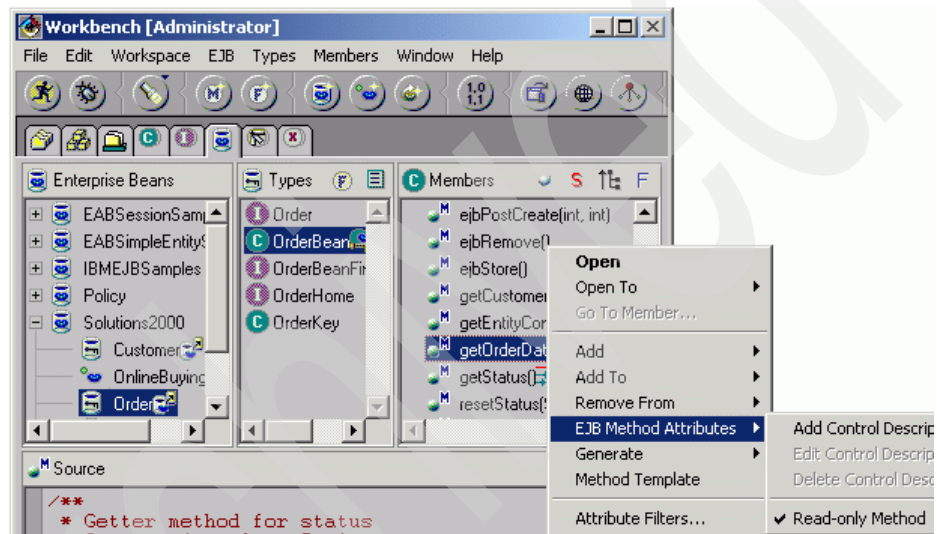


Figure 6-3 Setting read-only methods in VAJava

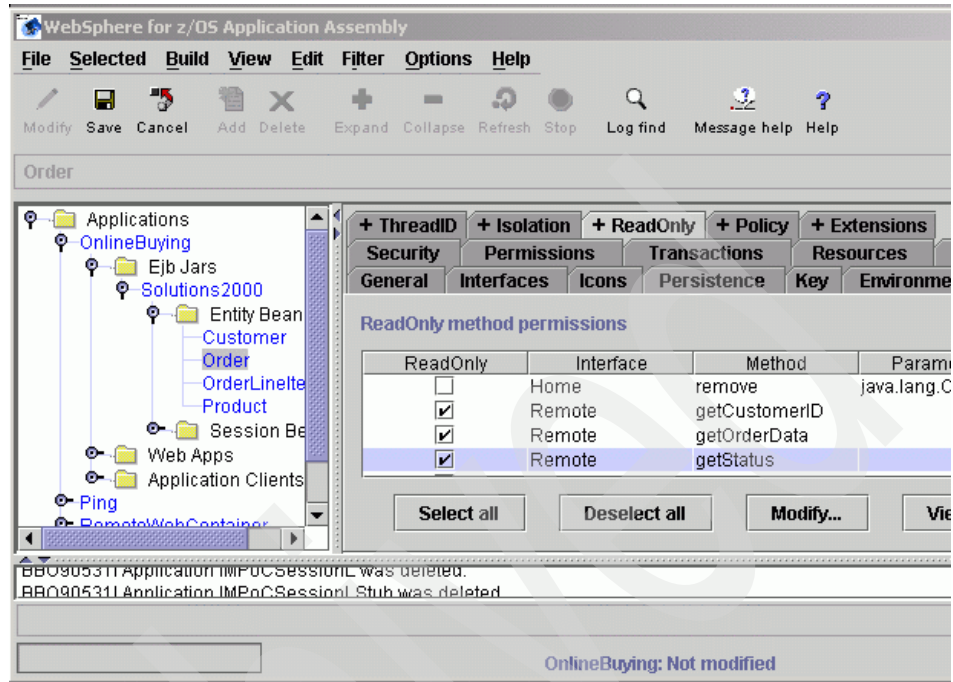


Figure 6-4 Setting read-only in AAT

The settings done in VAJava or AAT result in an entry in the META-DATA\application-xdd.xml file in the EJB jar file.

Example 6-2 Deployment descriptor with read-only attribute

```
<read-only>
  <method>
    <ejb-name>Order</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getOrderData</method-name>
    <method-params />
  </method>
  <method>
    <ejb-name>Customer</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getOpenOrder</method-name>
    <method-params />
  </method>
  <method>
    <ejb-name>Order</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getStatus</method-name>
```

```
<method-params />
</method>
</read-only>
```

We did a test with the OnLineBuying application by enabling and disabling the “read-only” option on the Order CMP EJB’s getOrderData() method. The execution time was slightly shorter when the “read-only” option was on.

6.3 Transaction attribute

A transaction attribute is a value associated with a method of an enterprise bean’s remote or home interface. It specifies how the container must manage transactions for a method when a client invokes the business method via the enterprise bean home or remote interface. Enterprise JavaBeans defines the following values for the transaction attribute: NotSupported, Supports, Required, RequiresNew, Mandatory, and Never.

The definitions are as follows:

- ▶ **NotSupported**
The method cannot be run as a transaction. If the method is called in a transaction context, the container suspends the transaction association before invoking the method. When the method completes, the container resumes the transaction association.
- ▶ **Supports**
The method runs in a transaction if a transaction is already running; otherwise, it runs with no transaction.
- ▶ **Required**
The method must always run in a transaction. If no transaction is running, the container starts one.
- ▶ **RequiresNew**
The method must always run in a transaction, and it must be a new transaction started by the container. If a transaction is already running, that transaction is suspended and a new one started.
- ▶ **Mandatory**
A transaction must already be running when the method is called. If one is not running, the container will not automatically start one and will throw an exception.

► Never

The method will not run in a transaction. If the method is called in a transaction context, the container throws an exception.

For WebSphere Application Server, the transaction attribute is defaulted to Required, and must be manually set if a different value is needed. To set the value, you can use VisualAge for Java and go to the panel shown in Figure 6-1 on page 135. The attributes are propagated through AAT. Another way is to set it in AAT. See Figure 6-5.

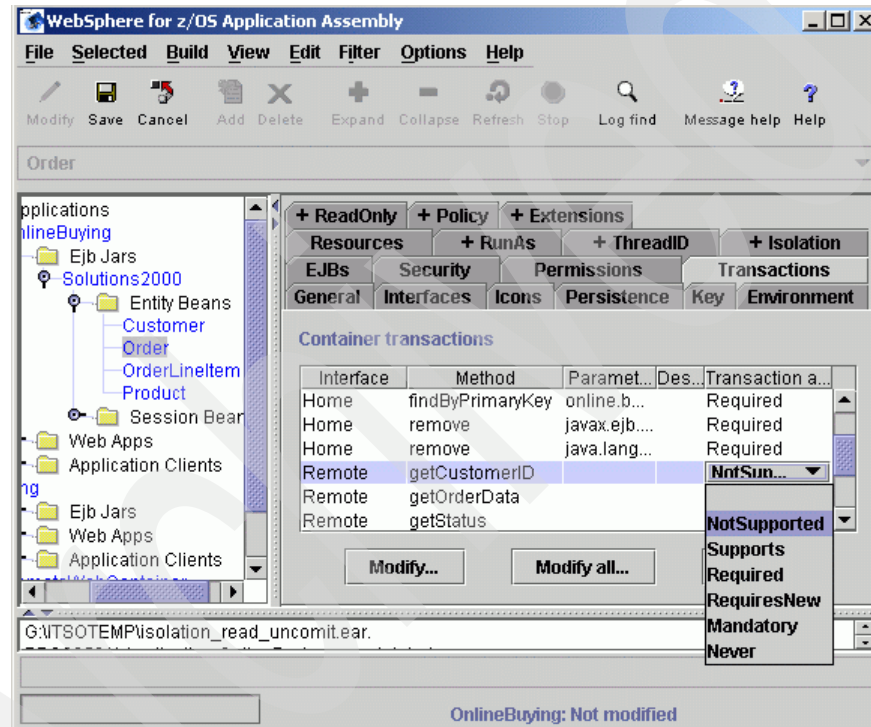


Figure 6-5 Set the transaction attribute using AAT

The attributes set in VAJava or AAT are reflected in META-INF\ejb-jar.xml in the EJB jar file.

Example 6-3 META-INF\ejb-jar.xml

```
<container-transaction>
  <method>
    <ejb-name>Order</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getCustomerID</method-name>
    <method-params />
  </method>
</container-transaction>
```



```

    </method>
    <method>
    <ejb-name>Order</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getStatus</method-name>
    <method-params />
    </method>
    <trans-attribute>NotSupported</trans-attribute>
  </container-transaction>
  <container-transaction>
    <method>
    <ejb-name>Order</ejb-name>
    <method-intf>Home</method-intf>
    <method-name>create</method-name>
    <method-params>
    <method-param>int</method-param>
    </method-params>
    </method>
    <method>
    <ejb-name>Order</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>setStatus</method-name>
    <method-params>
    <method-param>java.lang.String</method-param>
    </method-params>
    </method>
    <trans-attribute>Required</trans-attribute>
    ....
  </container-transaction>

```

In general, if transaction behavior is not required, we recommend that you set the attribute value to `NotSupported` or `Supports` for better performance. For example, to simply fetch a row from a DB2 table, we can use transaction attribute `NotSupported`.

An entity bean can be `reentrant`, and this can have an important impact on transactional management. The `reentrant` property specifies whether or not an entity bean can invoke methods on itself or call another bean that invokes a method on the calling bean.

Only entity beans can be `reentrant`. If an entity bean is not `reentrant` and a bean instance is executing a client request in a transaction context and another client using the same transaction context makes a request on the same bean instance, the EJB container throws the `java.rmi.RemoteException` exception to the second client. If a bean is `reentrant`, the container cannot distinguish this type of illegal loopback call from a legal concurrent call and the bean must be coded to detect illegal loopback calls.

6.4 Load at server startup

Servlets, JSPs and EJBs should be loaded into the server region during server initialization time if configured so. For servlets and JSPs, this is specified using “load-on-startup” elements in the web.xml file packaged in the Web archive file (*.war) file. The load-on-startup element indicates that this servlet should be loaded on the startup of the Web application. The optional contents of these elements must be a positive integer indicating the order in which the servlet should be loaded. Lower integers are loaded before higher integers. If no value is specified, or if the value specified is not a positive integer, the container is free to load it at any time in the startup sequence.

In WebSphere Application Server for z/OS, a positive number indicates the servlet is loaded right after the server region is started or a JSP is precompiled and loaded. This setting can also be done in AAT, as shown in Figure 6-6.

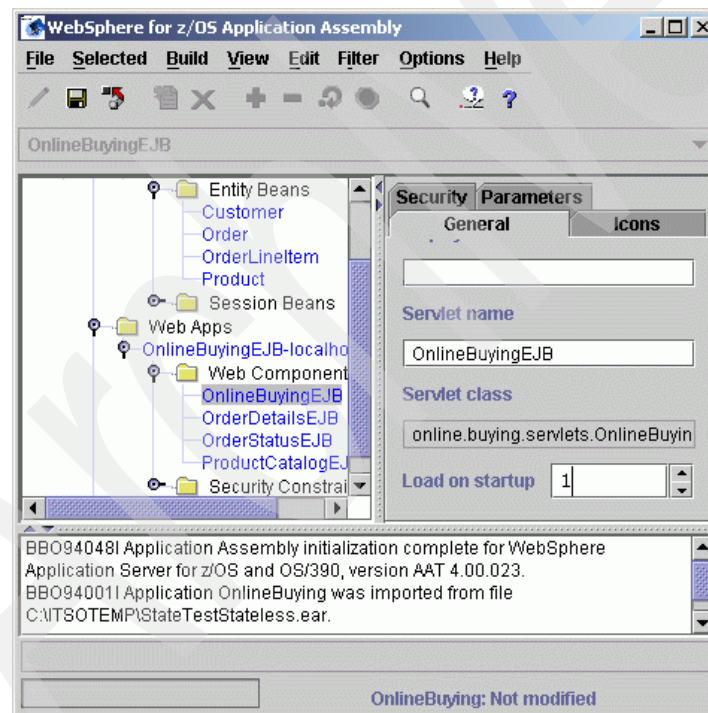


Figure 6-6 Set load-on-startup option in AAT

Configuring servlets and JSPs to be loaded at server start-up time moves the overhead of loading servlet classes and compiling JSPs from runtime to initialization time. This eliminates the long response time problem that the first user request experiences.

6.5 JNDI lookup

WebSphere for z/OS uses the LDAP component of the z/OS or OS/390 Security Server to provide naming and directory services. When your system programmers install and customize WebSphere for z/OS, they set up an LDAP server that uses DB2 tables to store name and directory information.

To access WebSphere for z/OS naming services, J2EE application components and clients set the `java.naming.factory.initial` property to the `com.ibm.websphere.naming.WsnInitialContextFactory` value, and then use JNDI calls to use naming services.

In the WebSphere for z/OS naming system, there are two namespaces:

- ▶ The global namespace, which initially contains home and object references for J2EE application components that are installed in J2EE servers
Your installation may use LDAP access control lists (ACLs) to control access to the global namespace by J2EE servers, application components and clients. The global namespace is accessible by a program in any Java Virtual Machine (JVM), z/OS or OS/390 system, or J2EE server region, as long as the program has appropriate authorization.
- ▶ The local namespace, which contains naming subcontexts represented through `java:comp`
The local namespace is accessible only by the Java programs running within a single JVM and this namespace is always in memory. All J2EE application components that run in the same J2EE server region may access the same local namespace. A single Java client running on z/OS or OS/390 may access its local namespace.

WebSphere for z/OS uses caching to increase the performance of JNDI lookup operations. To speed up subsequent lookups, WebSphere for z/OS caches JNDI context objects as they are bound or initially looked up. Under most circumstances, J2EE application components benefit from the default caching behavior in WebSphere for z/OS. You may change JNDI caching behavior only for the global namespace, not for `java:comp` lookups.

Your installation may only change the caching behavior related to the global namespace, as follows:

- ▶ To change cache behavior for a J2EE server, use a JVM properties file.
- ▶ To change cache behavior for a Java client running on z/OS or OS/390, use one of the following:
 - An environment Hashtable in the client code
 - A jndi.properties resource file

Note: Make sure that this JNDI properties file is on the CLASSPATH before the JNDI resource file supplied with WebSphere for z/OS.

- The Java command line, using the -D switch (not recommended)
- ▶ To clear the Java cache completely:
 - Set this property value:


```
com.ibm.websphere.naming.jndicache.cacheobject=cleared
```
 - Then call `getInitialContext` to clear the cache associated with that context immediately.
- ▶ To have the cache clear itself completely every X minutes:
 - Set this property value to some integer (we'll call it X):


```
com.ibm.websphere.naming.jndicache.maxcachelife
```
 - Then call `getInitialContext` and from then on the cache associated with that context will clear itself every X minutes.
 - To have the cache clear individual entries based on their age, clear entries that are Y minutes old.
 - Set this property value to some integer (we'll call it Y):


```
com.ibm.websphere.naming.jndicache.maxentrylife
```
 - Then call `getInitialContext` and from then on the cache associated with that context will clear entries that are Y minutes or more old.
 - Use JRAS for application logging (and use it wisely).
 - Before writing an event, check to see if the event is enabled.

For more information, see *Assembling J2EE Applications*, SA22-7836.

In addition to the WebSphere caching mechanism, the application affects the performance also. We recommend caching the results from JNDI lookups (Home interfaces) in the code. Save the Homes in variables and reuse these values instead of doing additional lookups.

During the lookups, JNDI traverses the nametree until it finds the requested object. The depth (number of levels) of the tree directly affects the navigation time. Keep the JNDI nametree as flat (lower number of levels) as possible to save navigation time. We conducted tests to time the lookups by varying the location of an object in the nametree. In one run, we placed the object at one level below the root. In the other run, we placed the object 20 levels below the root. The tests showed the lookup of the first object to be 20%-30% quicker than its counterpart 20 levels deeper.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 148.

- ▶ *z/OS for e-business: Introduction to System Tuning*, SG24-6542
- ▶ *e-business Cookbook for z/OS: Java Development*, SG24-5980
- ▶ *e-business Cookbook for z/OS: Infrastructure*, SG24-5981

Other resources

These publications are also relevant as further information sources:

- ▶ *Design Patterns: Elements of Reusable Object-Oriented Software*, ISBN 0-201-63361-2
- ▶ *Assembling J2EE Applications*, SA22-7836
- ▶ *Application Programming and SQL Guide*, SC26-9933
- ▶ *WebSphere for z/OS Operation and Administration*, SA22-7835

Referenced Web sites

These Web sites are also relevant as further information sources:

- ▶ WebSphere for z/OS support page
http://www-3.ibm.com/software/webservers/appserv/zos_os390/support.html
- ▶ IBM patterns website
<http://www-106.ibm.com/developerworks/patterns/>
- ▶ SUN Java J2EE design patterns
http://java.sun.com/blueprints/patterns/j2ee_patterns/index.html
- ▶ Patterns educational page
<http://pages.cpsc.ucalgary.ca/~kremer/patterns/>

- ▶ PreciseJava page about patterns
<http://www.precisejava.com/javaperf/j2ee/Patterns.htm>
- ▶ The Serverside patterns repository
<http://www.theserverside.com/patterns/index.jsp>
- ▶ zSeries Java performance site
<http://www-1.ibm.com/servers/eserver/zseries/software/java/perform.html>

How to get IBM Redbooks

You can order hardcopy Redbooks, as well as view, download, or search for Redbooks at the following Web site:

ibm.com/redbooks

You can also download additional materials (code samples or diskette/CD-ROM images) from that site.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Index

A

AAT 131, 140, 142
Access bean 17
access control list 143
Application Assembly Tool 4, 135, 137
application logic model 8
Application patterns 18
array 75
ArrayLists 75
authorization 143

B

bind 125
BMP 123–124, 127–128
bottleneck 25
bottlenecks 31, 35
business logic 12
Business patterns 17

C

cache 17, 144
caching 143–144
CMP 123–124, 128
Codepage 76
components 11
Composite patterns 18
connectors 6, 11
constructor 70
container 27, 139, 141
Container Managed Persistent (CMP) 137
containers 11–12
Controller 9
controller logic 12
conversational state 16, 129
Copy helper 17
copy helper method 120
CPU utilization 3
current.env 127

D

data integrity 136
DataSource 125

DB2 for Java profile customizer 125
DB2 Plan/Package 124
db2prof 125
db2sqljjdbc.properties file 126
deadlock 27
Dirty read 134

E

EBCDIC 76
EJB 26
EJB container 26, 119
ejbActivate 130–131
ejbPassivate 130–131
ejbRemove 131
EJBROLE 11
Enterprise Information Systems 6
enterprise information systems 12
enterprise model 8
entity bean 14, 128
Exception handling 78

F

Final variables 79

G

Garbage collection 69
garbage collection 25, 27
getter method 16
global namespace 143
Granularity 71
granularity 13

H

hardware technology 6
hashtable 75
HiperSockets 24

I

initilization time 143
Integration patterns 18
Introscope 4

- Agent 38
- Agents 36
- alerts 38
- Console 38
- Enterprise Manager 36, 38
- Explorer 38
- Workstation 36, 38

Introspe 32

isolation level 134

IT component model 8

J

J2EE 7

Java bean wrapper 17

Java Virtual Machine (JVM) 143

JavaBean wrapper 14

JavaServer Page 13

JavaServer Pages (JSPs) 10

JDBC 49, 124, 127–128

JInsight 32, 44, 122–123, 127

Jinsight 4

JNDI 143

- java:comp 143
- jni.properties 144
- lookup 143

JProbe 4, 32, 49

- Garbage Monitor 56
- Java heap 56

JProbe Console 50

JProbe Coverage 49

JProbe Memory Debugger 54

JProbe Profiler 49

JProbe Threadalyzer 49

JRAS 144

JSPs 20

JVM 14, 25, 69, 123

JVM properties file 144

L

latency 20, 23–24

LDAP 143

lifecycle 129, 131

lifetime 131

load-on-startup 142

local interface 14

local namespace 143

Local variables 72

lock 27

logical partition (LPAR) 24

logical tiers 6, 19

Loop optimization 71

loopback call 141

M

memory leak diagnosis 44

Memory usage 3

Model 9

Model View Controller (MVC) 9

model-view-controller (MVC) 19

multi-threaded 77

multi-threading 26

N

namespace 143

nametree 145

network complexity 6

network latency 15

Nonrepeatable read 134

O

object interface diagram (OID) 120

object reuse 69

overhead 27

P

package 125

pathlength 20, 23–24

Patterns 17

patterns 58

- abstract factory 58–59
- adapter 62, 64
- behavioral 58
- creational 58
- facade 62
- flyweight 62
- prototype 58
- proxy 62–63
- singleton 58, 61
- structural 58

performance analysis 31, 44

Phantom read 134

physical tiers 6, 20

pool 131

presentation logic 12

primary key 128

Primitive data types 70
Product mappings 18
profiling tools 32

R

read-only 16
Read-only method 137
ready state 130
Redbooks Web site 148
 Contact us ix
reentrant 141
remote call 15, 119
remote interface 17
remove 131
Response time 3
response time 13, 23
reusable 17
RMI 49
RMI/IIOP 123
Row-set 17
Runtime patterns 18

S

Scalability 25
scalability 16, 25–26
scalable 129
security 6
servlet 7, 13, 17, 26
session bean 14, 27
session handling 25
skeleton 119
SMEUI 125
software technology 6
SQL 123
SQLJ 123–124, 127–128
stateful 27
Stateful session bean 128
stateful session bean 129, 131
stateless 27
stateless session bean 16, 128–129
static final 79
String handling 74
StringBuffer object 74
stub 119
Synchronization 27, 77
synchronized 27, 77
System Management End User Interface (SMEUI)

4

T

thin client 21, 23
threading 25
threads 77
Throughput (requests per second) 3
tiers 19
timeout attribute 131
topologies 21
topology 24–25
transaction 12, 27, 128, 130, 134, 139–140
Transaction_Read_Committed 134
Transaction_Read_Uncommitted 134
Transaction_Repeatable_Read 134
Transaction_Serializable 134
transactions
 Mandatory 139
 Never 140
 Not supported 139
 Required 139
 RequiresNew 139
 Supports 139

U

use case 13

V

vector 75
View 10
Visual Age for Java 137
VisualAge for Java 123–124, 135, 140

W

Web container 26, 119
workload 25

Z

z/OS or OS/390 Security Server 143



Redbooks

Writing Optimized Java Applications for z/OS

Tips for Java optimization

How to use optimization tools

Deployment tips

This IBM Redbook is designed to help you write optimized Java applications for z/OS. It addresses different aspects of performance, and discusses architectures, design patterns, coding techniques, deployment, and data access issues. It applies to WebSphere for z/OS Version 4.0.1 for use with the z/OS Operating System Version 1.1 and above.

Using tips on writing Java-based solutions that have been collected from many different sources, we show how applying many of these tips and measuring performance can help you tune your Java application to perform better on z/OS.

We describe how to use the optimization tools Introscope, Jinsight, and JProbe, and provide recommendations regarding application architecture and Java coding practices. Ultimately, we illustrate how deploying the application on the z/OS WebSphere platform will provide enhanced workload management and high availability, as well as improved scalability and security.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks